Parallel Synchronous Code Generation for Second Round Light Weight Candidates

Pantea Kiaei[1], Archanaa S. Krishnan[2], and Patrick Schaumont[1]

[1]Worcester Polytechnic Institute, Worcester, MA 01609, USA,
{pkiaei,pschaumont}@wpi.edu
[2]Virginia Tech, Blacksburg, VA 24061, USA, archanaa@vt.edu

Abstract. In this contribution we investigate a novel approach to the implementation of the second round light weight crypto candidates. We follow a methodology, called Parallel Synchronous Programming (PSP), which converts the control flow of a crypto-algorithm together with the data processing into bitsliced format. The resulting design executes N parallel versions of a crypto algorithm as a sequence of discrete steps with constant computational effort. The aggregate of these N parallel versions completes in constant-time, not only at the level of the cryptographic kernels, but also over the aggregate of N messages. PSP is especially useful when dealing with algorithms with complex system control such as the multiple phases of AEAD. We present a methodology to map permutations as well as AEAD primitives into PSP form. We present sample results for ASCON-p^{12}, GIFT-128, WAGE permutation, ACE permutation, as well as their AEAD modes. We compare with related work in automatic generation of bitsliced code.

Keywords: ASCON-128, ASCON-p^{12}, GIFT-128, WAGE-$\mathcal{AE}$-128, WAGE permutation, ACE-$\mathcal{AE}$-128, ACE permutation, ARM Cortex-M4, Parallel Synchronous Programming

1 Introduction

To preserve the privacy and security of security-sensitive applications, cryptographic algorithms are required. The authenticated encryption with associated data (AEAD) is a form of encryption that guarantees the confidentiality, integrity, and authenticity of encrypted message and the integrity and authenticity of its associated data. AEAD ciphers are essential in emerging areas, including but not limited to cyber physical systems, medical devices, smart grid, and sensor networks. However, their widespread adoption is hindered by the budget of resource-constrained devices. In 2018, National Institute of Standards and Technology (NIST) published a call for lightweight cryptographic algorithms that are suitable for use in constrained environments [16] which motivated the design of several AEAD ciphers that are suitable for such constrained platforms.

Vulnerabilities of Non-Constant Time Software: Apart from the mathematical sanity of security guarantees, it is important to pay attention to the implementation of cryptographic algorithms. In a physical implementation of any design, there are indirect sources of information that potentially can be used to gain adversarial information about the internal state of ciphers. These sources of information are called side-channels. There is a great effort dedicated in the research community to the expansion of side-channel-related defense techniques.

Timing side-channel is one of such indirect sources of information. A wide array of attacks exists in which the run-time of the program is used to gain information about a running algorithm. This information can be used to gain adversarial information about the the program. For example, the Flush+Reload [19] attack uses the shared cache (L3) access time to break both secret-key [11] and public-key [9, 18] algorithms. As another example, Prime+Probe [17] uses the same concept of cache access time to break cryptographic algorithms when memory space is not shared between the attacker and the victim. Furthermore, the use of table lookups that are secret key-dependent in the implementation of cryptographic algorithms, while beneficial for performance, draws a correlation between the run-time of the program and the secret key of the cryptographic algorithms and can be utilized for the attackers' benefit [8, 15]. This means that at implementation-time, the designer of such secure algorithms must be aware of the security premises and avoid leaving exploitable gaps. However, this is not a straight-forward task and trying to maintain the security of an algorithm, requires tremendous engineering effort and several iterations for each implementation.

Inconstancy of run-time in a program, in general, can have three main causes: First, the implemented algorithm can be data-dependent and this data is measured at run-time. Second, the memory hierarchy present in most processor architectures can cause uncertainty of run-time during software development. Third, access to shared devices on the system can cause contention-based run-time variability. In the following, we discuss existing techniques to address these causes.

Bitslicing: Bitslicing was originally proposed to provide full utilization of the processor's word-length. However, bitslicing has noticeable characteristics that make it attractive for secure applications, specifically for avoiding timing side-channel vulnerabilities. In bitslicing, every data in the program is transposed to a vertical format; In an N-bit processor, each variable is scattered among 1-bit of N memory locations. In such programs, the cache access time does not provide useful information for the attacker. Moreover, by nature, any algorithm in bitsliced format has to be calculation-based meaning no table-lookups are possible.

However, bitslicing has its downsides; In order to implement an algorithm in a bitsliced manner, only a set of basic logic instructions can be used and they do not support control flow computations. The control-related data in the algorithms are therefore excluded from the bitsliced model. The designer has to decide between completely unrolling the algorithm to be able to have an entirely bitsliced design or exclude the control flow from bitsliced format. Furthermore, these properties of bitslicing, make bitsliced implementations resilient to memory hierarchy-based timing uncertainties. However, the data-dependency of the algorithm and its access to shared devices in the system can still cause cause variations in execution time.

