IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 3, pp. 377–417. DOI:10.46586/tches.v2024.i3.377-417

CrISA-X: Unleashing Performance Excellence in Lightweight Symmetric Cryptography for Extendable and Deeply Embedded Processors

Oren Ganon[1] and Itamar Levi[1]

Bar-Ilan University, Ramat-Gan, Israel.
oren.ganon@biu.ac.il,itamar.levi@biu.ac.il

Abstract. The efficient execution of a Lightweight Cryptography (LWC) algorithm is essential for edge computing platforms. Dedicated Instruction Set Extensions (ISEs) are often included for this purpose. We propose the *CrISA-X*-a Cryptography Instruction Set Architecture eXtensions designed to improve cryptographic latency on extendable processors. *CrISA-X*, provides enhanced speed of various algorithms simultaneously while optimizing ISA adaptability, a feat yet to be accomplished. The extension, diverse for several computation levels, is first tailored explicitly for individual algorithms and sets of LWC algorithms, depending on performance, frequency, and area trade-offs. By diligently applying the Min-Max optimization technique, we have configured these extensions to achieve a delicate balance between performance, area utilization, code size, etc. Our study presents empirical evidence of the performance enhancement achieved on a synthesis modular RISC processor. We offer a framework for creating optimized processor hardware and ISA extensions. The *CrISA-X* outperforms ISA extensions by delivering significant performance boosts between 3x to 17x while experiencing a relative area cost increase of +12% and +47% in LUTs. Notably, as one important example, the utilization of the ASCON algorithm yields a 10x performance boost in contrast to the base ISA instruction implementation.

Keywords: Application specific · Configurable · Extensible · Modular Processor Architecture · Lightweight Cryptography

1 Introduction

Lightweight cryptography algorithms employ symmetric-key encryption and decryption to secure edge-devices data. In 2013, the National Institute of Standards and Technology (NIST) initiated a competition to choose authenticated encryption and hashing schemes for such devices [TMC+21, MAA+20]. Ten finalists were selected in 2021, as listed in Table 1. Ascon LWC algorithm was then selected in 2023 as the standard for lightweight cryptography. The CAESAR committee also launched a contest with the same outcome. Researchers are studying the acceleration of LWC algorithms on edge computing environments with limited resources, as the selection process evaluates candidates based on their performance in cycle count, area, and footprint measurements, as reported in literature [XJL+19, SR20]. Lightweight cryptography involves multiple rounds of computation, leading to significant latency for encryption, decryption, and hashing of each data set. By increasing the speed of these computational blocks, it becomes possible to process more extensive encryption data on these devices without compromising their performance or battery life [GPKT09, DFA+20]. Faster cryptographic encryption in edge devices requires incorporating optimized software libraries or dedicated hardware accelerators. Software is

Received: 2024-01-15 Accepted: 2024-03-15 Published: 2024-07-18

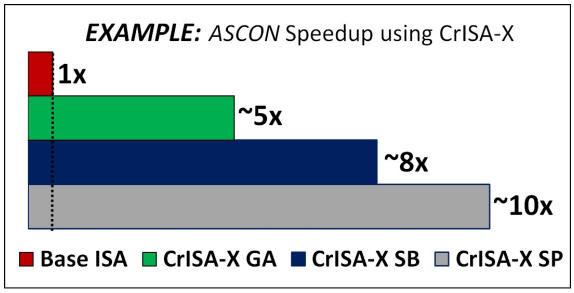

Figure: 1. ***Example*** *of Enhancing ASCON Performance with CrISA-X: A Comparative Latency Analysis of Base ISA Instruction (Baseline) vs. Replacement Code*

flexible but not suitable for low-frequency devices with strict latency requirements. Hardware accelerators perform better for high-frequency devices but require more hardware space, suffer from data transfer delay, and are less flexible than software. As a result, there is a growing emphasis on developing processor-based mechanisms, such as ISEs, that enable efficient encryption while maintaining low-latency requirements and keeping reusability and flexibility. Researchers exploring this domain are looking specifically at one vector of leveraging the processor's basic ISA and supplementing it with additional instructions to encapsulate a certain amount of computation [MB20, VVP+16, TMC+23, Sab23, GS22]. However, to gain a quantitative understanding of how additional extended instructions, with various levels of computation, can improve the speed and reduce memory costs, it is necessary to tighten the instruction extension together with extended processors architecture and associated hardware. This study aims to explore the potential of instructions extensions with any extendable RISC processor, such as the RISC-V open-standard ISA, to enhance the performance of lightweight cryptographic algorithms. The paper provides technical details regarding the implementation process, which includes the design and development of ISEs, their integration with processors, and the overall impact on system performance and area. The study's findings have significant implications for accelerating LWC algorithms, especially for Ascon, as shown for example in Figure 1 on RISC processors, providing an efficient solution for LWC. As detailed below we have evaluated all finalists for demonstration, namely *ASCON, Elephant, GIFT-COFB, Grain128-AEAD, ISAP, Photon-Beetle, Romulus, Sparkle, TinyJambu, Xoodyak*, and customized a vast set of supporting hardware architectures.