Parallel Synchronous Programming: The recently presented programming model, Parallel Synchronous Programming (PSP) [13], overcomes the majority of these downsides of bitslicing. Unlike bitslicing, PSP treats the control-flow the same as any other data in the program and adds the control-flow to the bitsliced presentation. By doing so, PSP software is resilient to the data-dependency-based timing side-channel in addition to providing the security guarantees of bitsliced software.

The main building block of PSP is a core function, *i.e.* PSP kernel, with a completion output signal. The PSP kernel is in bitsliced format, therefore calculates multiple instances of the algorithm in parallel, but includes the control flow as well. This core function should be called consecutively until the completion signal shows that the results are ready. To think about the PSP implementation of an algorithm, is to think of it in terms of a Finite State Machine with Datapath (FSMD). To make this discussion simpler, we choose a simple 4-bit counter as our driving example. To write the PSP implementation of this algorithm, the first step is to define it as an FSMD. The following listing shows the complete implementation of this FSMD in Verilog and Figure 1 shows the schematics of the same. This counter is reset to zero and takes a 4-bit input (`CNT`) as the upper limit to count to. In each clock cycle, the counter will increment by one until it reaches the `CNT` value. Once this value is reached, the output `done` signal is set to one.

```
module counter_v (
    input       clk,    // Clock
    input       rst,    // Synchronous reset active high
    input [3:0] CNT,    // Count destination
    output      done    // done flag
);

reg [3:0] counter_reg;

assign done = (counter_reg == CNT);

always @(posedge clk) begin
    if(rst) begin
        counter_reg <= 4'h0;
    end else begin
        if (!done)
            counter_reg <= counter_reg + 4'h1;
        else
```

```
                counter_reg <= counter_reg;
        end
end

endmodule
```

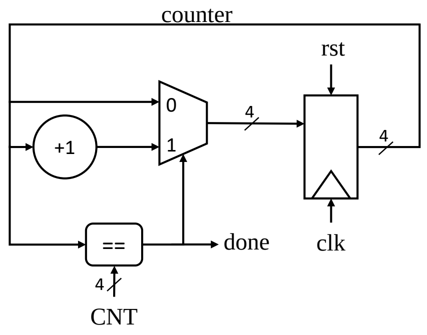


Fig. 1: Schematic view of the 4-bit counter example

The PSP code for this counter, will implement the counter logic with logic operations in the target processor ISA and map the flip-flops to static variable assignments in the C code. The following shows a snippet of the PSP core function that is generated automatically from the Verilog implementation. Every call to this function calculates the same logic as every clock cycle in the Verilog implementation. Furthermore, since the number of operations do not change, every call to this function will take the same amount of time to run regardless of the input values.

```
void counter_c(MDTYPE clk, MDTYPE rst, MDTYPE* CNT, MDTYPE* done)
{
...
  NOT1(rst, n01_);
  XOR2(counter_reg[0], CNT[0], n02_);
  XOR2(CNT[1], counter_reg[1], n03_);
  OR2(n02_, n03_, n04_);
  XOR2(CNT[3], counter_reg[3], n05_);
  XOR2(CNT[2], counter_reg[2], n06_);
  OR2(n05_, n06_, n07_);
  OR2(n04_, n07_, n08_);
  NOT1(n08_, *done);
  AND2(counter_reg[0], n08_, n09_);
  OR2(counter_reg[0], n08_, n10_);
  AND2(n01_, n10_, n11_);
  BIC2(n11_, n09_, n00_[0]);
...
  DFF(clk, n00_[0], counter_reg[0]);
  DFF(clk, n00_[1], counter_reg[1]);
  DFF(clk, n00_[2], counter_reg[2]);
  DFF(clk, n00_[3], counter_reg[3]);
}
```

To use this PSP core function, we write a main function wrapper, *i.e.* PSP wrapper, which prepares the inputs at the beginning in bitsliced format and keeps calling the `counter_c(...)` function until all the parallel calculations are complete. This ensures that regardless of the input values, the run-time will be the same for all inputs. The following code snippet shows a simple wrapper for the `counter_c` example.

```
int main() {
    // prepare inputs in bitsliced format
...

    // reset:
    MDTYPE rst = 0xffffffff;
    counter_c(clk, rst, CNT_val, &done);

    // keep calling the counter_c() until all calculations complete:
    rst = 0;
    while (done != 0xffffffff) {
        counter_c(clk, rst, CNT_val, &done);
    }

    return 0;
}
```

In our previous work [13] we introduced an automated methodology to generate the PSP kernel from a Hardware Description Language (HDL) (such as Verilog) of the algorithm's FSMD. Throughout this paper we refer to this methodology as PSP Code Generator (PSPCG).

Execution time of AEAD ciphers: A normal implementation of the AEAD ciphers can have both data- and operation-dependent variations in run-time. On the one hand, AEAD ciphers entail several modes of operation to ensure integrity, confidentiality, and authenticity of data. In a software implementation of these ciphers, each one of these modes takes a certain amount of time that is not necessarily the same as the other modes' execution times. On the other hand, the run-time of some modes is dependent on the length of the message. These sources of run-time variation makes an implementation of AEAD ciphers prone to timing side-channel attacks.

The constant-time characteristics of PSP drive us to look into PSP implementations for a number of LWC candidates. We compare available reference implementations and bitsliced implementations of selected round two candidates with their PSP counterparts in terms of run-time, performance, and code size. Furthermore, we discuss an automated implementation methodology for PSP implementation of the selected ciphers. In Section 2, we describe the LWC ciphers studied in this work. In Section 3, we demonstrate our technique to incorporate constant-time property to AEAD implementations and to automate it using PSPCG. In Section 4, we study different aspects of our implementations of the chosen candidates and compare our results with the existing work. We conclude the paper in Section 5.

2 Selected Round 2 Candidates

Round 2 Candidates	**Primitve**	**Permutation/ Block cipher**	**AEAD**
ACE	ACE	Permutation	ACE-$\mathcal{AE}$-128
Ascon	p^{12}	Permutation	Ascon-128
GIFT	GIFT-128	Block cipher	GIFT-COFB
WAGE	{WAGE}	Permutation	WAGE-$\mathcal{AE}$-128

Table 1: List of selected Round 2 candidates for PSP evaluation

NIST LWC competition candidates are ideal for PSP code generation, where both the data and control flow of the algorithm is bitsliced. The candidates contain four broad phases in their AEAD computation. First, initialization and loading phase where the key and nonce are loaded onto the state. Second, processing associated data phase where the associated data is loaded onto the state. Third, encryption phase where the plaintext is loaded onto state to generate corresponding ciphertext. Fourth, finalization phase where the authentication tag is computed. The complex control involved in different phases are ideal for PSP. We focus on four Round 2 candidates for PSP code generation. The experiments performed in this paper and their corresponding measurements do not compare the selected candidates among each other. Instead, we focus on the feasibility of PSP code generation and its performance in AEAD operations of the selected candidates. We chose these submissions for several reasons, including but not limited to the availability of C reference implementation, Verilog or VHDL implementation, and existing bitsliced code. Other ciphers also posses the above resources, but in the interest of time we decided to only proceed with the four choices listed in Table 1.

ACE [1] is a 320-bit permutation which is used in different modes of operation for AEAD. The ACE permutation is based on sponge duplex construction in sLiSCP sponge mode [3]. The 320-bit state is a modest size for hardware and is a multiple of 32 and 64-bits for software implementation. The internal permutation of ACE consists of 16 iterations of ACE-step on the 320-bit state. This permutation is repeatedly applied to the state with different inputs, including the key, nonce, plaintext, and associated data, to perform the different steps of ACE-$\mathcal{AE}$-128. We focus on ACE-$\mathcal{AE}$-128 and ACE permutation to study the effects of our PSP bitslicing.

Ascon [10] contains a suite of authenticated encryption ciphers, mainly based on Ascon-128 and Ascon-128a. It was also selected as the primary choice for lightweight applications in resource constrained environments in the final portfolio of Competition for Authenticated Encryption: Security, Applicability and Robustness (CAESAR). It is a permutation based cipher, where the 320-bit state undergoes several p^a and p^b permutations. Its suite consists of a family of authenticated encryption ciphers based on their parameters, including the size of the secret key, the data rate, and the number of rounds of permutations (a and b) in different steps of encryption and decryption. We focus on Ascon-128 and its p^{12} permutations in our study.

WAGE [2] is a 259-bit lightweight cipher designed for efficient hardware implementation of AEAD. It also adopts the sLiSCP sponge mode [3] which is a variant of the traditional duplex mode. WAGE permutation consists of the tweaked initialization of Welch-Gong Permutation (WGP) [12]. A combination of WAGE LFSR with WGP and a 7-bit sbox provides a good trade-off between security and hardware efficiency, which was its design goal. It is used in unified sponge duplex mode for the AEAD function. We focus on the WAGE-$\mathcal{AE}$-128 algorithm and its permutation as candidates for PSP code generation.