Table 1: NIST LWC finalist algorithms

Algorithm	Variant	Feedback
Grain-128AEAD	AEAD	Stream Cipher
GIFTcofb	AEAD	Permutation GIFT-128
Romulus	AEAD,Hash	Block Cipher Skinny-128-384+
ASCONf	AEAD,Hash	Permutation Ascon-p
Elephant	AEAD	Permutation Keccak-f[m]
ISAP	AEAD	Permutation Keccak-f[m]
PHOTONbee	AEAD,Hash	Permutation PHOTON256
SPARKLE	AEAD,Hash	Permutation Sparkle
TinyJAMBU	AEAD	Permutation TinyJAMBU
Xoodyak	AEAD,Hash	Permutation Xoodoo

2 Related Work and Our Contribution

Accelerating cryptographic algorithms using custom instruction-set architectures or hardware/software co-design, have long been considered a potential means of enhancing their computation speed. Although the concept is familiar, limited studies have been conducted on the topic. Specifically, existing research has been divided into three categories: first, efficient implementations of the candidates over standard processor architecture, software-hardware and ISAs; second, works that focus on accelerating only one candidate over simple cores[1]; and lastly, some works aim to jointly optimize a small subset of algorithms with minimal or no adaptation of extended processor architectures.

2.1 Related Work Overview

This section aims to provide an overview of the research related to the topic. It will describe the key findings of the relevant studies conducted by previous researchers. Table 2 summarizes the main research conducted in this area. Renner and Pozzobon [RPM22], and Sebastian and Enrico [WYY22] conducted a comprehensive analysis of the final ten ciphers from NIST's lightweight cryptography project. They assessed the efficient software implementation performance of these ciphers on different 32-bit processor platforms with varying architectures and provided valuable insights into their performance characteristics under specific test scenarios. The authors' research can assist developers in establishing a performance baseline for LWC software implementation on micro-controller platforms. Edwards and Forrest [EF96] studied hardware/software codesign as a tightly coupled accelerator to enhance software acceleration. Critical regions are parts of an application where software alone cannot meet performance requirements, necessitating hardware-based solutions or where implementing specific regions in hardware significantly improves performance. Their research highlights the importance of hardware/software codesign to achieve optimal performance. As the popularity of the RISC-V architecture continues to rise, researchers are exploring ways to efficiently implement LWC algorithms through RISC-V ISA Extensions as co-processors. Two teams, Altınay and Örs [AÖ21], and Steinegger and Primas [SP20] have developed ISA Extensions to speed up Ascon permutation. They have designed dedicated instructions to perform an entire Ascon-p round efficiently in hardware. However, these ISA Extensions are designed to target a single algorithm and require tight integration with a specific processor core. They employ hard-coded registers to store the state and focus on a single data path without instruction or memory-level parallelism. Marshall, Page, and Hung Pham [MPP21] have also developed ISA Extensions to accelerate the ChaCha stream cipher by using a similar approach of dedicated coprocessor hardware for a specific stream cipher. While the last mentioned team focused on a single algorithm, Tehrani et al. [TGMD20] focused on a few, but limited, LWC algorithms, including 64-bit block ciphers like GIFT-64-128 and Skinny-64-128. In their article, ZiBin Dai and XueRong Yu [DYSC07] aim to improve the speed of cryptography processing by utilizing a flexible, albeit limited, processor architecture. The authors suggest a specific method to enhance the efficiency of analyzed algorithms by introducing a new set of very long instruction word (VLIW) instructions. These instructions leverage paralleling processing elements for specific instructions and follow a conventional hardware/software design approach to enhance the performance and flexibility of cryptographic processing. However, their analysis is limited to older ciphers and does not cover lightweight ones. The authors focus on the VLIW and additional ISA as a single mechanism to improve performance, without extending the hardware architecture. In this paper we investigate and elaborate further on this approach. In parallel to our conducted research and our

[1] Referring to a single processing element, limited memory and instruction-level parallelism, and other related features from the architecture designer tool-set

basic short-version publication [GL23][2], Hao Cheng et al. [CGM+23] presented the design, implementation, and evaluation of Instruction Set Extensions (ISEs) for nine of the ten LWC final round submissions, providing evidence for richer evaluation with respect to metrics related to implementation. The authors analyze the performance of these ISEs on a RISC-V processor and compare them to software implementations. The article focuses on the RISC-V additional ISA while keeping the processor architecture as 32-bit ***native*** processor, without hardware or conceptual architectural changes, more related to the realm of extendable and modular processors. Among other aspects and differences, we touch upon this significant aspect and evaluation void in this paper. Notably, the work from Cheng et al. focused on one ISE without the general goal of searching for an ISE which solves the min-max paradigm as we define it: "one minimal ISE which maximises its' performance impact on sets of algorithms jointly". Instead, their proposed ISE can be seen as a union of separate ISE per algorithm altogether.

2.2 Our Contribution

The study provides a detailed description of *CrISA-X*, a classification of processor instruction extensions and tightly coupled acceleration logic for LWC algorithms. Accurately categorising these extensions is crucial in evaluating the trade-off between performance and area when integrating new instructions into a processor. In particular, this classification can aid in determining the optimal balance between hardware acceleration and software execution for a given processor architecture. We explore various ways to expand processor hardware to accommodate new instruction sets and discuss the necessary process architecture required to support these extensions. These enhancements are carefully designed to support a range of LWC algorithms simultaneously.

The *CrISA-X* ISA-extension instruction subsets include *Generic-Atomic, Specific-Block*, and *Specific-Procedure* categories, all of which greatly enhance the performance of NIST and CAESAR AEAD competitor algorithms. *Generic-Atomic* new instructions are a combination of bitwise operations designed for efficient computation with a small number of operands, allowing for the composition of a wide cryptographic permutation spectrum. The new instructions are executed efficiently within a single clock cycle. In contrast, *Specific-Block* are designed for extensive computation and serve as a foundational component for composing cryptographic permutation code. These new instructions exploit substantial memory- and instruction-level parallelism. To further enhance the acceleration of even larger blocks covering entire permutations, we introduce the *Specific-Procedure* instructions which establish the optimal subset to maximize acceleration. As significant computation is required for *Specific-Block* and *Specific-Procedure*, they are incorporated as tightly coupled acceleration logic **within the processor pipeline**. The research aims to showcase our capability in solving a min-max problem by creating an extended ISA that can be used seamlessly by designers and cryptographers. The new set of instructions is designed to *maximize* impact across various cryptographic algorithms while *minimizing* costs and the number of required extended instructions.

Our main contributions are listed below:

1. The CrISA-X ISA new instructions stand out for their innovative computing capabilities, distinguishing between atomic, block, and procedure instructions, which cover a wide range of tradeoffs when it comes to latency performance and area utilization.

2. New instructions hardware implementation is carefully designed as single-cycle or complex multi-cycle specific operations to accommodate various architectural constraints and minimise the negative impact of the processor's critical paths and thus performance.

[2]Best-paper award winner at IEEE NEWCAS conference

Table 2: Summary of Related Work
Academic Work on Accelerating Cryptography Algorithms using ISA Ext'

Related Work	Characteristics /#Algos'	Extendable Processor	Extendable ISA
[MPP21]	Single Algo, Native C-Code	64bit,No extend HW	Extended RISC-V ISA
[CGM+23]	All finalist, Native C-Code	32bit,No extend HW	Extended RISC-V ISA
[CJL+20]	All finalist, ASM-Code	32bit,No extend HW	Base RISC-V ISA
[WYY22]	Several Algos', Native C-Code	32bit,No extend HW	Standard ARM ISA
[RPM20]	All finalist, Native C-Code	32bit,No extend HW	Standard RISC-V ISA
[RPM22]	All finalist, Native C-Code	32bit,No extend HW	Standard RISC-V ISA
[ABCdS+22]	All finalist, Native C-Code	32bit,No extend HW	Standard TI ISA
[AÖ21]	Single Algo, Native C-Code	32bit,Partly extend HW	Extended RISC-V ISA

3. We demonstrate the extension of the base processor hardware to facilitate the support of high levels of Instruction-Level Parallelism (ILP) and Memory-Level Parallelism (MLP) within the new Crisa-X ISA extension.
4. The CrISA-X classification also categorizes instructions into algorithm-agnostic and algorithm-specific types. For algorithm-agnostic applications, the CrISA-X ISA offers a versatile set of instructions suitable for all algorithms.
5. Depending on the required speedup, CrISA-X can be used for specific applications or a wide range of lightweight cryptography candidates while considering area budget.
6. This new instruction set incorporates Min-Max optimization to identify the computation level of ISA that minimizes processor area implementation, maximizes ISA versatility among various algorithms, and boosts performance, delivering optimal value. The search process within this extensive design space as elaborated.
7. The paper presents quantitative execution time analysis on a real synthesis modular processor design, offering insights into the actual performance of the new ISA set for various LWC algorithms.
8. The research introduces an efficient, automated methodology encompassing the identification of computation-heavy software blocks, ISA extension design for diverse classifications, simulation, validation, and the generation of synthesis processor RTL.
9. The CrISA-X ISA's toolchain support is versatile and compatible with various hardware platforms, including the Cadence Tensilica framework. Furthermore, it is designed to be freely shared within the academic community and released under an open-source license to promote transparency and collaboration.

2.3 Organisation

The paper is divided into several sections. The Introduction Section provides a background for this study and includes an in-depth analysis of related work in this domain. The list of

our contributions is then presented. In the Methodology section, we describe CrISA-X in detail. We begin by describing our design flow and profiling, followed by the acceleration model, extendable processor, instruction extension, and the challenges we faced. In the Design section, we delve into the design considerations of CrISA-X for various LWC algorithms. Lastly, we introduce multiple implementation techniques and present the CrISA-X evaluation results on real hardware and simulation, and compare to all publicly available prior-art and available designs and public codes.

2.4 Scope of our work

We have narrowed our focus to handle the extensive design space and engineering effort:

1. We are discussing a modular extensible processor with a base ISA that operates on 32-bit architecture. Our proposal for an additional new ISA encoding space ranges from 32-bit to 128-bit, which provides a wider set of ISAs and makes it more generally powerful. It is worth noting that the NIST call outlines a requirement to consider "8-bit, 16-bit, and 32-bit microcontroller architectures" [NIS22a], which is in line with our direction.

2. We focus solely on the AEAD encrypt and decrypt API, rather than both the AEAD and hash function API. The same kernel is used for both APIs.

3. We do not consider ISAP as the Ascon-p permutation used is already covered by other algorithms like Ascon.

4. In this work, we do not provide support in the base ISA or instruction extensions for implementing countermeasures against implementation attacks. This is left for continuous research.