GIFT-COFB [4] is a block cipher based AEAD candidate which uses COmbined FeedBack (COFB) block cipher mode of operation of AEAD on the hardware optimized GIFT-128 block cipher. It is also designed for hardware, with a focus on minimal hardware implementation size. GIFT-128 is 40-round iterative block cipher based on a 128-bit substitution permutation network (SPN). There are three steps involved in each round which operates on four 32-bit segments of the 128-bit state. First, SubCells performs substitutions after some computation. Second, PermBits applies different 32-bit permutations to the state. Thirds, AddRoundKey adds the round key and round constant to the state. We focus on WAGE permutation and its AEAD algorithm WAGE-$\mathcal{AE}$-128 as an example of block cipher based AEAD candidates for PSP code generation.

3 Implementation

This section describes how we implemented PSP versions of several lightweight ciphers. We studied four different round two candidates for NIST, listed in Table 1.

Traditionally, bitsliced code for cryptography is developed by hand, by expanding the operations in a cryptographic algorithm at the bit-level. Typical implementations that are developed in this manner concentrate on the data operations, while control operations remain implemented using conventional operations such as loops and `if-then-else` blocks. We observe two disadvantages with this approach. First, this data-centric bitsliced code only partially achieves the goals of PSP, because the control constructions maintain their time dependency on data. For example, with AEAD constructions, the resulting designs will maintain a time dependency on the length of the associated data and the length of the input message. Second, the development of bitsliced code is tedious as it requires the programmer to develop algorithms as Boolean programs (bit-level programs).

With this work, we wish to demonstrate how to overcome both of these disadvantages. We use two different automatic tools to generate bitsliced code. First, we have developed a PSP synthesis tool, PSPCG, which starts from RTL level hardware descriptions in Verilog and which generates true PSP code in C [13]. Our PSP synthesis tool creates C code where both the data processing as well as control processing exists in bitsliced form. The tool is developed on top of an open-source hardware logic synthesis tool. Second, we also use the Usuba compiler, which offers a combination of a dedicated programming language and a code synthesis tool. Usuba has been demonstrated for a wide range of lightweight cryptographic algorithms [6]. Usuba produces pure bitsliced C programs and it does not produce bitsliced control operations.

Implementation of PSP code: We illustrate the implementation of bitsliced code design for PSP by the following code snippets. We consider a GIFT-128 module definition in Verilog[1]. The module accepts a plaintext `P`, a key `K`, and produces a ciphertext `C`. The encryption is started through control input `ld` and completion is indicated through control output `done`. The full encryption requires 40 rounds, which are executed by the RTL design in 40 clock cycles.

```
module gift128(input  wire clk,
               input  wire [127:0] P,
               input  wire [127:0] K,
               input  wire ld,
               output wire [127:0] C,
               output wire done);
```

Our PSP code synthesis tool generates a C version out of the Verilog description in two steps. First, the Verilog RTL is converted into a Boolean Program. Next, the Boolean Program is expressed in terms of bitwise operations on the target processor. The RTL is converted using logic synthesis into a gate-level netlist. Next, the netlist is topologically sorted from primary input to primary ouput, and from flip-flop output to flip-flop input. This creates a Boolean Program, a sequential evaluation of the gate-level netlist obtained from the Verilog RTL program. Next, the PSP code synthesis tools converts the Boolean program into a C program by converting each Boolean operation into a bitwise operation. The bitwise operations are optimized towards the instruction set of the target processor. For example, the ARM Cortex series of processors have bitwise instructions that complement one operand, such as `BIC` and `ORN`. In addition, we

[1] The complete implementation of the GIFT-128 module is available on GitHub `https://github.com/Secure-Embedded-Systems/psp-nistlwc20`

use inline assembly for computations, leaving only register spilling (memory-load and memory-store) to the C compiler.

The following C header is created for the GIFT-128 example illustrated earlier.

```
#define MDTYPE uint32_t
void gift128(/* MDTYPE clk, -- notused */
             MDTYPE P[128],
             MDTYPE K[128],
             MDTYPE ld,
             MDTYPE C[128],
             MDTYPE* done);
```

The inputs to this function (control signals as well as data) are in bitsliced form: 128 bits are delivered through a 128-element array. MDTYPE is a machine-dependent data type representing the natural wordlength of the target processor. A single call to gift128 executes one cycle of the original Verilog design, but for 32 parallel instances of GIFT-128. As the original Verilog program takes 40 clock cycles to complete an encryption, the PSP program takes 40 calls to gift128 to complete an encryption. The following snippet illustrates the generated function body of gift128 with a code snippet. Logical functions are called in topological order (eg. output n074_ is produced in the first AND2 and consumed in the first OR2 below that). State elements (such as flip-flops in the netlist) are implemented using static variables in C. A flip-flop is updated by assignment of the static variable. In the PSP program, the clock signal is implicit since each PSP function call corresponds to one clock cycle.

```
  ..
  MDTYPE n0974_, n0975_, n0979_;
  MDTYPE Mstate_reg_92__D;
  MDTYPE Mstate_reg_107__D;
  MDTYPE Mroundkey_reg_31__D;
  static MDTYPE Mstate_reg_107__Q;
  static MDTYPE Mroundkey_reg_31__Q;
  ..
  AND2(ld, P[92], n0974_);
  AND2(n0979_, C[92], n0975_);
  OR2(n0974_, n0975_, Mstate_reg_92__D );
  ..
  DFF(/* clk, */ Mstate_reg_107__D , Mstate_reg_107__Q );
  DFF(/* clk, */ Mroundkey_reg_31__D , Mroundkey_reg_31__Q );
  ..
```