3 Methodology

3.1 CrISA-X Design Flow

Designing an application-specific custom processor, along with custom instructions, is a challenging task for designers. They have to ensure that the resulting design is optimal constantly. It is a time-consuming process that requires designers to evaluate numerous combinations of available parts that make up the processor. Achieving an optimal design is a difficult problem requiring much effort. Different design flows have been proposed to overcome such a difficult design choice. Figure 2 gives one such design flow, which outlines the process for designing CrISA-X instructions set for LWC algorithm. This design flow profiles a set of LWC application programs and associated data. Profiling is performed upon numerous processor configurations. Each configuration consists of the base processor with one or more coprocessors. For example, the configuration might be a base processor and a dual load-store unit. We gather extensive profiling analyzer data to identify "hotspot" functions across LWCs, together with structural and functional aspects of the code, particularly to the permutation implementation. Structural code analysis involves examining its physical structure or form without necessarily delving into its functionality. This analysis emphasizes the code organization, relationships between different program components, and the overall architecture of the algorithm. We identify the flow of control in the algorithm, including loops, conditionals, and branching. Additionally, we examine how data is organized and manipulated within the algorithm, including arrays, linked lists, trees, and more. This analysis helps us understand the code organization and flow and aid in identifying potential areas for optimization by the new ISA extension. Functional analysis, however, focuses on what the algorithm specifically does and how

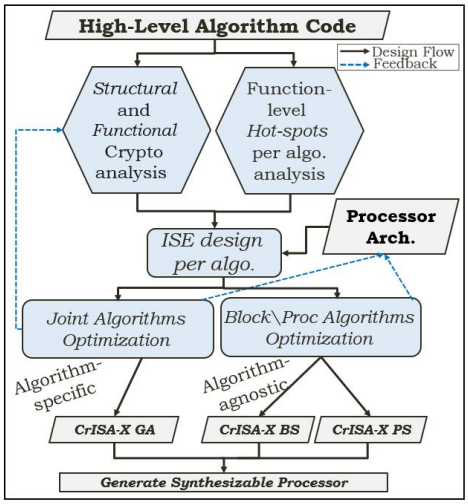

Figure: 2. *High-Level design flow for CrISA-X instructions sets.*

well it achieves its intended purpose. The profiling data allows us to learn where the program spent its time and which functions called which other functions while executing. This information can show which pieces of the program are slower than expected and might be candidates for rewriting to make LWC programs execute faster. It can also tell us which functions are being called more or less often than expected. The analysis of profiling data and cryptography's structural and functional elements holds significant importance in verifying code computations and their intensity. This analysis leads to developing new instructions to meet the increasing computational needs. Once the profiling has been performed on all these, a heuristic is used to select one of the configurations for further optimization. In the second step, we develop an ISE for each computation code. As the new instruction is tight to a processor architecture, we will examine both aspects during the implementation. Instruction extension is realized through a hardware implementation scripted in a specialized Hardware Description Language (HDL). The CrISA-X design is classified based on the encoding space and computation logic utilized to perform instructions. Other parameters, such as the number of operands used, input and output data, and computation logic, are also considered. Furthermore, we endeavor to explore potential techniques for extending the modular processor to enable support for varying levels of logic in instructions. This extension will cater to instruction sets implementing low, mid, and high amounts of logic. The data output is designed to be ready in a single cycle for instructions categorized as low computation logic. However, for mid to high levels of computation logic, the implementation of computations spans multiple clock cycles, mostly two cycles. This is all made to minimize the impact on processor operation frequency. In the final step, we will analyze the tradeoffs and opportunities of using the new ISA extension among a larger set of LWCs. We will explore how multi-LWC algorithms can be created by building upon the single ISE designs and also search for opportunities to optimize joint algorithms. As a result of this optimization flow, each new instruction will be targeted for *Specific-Procedure* or *algorithm-agnostic* use and algorithm C-code will update accordingly to use the new ISA extension. Every new instruction incorporated into an algorithm adheres to functional correctness. Also, each LWC application validates that the algorithmic behavior aligns with the original across various test patterns, which will be expounded upon later in this paper.

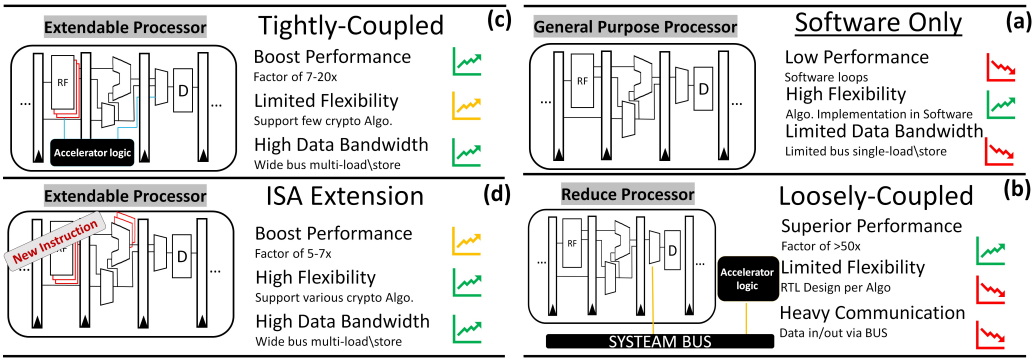

Figure: 3. *hardware/software co-design strategies (a) software only (b) loosely-couple (c) tightly coupled and (d) ISA extension.*

3.2 Selecting the Accelerator Models

To cope with reaching high-performance challenges, hardware/software co-design strategies are usually employed. For a given LWC algorithm, the goal is to find an optimal assignment of tasks between software running on reduced instruction set computer (RISC) processors and hardware implemented as accelerator logic. By splitting the tasks of an algorithm into hardware and software elements, high-speed and flexible implementations can be developed. Various options exist for accelerating cryptographic solutions based on embedded processor core [PMMB22, SZII11, Sea01]. These options differ in throughput, flexibility, range of solutions, and area cost. Refer to Figure 3 for illustrations of the main implementation options. *Software implementation* of cryptographic algorithms is clearly a simpler and easier approach. It does not require a specialized hardware block. It does not impact the die size and therefore does not raise system costs. This approach requires some code space in the instruction memory, making updating and maintaining the cryptographic code easy and affordable. The processor is designed to run algorithms with pipeline architecture efficiently, supporting conditional execution, function calls, and stack memory. It is also optimised for complex mathematical operations, including integer multiplications and floating-point calculations. However, processors may be less effective when executing cryptographic procedures involving many bitwise operations within a loop. This is because each bitwise operation can only be implemented as a single processor instruction in a single cycle [PS00]. Software-only implementations should be considered a last resort when high-computation cryptography algorithms are required for low-latency applications on general-purpose processors.

A high throughput approach for accelerating cryptographic algorithms uses hardware logic, illustrated in Figure 3(b,c). Hardware logic accelerators can execute code in a few or single cycle and achieve significant speedup. There are two categories of accelerators: *tightly* and *loosely coupled*. In a *loosely coupled* (LC) implementation, an accelerator is positioned outside the processor core, and it operates independently like a co-processor. The LC accelerator logic works similarly to a hardware function call. It is designed to be out-of-core and communicate via a system bus. This prevents the logic from degrading the processor pipeline's performance. As a result, the accelerator logic blocks can be coarse-grained with complex data paths that accelerate a complete LWC kernel implementation. On the other hand, the BUS interface can take care of heavy communication back and forth from the processor to the accelerator logic. LC hardware implementation requires additional hardware space and power consumption for the overall cryptographic solution. This design mostly speeds up a specific cryptographic algorithm with limited configuration space. However, LC hardware implementation is not dependent on the operating system or runtime code. The operation frequency executed in out-of-core hardware can be different

and faster than the core.

A *tightly coupled* (TC) approach involves adding accelerator logic fused into the core pipeline and adding new ISA instructions to use this logic appropriately. This approach adds hardware functional units that can operate in parallel with existing ones to expedite critical parts of an application kernel. Extending the processor hardware to support this additional logic can minimize data processing bootblack and boost the application performance. TC logic is part of the processor pipeline and interacts during pipeline stages. TC logic and the core pipeline can access the same resources, such as the register file, processor state, data memory, etc. New instructions designed as TC potentially stalled in case of data dependency constraints. TC includes special instructions in its ISA to manage operations, typically communicated to software through compilers or low-level libraries. A typical application case for TC is a sequence of bitwise operations that rely on different logic, like LWC permutation. Integrating TC accelerator logic can be challenging for hardware designers. Firstly, it can complicate the CPU design. Secondly, it may pose timing closure challenges since the TC logic is required to meet the same clock-frequency constraints as the processor. Finally, TC accelerators have limited flexibility in terms of portability across different LWC algorithms. From a system-level perspective, decoupling the LC accelerator from the processor provides greater flexibility than the TC model. With the LC accelerator running, the processor can perform other tasks or shut down to conserve energy. On the other hand, the TC model requires less effort from the compiler and toolchain since the LC extension is integrated into the pipeline and extended ISA.

A standard general-purpose processor can only perform single bit-wise logical operations on limited operands per cycle [PS00]. An alternative design strategy is to augment the core processor with custom 'light' instructions that fuse a few base ISAs together, as described in Figure 3(d). Using those new custom instructions with extensible processors that have wide data and instruction interfaces will bring higher levels of data and instruction parallelism. These new instructions, which form a minimal set of additional instructions to the base ISA, are supported by the processor toolchain and can be executed in a single cycle compared to the same functional implementation with only basic ISA instructions. lightweight custom instruction can potentially improve computation block speed and reduce memory space without significantly impacting the processor timing paths and clock rate. Using the ISA extension method, we can implement LWC with fewer instructions as the new instructions are inline with kernel implementation, resulting in a significant boost in performance for long-latency calculations. As those new custom instructions fuse only a few operations together, they can potentially cater to a wide range of cryptographic applications. Designing a lightweight extension ISA to support LWC achieves balance against high hardware costs, requiring expertise in application, instruction set design, and processor design.

3.3 Extensible Customizable Processors

The extensible customizable processor is rooted in the principles of classical Reduced Instruction Set Computing (RISC) architectures. It boasts the ability to be configured in numerous dimensions, as is illustrated in Figure 4.