Execution-time properties of PSP code: With the structure of the code as presented, we can make a more precise statement on the execution-time properties of PSP code. While it is common to call bitslice code 'constant-time', the reality is that this property is rarely achieved in practice, even for bitsliced code. Execution time depends on program logic, the underlying processor architecture and memory hierarchy, and resource-sharing effects in the processor. PSP programs achieve the following properties with respect to their execution time.

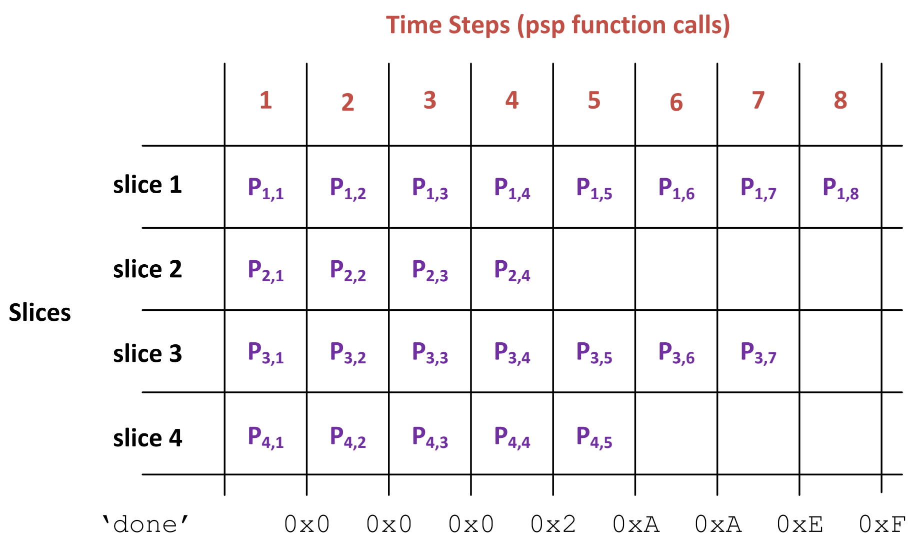
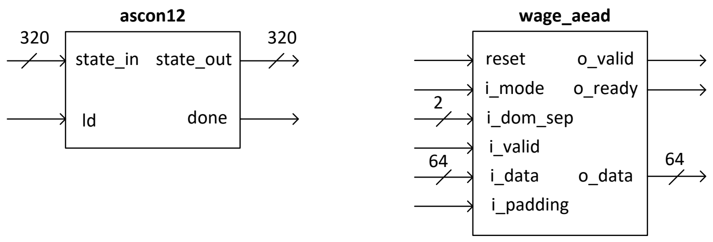

Time Steps (psp function calls)
1
2
3
4
5
6
7
8
Slices
slice 1
$P_{1,1}$ $P_{1,2}$ $P_{1,3}$ $P_{1,4}$ $P_{1,5}$ $P_{1,6}$ $P_{1,7}$ $P_{1,8}$
slice 2
$P_{2,1}$ $P_{2,2}$ $P_{2,3}$ $P_{2,4}$
slice 3
$P_{3,1}$ $P_{3,2}$ $P_{3,3}$ $P_{3,4}$ $P_{3,5}$ $P_{3,6}$ $P_{3,7}$
slice 4
$P_{4,1}$ $P_{4,2}$ $P_{4,3}$ $P_{4,4}$ $P_{4,5}$
'done'
0x0 0x0 0x0 0x2 0xA 0xA 0xE 0xF

Fig. 2: Aggregate execution of 4 slices over 8 time steps

- Each call to a PSP function will execute the same number of Boolean operations and load/store operations. Each call to a PSP function completes one logical step in an algorithm, corresponding to one clock cycle from a reference RTL description. In Figure 2, each column represents one such time-step.
- The PSP function computes (N=32) Boolean programs in parallel, and each of these programs can be independently controlled through bitsliced control. For the `gift` example above, `ld` initiates the execution of any combination of 32 parallel GIFT-128 algorithms. Similarly, `done` indicates the completion of any combination of 32 parallel `gift` algorithms. In Figure 2, each row represents one instance of the algorithm. The value of the aggregate `done` signal is shown on the bottom row.
- Because control is bitsliced, the parallel versions of the Boolean programs do not have to take the same amount of time steps. Furthermore, each of the parallel versions can start and end independently. In Figure 2, four instances start at the same time. But each instances takes a different amount of time steps to complete. At macro-level, however, the PSP function is executed 8 times to complete all 4 instances of the algorithms. Constant time over the aggregate is achieved using a simple loop:

```
while (*done != 0xF)
    gift128(P, K, ld, C, &done);
```

– Within the PSP function, additional load/store operations are used as a result of register spilling. These load/store operations cannot be exploited for side-channel leakage because they store a single bit from an aggregate of 32 parallel algorithms.



```
void ascon12(MDTYPE state_in[320],
             MDTYPE ld,
             MDTYPE state_out[320],
             MDTYPE *done);
```

```
void wage_aead(MDTYPE reset,
               MDTYPE i_mode,
               MDTYPE i_dom_sep[2],
               MDTYPE i_valid,
               MDTYPE i_data[64],
               MDTYPE i_padding,
               MDTYPE *o_valid,
               MDTYPE *o_ready,
               MDTYPE o_data[64]);
```

Fig. 3: Ascon-p^{12} permutation interface and WAGE-$\mathcal{AE}$-128 interface examples

Processing of AEAD Implementations: In our experiments with Ascon-128, ACE-$\mathcal{AE}$-128, WAGE-$\mathcal{AE}$-128, and GIFT-COFB, we investigated both stand-alone permutations as well as AEAD implementations. Both implementations used the same methodology to create a PSP implementation. However, the AEAD versions are considerably more complex in terms of processing, as they go through independent initialization, AD processing, message processing and finalization phases.

Figure 3 illustrates the interfaces for a typical permutation (Ascon-p^{12}) and a typical AEAD implementation (WAGE-$\mathcal{AE}$-128) analyzed in our experiments. Because a PSP implementation processes a sequential version of the original RTL design, a wide interface such as 320 bits for the case of Ascon-p^{12}, does not pose a significant challenge.

A PSP design offers parallelism on the order of the wordlength of the target processor. This manifests itself in two areas. First, the inputs of a PSP function must be transposed in the same manner as a bitslice function. This transposition can be performed before any processing starts, at the cost of extra storage. In our experiments, we account separately for the overhead caused by this transposition. Second, we need to keep all parallel algorithm instances embedded in a PSP function busy. We believe this may be handled by making use of a parallel mode of operation, such as Farfalle-like parallel modes for sponges [7]. We identify this as future research.

AEAD	Implementation	Code size(B)	Cycles/byte
Ascon-128	Reference	15,709	2,751
	PSP	210k	6,855
WAGE-$\mathcal{AE}$-128	Reference	24,439	9,305
	PSP	47,304	21,032

Table 2: Overhead of complete PSP implementations of Ascon-128 and WAGE-$\mathcal{AE}$-128 when compared with reference implementation. The measurements were computed for processing 8B of plaintext and 8B of associated data

4 Results

In this section, we present the overall overhead, including cycle count, instruction count, and code size, of our PSP implementations of selected Round 2 candidates listed in Table 1. We use the reference implementations from Round 2 candidates[2] to derive our baseline. We compare our PSP implementation with Usuba's bitsliced implementation[3][5] to highlight the advantage of automatic bitsliced code generation of the full AEAD modes when compared to the block cipher or permutation. We use Texas Instrument's MSP432P401R SimpleLink microcontroller LaunchPad development kit [14] for our experiments. It is equipped with a 32-bit Cortex-M4F microcontroller with a clock frequency upto 48 MHz. With several ultra-low operating modes and a Flash memory of 256KB, we consider MSP432P401R as a representative of low-power devices that may require AEAD to secure its applications. We use TI's Code Composer Studio (CCS) as our development environment. The measurements reported in this paper are computed with the MSP432P401R operating at 48 MHz. The implementations were also optimized for speed at compile time.

Comparison with reference implementation: We compared the reference implementation of Ascon-128 and WAGE-$\mathcal{AE}$-128 with our PSP implmentations of the same in Table 2. The cycles per byte for transposing input data is not included in these measurements. Even though, the increase in code size in our bitsliced implementation is not consistent across Ascon-128 and WAGE-$\mathcal{AE}$-128, the latter has less than two times increase in code size. This inconsistency may stem from the cipher design, which requires further study. The PSP implementations are fully unrolled with each execution of the core function taking the longest critical path irrespective of the input. This adds additional overhead to PSP implementation in processing the same input as reference implementation. Our PSP implementations of both Ascon-128 and WAGE-$\mathcal{AE}$-128 has less than 2.5 times increase in cycles per byte of AEAD operation.