Firstly, in terms of the basic design, several core architectural key elements can be incorporated to enhance the base processor, including functional units, instruction and data bus width, number of load-store units (LSUs), number of instruction fetch, etc. By carefully selecting and adapting the appropriate hardware block to the data and instruction processing needs, the processor architecture can be tailored to meet the specific requirements of LWC applications, resulting in improved algorithm execution times. Secondly, the base instruction set can be extended by user-defined custom instructions. Performance-critical code portions that require multiple instructions on a generic RISC architecture can be compressed into a single, user-defined instruction to obtain a significant speedup. Thirdly,

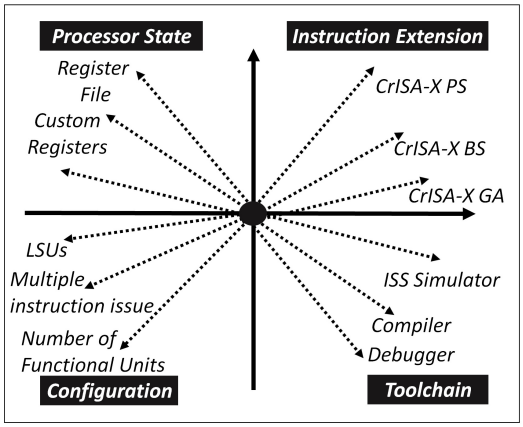

Figure: 4. *Multidimensional Design Space for Customizable Processor*

The processor possesses an array of registers and states, including processor state, custom registers, and register files, that are expandable. By expanding these registers, we can boost the number of registers available or adapt them to varied data formats. This increases storage capacity for essential data, intermediate results, or control signals. With more register, the processor can access and manipulate crucial information more efficiently, reducing the requirement for frequent data transfers to external memory and elevating overall performance. Lastly, implementing a new architecture necessitates a comprehensive update of the toolchain, including the instruction-set simulator, compiler, debugger, assembler, and all associated tools. By doing so, developers can create applications for the processor and evaluate their performance with greater efficiency and accuracy. It is imperative that this update is executed promptly to ensure the seamless integration of the new architecture into existing systems and to facilitate the development of optimized software for enhanced performance.

3.4 Challenges with Instruction Extension

The augmentation of an existing processor's ISA with custom instructions extension has emerged as a popular approach for enhancing application execution speed. Nonetheless, designing a new instruction set architecture is a critical task requiring meticulous planning and consideration. Developing an ISE poses several challenges that must be carefully addressed to ensure an efficient and effective design.

The first challenge in designing an extension is identifying the proper compute block to implement as new instructions. Cryptography tasks involve intensive mathematical computations using bitwise operations, memory access, and flow control, including data movement, manipulation, and boolean logic operations. Breaking down the function execution time is crucial in identifying software bottlenecks suitable for replacement by the new ISA extension acceleration. The design of the instruction extension must show measurable performance gains on the target workload. The instruction extension should focus on accelerating the application code's *hottest* and most frequently executed sections. This strategy is an application of Amdahl's law, which suggests that the useful effect of optimization is maximized by covering the dominant areas of the program code with acceleration. However, just replacing the group of base ISA sequences with new instructions doesn't guarantee a performance boost as there is a certain overhead for register load/transfer associated with a new instruction.

In order to guarantee that newly introduced instructions have a substantial impact on latency, **the second** challenge lies in identifying, within the *hottest* compute block, the specific components that should transfer the new instructions. This analysis should take

into account a few parameters: (1) the input and output register amount and immediate values. this will impact the number of register file ports for input and output ultimately defining the bandwidth to and from the extension hardware. (2) registers width and datatype (3) points in logic data flow where there are no register mutual dependencies,and the instruction extension can be scheduled.

Designing a new ISA extension requires careful consideration of instruction encoding space as it is a limited resource, which makes instruction encoding **the third**. As the encoding space is inherent in the processor bus width, the encoding scheme should be efficient, compact, and compatible with the existing encoding formats. While designing a new instruction, the challenge is to allocate the minimum space, considering the opcode, number of registers, immediate, and states, while achieving the maximum computation. This challenge will be addressed later in this paper by categorizing instruction according to encoder space and computing level. With a modular processor, we extended the instruction bus to be large, with more space for long instructions. Moreover, the VLIW option for partitioning the wide instructions into custom slots will create a multi-slot VLIW machine, each slot capable of executing one of a set of operations in parallel. We must carefully analyze and decide which instructions will get into groups based on the dependencies between instructions. There can be dependencies between instructions in a group, but they can cause pipeline hazards. This analysis should be per the LWC algorithm, and once we reach a solid optimal design, we should go one step further and tune which instructions will get into groups from joint algorithm aspects.

The power-performance-area (PPA) trade-off is **the forth** challenge. New instructions have direct implications for processor hardware. The processor area and timing are the most viable processor characteristics that get impacted and must be taken care of carefully. Heavy combinational logic embedded in the instruction logic impacts the processor area and can cause long-timing paths. For instance, new instructions may involve adding more function units that should work sequentially. Moreover, in the case of a few institutions running in parallel, there is a potential scenario in which more than one instruction needs to access the register file. Therefore, the register file design would have many ports, potentially muxing together, creating a more complicated access path, being prominent in the area, and possibly bringing timing constraints. Instruction extension design must also consider the cost and power implications of adding new instructions to the processor. New instructions can increase the processor's size and complexity, leading to higher costs and power consumption and decreasing the max processor frequency. A well-designed ISE can result in a lower footprint in the hardware area, i.e., gate-gate.

3.5 Profiling

Identifying and adding instruction extensions to existing base processors has been extensively studied. A significant body of literature has been published on this subject over the years [CMS07, CZM03, KR06, YM04, CHKP98]. The typical approach involves starting from the application code, identifying a large set of ideal ISE candidates from a compiler intermediate or from generated convex sub-graphs from the data flow graph (DFG), and then searching for the optimized ISE. Although, those methods satisfy the processor I/O constraints, the outcome is limited for specific processor architecture. It does not consider the various options to extend the processor hardware for DLP and ILP to gain more speedup for computation block across more than one algorithm. Our philosophy is different, and we used a profiling analyzer based on the GNU gprof [FS88] to identify "hotspot" functions across a set of LWCs. The "hotspot" function consumes a substantial amount of the execution time of the specific algorithm. The profiling step gathers data about the application instructions executed with "hotspot" functions from the actual execution of the program. Instruction Set Simulator collects profile data from the target application in a non-intrusive manner so that it does not affect the actual functioning of the original code.

The profiling tool records the execution of every instruction in a cycle accurate manner. We use different types of analysis outputs to understand our program's performance. The flat profile shows how many times each function was called and how many cycles it used. This information helps us identify the functions that require the most computation. The call graph displays the amount of time taken by a function and its associated functions. By analyzing this information, we can identify functions and the function assembly code, that may not take much time but trigger other functions that consume substantial amounts of time. Finally, we use line-by-line profiling, assigning histogram samples to individual assembly lines of source code instead of functions.

Profiling data guides the block computation selection of which translate to CrISA-x instructions set. The profiling step of the process is performed using a cycle-accurate instruction set simulator (CA-ISS) that supports both the entire base ISA and the introduced new extended CrISA-x instructions. The CA-ISS model simulates the processor at an abstraction level between the RTL and the functional model. It presents the architectural details necessary for the processor dimension to evaluate its performance capabilities in advance. A CA-ISS runs on a host machine to mimic the functional behaviors of instructions running on target hardware. CA-ISS allows the estimate of the execution time of software in a cycle accurate way and validates a system even when its target hardware does not yet exist or is not available. A CA-ISS reflects a target hardware logic and conforms to the cycle-by-cycle behavior of the target system. It can produce the same number of cycles as the actual execution on the target hardware. We used the execution statistics gathered by the profiling tool, combined with instruction timing values, like cycles required to execute each instruction, to give an accurate number of cycles required to execute the application on the base ISA and with CrISA-x instruction. The speedup of the enhanced ISA version relative to the base ISA version can be computed and compared fast. The profiling tool determines how many cycles are required to emulate each instruction executed from the full ISA, which is not supported by the base ISA. This is calculated by counting the number of base instructions executed by software modules used to emulate these instructions and scaling the results according to the number of clock cycles that are required to execute each base instruction executed.

3.5.1 CrISA-X illustration

An illustrative example will be presented in this section to provide a comprehensive understanding of the CrISA-X methodology. Furthermore, we will show the concept of issuing multiple instructions, including CrISA-X , on the same cycle. Later in this paper, we will delve deeper into these topics, examining them in greater details.

To illustrate these, consider the *ASCON permutation* described in listing 1, where profiling numbers (mark as (xK)) indicate that significant application's execution time is spent computing the c-code statement as part of permutation flow.

Listing 1: Basline c-code Ascon permutation

```
1 (10K) x0.e = x0.e^x4.e;
2 (10K) x0.o = x0.o^x4.o;
3                 .........
4 (10K) x2.e = x2.e ^ (~x3.e & x4.e);
5                 .........
6 (10K) x3.e  = x3.e ^ ((x3.e>>4)|(x3.e<<28));
```

The provided C code explains how the permutation works in both linear (lines 1-4) and nonlinear parts. For example, in the linear part (e.g., lines 1 and 2) section, two data input registers, one odd (o) and one even (e), are XORed, which requires two base instructions, not counting the instructions required for loading and storing the data. Later in the code,

in the nonlinear section, an intermediate value goes through a bitwise NOT operation, and then a bitwise AND operation is performed. The result is then XORed to obtain the final value, which takes three base instructions. In line 6, an instruction involves rotating a 32-bit input by performing a circular shift, followed by XORing the result to obtain the final value. This operation comprises two basic instructions.

Listing 2: Crisa-X code Ascon permutation

```
1    XOR2(x0.e,x4.e,x0.o,x4.o);
2                .........
3    x2.e = XORNOTAND(x2.e,x3.e,x4.e);
4                   .........
5    x3.e = XOROT(x3.e,x3.e,4);
```

As described in listing 2, these c-code and corresponded assembly instruction can be fused into **three** different single-cycle fused instructions called XOR2, XORNOTAND, and XOROT. Each instruction uses one or two input operands from the processor register file to compute the output value and save it back to the register file. The semantics of those instructions and others will be described later. The software toolchain updates as-well and recognizes the new instructions in addition to the processor base ISA. Fused instructions, as described in listing 2, are significantly cheaper in area and have a limited access port for the register file because they operate on restricted data sets. A computation that previously took seven assembly bitwise operations, each taking seven cycles per iteration on a base processor, now only requires three cycles, resulting in nearly a 3x performance increase. This kind of instruction will be defined later on in this paper as Crisa-X *Generic Atomic*.