Comparison with the state-of-the-art: Usuba's open sourced bitsliced implementations of Round 2 candidates are only available for their permutations and block

[2] NIST LWC Round 2 Candidates Submissions `https://csrc.nist.gov/projects/lightweight-cryptography/round-2-candidates`

[3] `https://github.com/CryptoExperts/Tornado/tree/master/src/usuba/nist`

Cipher/permutation	**Implementation**	**Code size (B)**	**Cycle count (cycles)**
ACE permutation	Reference	14,937	261,833
	Usuba	115k	1,408,096
ACE-$\mathcal{AE}$-128	PSP	84,280	34,662

Table 3: Code size and cycle count comparison of ACE-$\mathcal{AE}$-128 PSP core implementation with ACE permutation reference implementation and its Usuba bitsliced implementation.

ciphers and not their AEAD modes of operations. We studied their bitsliced implementation of ACE permutation. It is a fully unrolled implementation of the 16 iterations of ACE-step and the non-linear sbox operations. When compiled for MSP432P401R, Usuba based ACE permutation generates 115kB of code, as listed in Table 3. To use this bitsliced permutation in ACE-$\mathcal{AE}$-128, it will be used $5+l$ times, where l is the number of processed data in 64-bit blocks [1].

Our PSP core function is the bitsliced implementation of ACE-$\mathcal{AE}$-128. This core function will be repeated called by a wrapper to provide different inputs such as nonce, key, plaintext, and associated data. We discuss the increase in code size from this wrapper below.

Comparison of logic optimizers: An important part of automatic code generation is the optimizer. In the case of both the PSP code generator and the Usuba compiler, this optimizer should be able to compress the logic operations as much as possible to make the constant execution time shorter and the code size smaller. We take the two cipher candidates Ascon-128 and GIFT-COFB and generate the PSP core function of their permutations using the PSPCG and calculate the number of logic operations in the generated PSP code as discussed in Section 3. We also count the number of operations in a completely unrolled bitsliced code for the same permutation algorithm generated by Usuba compiler. Table 4 shows the number of logic operations for each of these implementations. As shown in this table, PSP core functions is 37% and 92% smaller in terms of logic operations for Ascon-p^{12} and GIFT-128 respectively.

This is not a one-to-one comparison since the functions that are being compared are not completely similar; the PSP core function is the building block of PSP program which should be called iteratively until the `done` output is set,

Cipher	**Code Generator**	**AND**	**ORR**	**EOR**	**MVN**	**total**
Ascon-p^{12}	Usubac	3840	0	16128	4657	24625
	PSPCG	6381	6034	1669	1324	15408
GIFT-128	Usubac	3840	1280	14080	1360	20560
	PSPCG	756	418	262	137	1573

Table 4: Comparison of number of instructions resulting from Usubac and PSPCG

whereas, the bitsliced unrolled code is the complete permutation. However, this comparison shows that the code size of a PSP implementation will be much smaller than a bitsliced implementation.

Cipher	AND	ORR	EOR	MVN	MOV	LDR	STR	overhead
ACE permutation	64	0	1168	832	11229	1469	25	86.04%
GIFT-128	192	64	704	66	6725	2041	18	89.54%

Table 5: Register spill of the bitsliced code generated by Usubac

Cipher	AND	ORR	BIC	EOR	ORN	MVN	MOV	LDR	STR	overhead
Ascon-p^{12}	1732	1296	281	808	1265	123	4904	10277	3862	77.57%
GIFT-128	530	57	30	58	54	2	120	1415	971	77.42%

Table 6: Register spill of the PSP code generated by PSPCG

Comparison of register spilling: One disadvantage of bitsliced and PSP code in performance is the pressure they put on the register file and the overhead of memory access instructions which are usually considered to be among the slow instructions of a microprocessors. Therefore, we compare the overhead of data-move instructions that result from compiling the generated codes from PSPCG and Usubac. Tables 5 and 6 show the overhead of such instructions calculated as the number of MOV, LDR, and STR instructions over the total number of instructions. The comparison shows that for both the bitsliced code and the PSP code, more than half of the instructions are dedicated to moving data between memory and register. The bitsliced code generated by Usubac has in average 87.79% overhead for load, store, and move instructions. This number for PSP codes is 77.49% which is approximately 10% lower overhead than bitsliced code. This experiment shows that PSP has a significant memory-spill yet it is smaller than a bitsliced code.

5 Conclusion

In this paper, we addressed the execution time characteristics of AEAD ciphers and their proneness to timing side-channel. We evaluated the feasibility of automatic constant-time code generation on selected Round two candidates, namely ACE-$\mathcal{AE}$-128, Ascon-128, GIFT-COFB, and WAGE-$\mathcal{AE}$-128 by generating the PSP version of the Verilog description of the cipher which transforms the data- and control- flow into bitsliced format and ensures constant-time execution. Furthermore, the effects of Usuba's bitslicing and our PSP implementation on selected candidates were studied. We analyzed our PSP code generation tool, instruction count, code size, and cycle count overhead of the PSP implementation of selected Round two candidates. Our PSP implementation provides constant-time execution with a modest overhead, as shown for Ascon-128 and WAGE-$\mathcal{AE}$-128.

Acknowledgements. This research was supported in part by NSF Award 1704176 and 1931639.

References

[1] Mark Aagaard, Riham AlTawy, Guang Gong, Kalikinkar Mandal, and Raghvendra Rohit. *ACE: An Authenticated Encryption and Hash Algorithm.* Submission to the NIST LWC Competition. 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/ace-spec-round2.pdf.

[2] Mark Aagaard, Riham AlTawy, Guang Gong, Kalikinkar Mandal, Raghvendra Rohit, and Nusa Zidaric. *WAGE: An Authenticated Cipher.* Submission to the NIST LWC Competition. 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/wage-spec-round2.pdf.

[3] Riham AlTawy, Raghvendra Rohit, Morgan He, Kalikinkar Mandal, Gangqiang Yang, and Guang Gong. "sLiSCP: Simeck-Based Permutations for Lightweight Sponge Cryptographic Primitives". In: *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers.* Ed. by Carlisle Adams and Jan Camenisch. Vol. 10719. Lecture Notes in Computer Science. Springer, 2017, pp. 129–150. DOI: 10.1007/978-3-319-72565-9_7. URL: https://doi.org/10.1007/978-3-319-72565-9%5C_7.

[4] Subhadeep Banik, Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. *GIFT-COFB v1.0.* Submission to Round 2 of the NIST Lightweight Cryptography project. 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/gift-cofb-spec-round2.pdf.

[5] Sonia Belaïd, Pierre-Evariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. *Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations.* Cryptology ePrint Archive, Report 2020/506. https://eprint.iacr.org/2020/506. 2020.

[6] Sonia Belaıd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. "Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations". In: *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part III.* Ed. by Anne Canteaut and Yuval Ishai. Vol. 12107. Lecture Notes in Computer Science. Springer, 2020, pp. 311–

341. DOI: 10.1007/978-3-030-45727-3_11. URL: https://doi.org/10.1007/978-3-030-45727-3%5C_11.

[7] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. "Farfalle: parallel permutation-based cryptography". In: *IACR Transactions on Symmetric Cryptology* (2017), pp. 1–38.

[8] Joseph Bonneau. "Robust Final-Round Cache-Trace Attacks Against AES." In: *IACR Cryptol. ePrint Arch.* 2006 (2006), p. 374.

[9] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. "Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme". In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2016, pp. 323–345.

[10] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon v1.2*. Submission to Round 1 of the NIST Lightweight Cryptography project. 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/ascon-spec.pdf.

[11] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a minute! A fast, Cross-VM attack on AES". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2014, pp. 299–319.

[12] Jong-Seon No, S. W. Golomb, Guang Gong, Hwan-Keun Lee, and P. Gaal. "Binary pseudorandom sequences of period 2/sup n/-1 with ideal autocorrelation". In: *IEEE Transactions on Information Theory* 44.2 (1998), pp. 814–817.

[13] P. Kiaei and P. Schaumont. "Synthesis of Parallel Synchronous Software". In: *IEEE Embedded Systems Letters* (2020), pp. 1–1.

[14] *MSP432P401R SimpleLink^{TM} Microcontroller LaunchPad^{TM} Development Kit (MSP--EXP432P401R.* https://www.ti.com/lit/ug/slau597f/slau597f.pdf?ts=1600385991996&ref_url=https%253A%252F%252Fwww.ti.com%252Ftool%252FMSP-EXP432P401R. Mar. 2015.

[15] Michael Neve and Jean-Pierre Seifert. "Advances on access-driven cache attacks on AES". In: *International Workshop on Selected Areas in Cryptography*. Springer. 2006, pp. 147–162.

[16] *NIST's Lightweight Cryptography Competition.* https://csrc.nist.gov/projects/lightweight-cryptography. 2018.

[17] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache attacks and countermeasures: the case of AES". In: *Cryptographers' track at the RSA conference*. Springer. 2006, pp. 1–20.

[18] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. "To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1843–1855.

[19] Yuval Yarom and Katrina Falkner. “FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack”. In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 719–732.