Instructions extensions that involve more input and output operands, and therefore implement more involved computation can be designed with logic that spans multiple clock cycles. Such instructions, defined later in this paper as Crisa-X *Specific-Block* and *Specific-Procedure* can be fully pipelined and can be issued back-to-back or in an iterative manner. Fully pipelined instructions achieve higher performance because they can be issued back-to-back, but they may also require extra implementation hardware to store intermediate results in the instruction pipeline.

Creating instruction extensions for customized processors goes beyond the fusion of several operations into a single instruction. We can improve instruction-level parallelism (ILP) by bundling multiple instructions into a single long instruction word (LIW) and executing them simultaneously in parallel. In VLIW, the compiler explicitly schedules multiple independent instructions into a single, long instruction word, which is then issued to the processor, executing all instructions in parallel. Different types of instructions can be used, including base instructions and dedicated CrISA-X instruction, which can be freely intermixed to optimize performance. The instructions are divided into slots. Each slot represents a group of instructions that can be executed in parallel within a single clock cycle. Slots are not equally sized. Any combination of the operations allowed in each slot as far as the encoding space allowed. With multiple instructions bundled into a single slot, the processor only needs to fetch and decode one instruction word per cycle. This simplifies the instruction fetch and decoding stages, reducing the complexity and overhead of these stages in the processor's pipeline. Slots allow the processor to efficiently utilize its available resources, such as functional units and registers

Consider the Ascon permutation example. The inner loop performs the actual computation using three-fused instructions. It also utilizes two L32I load instructions and one S32I store instruction to move the data as needed[3]. Additionally. listing 3 describe the

[3]Noted that the term "32" in this context pertains to operands with a bit length of 32, while "I" denotes an immediate value integrated directly into the instruction.

ILP way to accelerate this code. In this example we are using a 64-bit instruction format with one slot for the load and store instructions two slot for the computation instructions.

Listing 3: Ascon permutation slot setting

```
format flix3 64 {slot0, slot1, slot2}
slot_opcodes slot0 {LD_INST,ST_INST}
slot_opcodes slot1 {LD_INST, XOR2}
slot_opcodes slot2 {XOR2,XORNOTAND,XOROT}
```

The first declaration creates a 64-bit instruction and defines an instruction format with three opcode slots. The last three lines of code list base ISA instructions to be available in each opcode slot defined for this processor slot configuration. Note that all the instructions specified are predefined, some of them are core processor instructions, and others are CrISA-X. For this example, the compiler can compile the source code using slot extension definitions without any changes to the generated C/C++ program. The generated assembly code for this processor implementation is described in listing 4:

Listing 4: Ascon permutation assembly VLIW look

```
loop:
{$LD_INST;LD_INST;XOR2}
{$ST_INST;LD_INST;XORNOTAND}
{$ST_INST;XOR2; XOROT}
```

A computation that requires fourteen cycles (nine cycles for operation and 5 cycles for load-store) per iteration/data input on a base processor, with multi-issue and extended ISA only requires three cycles per iteration, resulting in nearly a 5x performance increase. In the context of supporting multiple algorithms and achieving pareto-optimality, it is necessary to establish an instruction slot format that ensures no algorithm can be improved without negatively affecting another algorithm. This is accomplished by striking a balance between the various algorithms, such that we add ISEs for each algorithm so that it can perform well on its own, and we increase as little as possible the complexity of these new ISEs so that they can be shared as much as possible between algorithms; clearly, without increasing cost and area significantly.

4 Design

In designing CrISA-X for performance-acceleration of LWC code, we have adopted a two-fold approach comprising the design of an instruction extension and processor extension. This approach optimizes LWC code performance by leveraging the ISA extension's capabilities and the extended RISC processor. By doing so, we can extend a group of custom hardware resources to support a new instruction set. Adopting a new instruction set and removing constraints on processor architecture minimize the processor area and maximize the utility of ISE designs. The relationship between software and hardware processors is illustrated in Figure 5. In the case of an extendable processor, CrISA-X instructions are defined and formatted as part of implementing the software layer for LWC. The base ISA is the sole means of interaction between applications and the underlying hardware at the lower layer. The CrISA-X Extended ISA specifies additional instruction using function unit extensions and registers throughout the processor's extended ILP and DLP hardware.

4.1 Extending Processor Architecture

The processor architecture can be extended in several dimensions: **(1)** The scalar Register File has been extended to support multiple input and output operands with multi-read

and write ports. **(2)** The width of the processor's data and instruction bus has been extended to allow for simultaneous transfer of instructions and operands, reducing memory accesses. **(3)** The processor fetch and execution units have been designed to execute multiple instructions in parallel by packing them into a single long instruction word. This parallel execution allows for improved performance by exploiting ILP. **(4)** Extending the processor with additional load/store hardware to allow multiple memory operations to be executed in parallel. **(5)** Additional constant, state, and array registers have been added to the processor's register space. These single-cycle access registers are built along the processor pipeline to provide dedicated or temporary storage that can be shared and used between new instruction sets. **(6)** Instruction extension includes fused and SIMD operations. The first creates new instructions composed of several simple bitwise operations and the second designs instructions that simultaneously perform operations on multiple data elements.

4.2 Extending Processor Instruction Set

Various implementation styles are available when designing an accelerator to support cryptographic algorithms on a general-purpose processor. These techniques can be algorithm-agnostic or algorithm-specific and use hardware, software, or a hybrid approach using Extended ISA. The approaches of Extended ISA can be categorized based on the number of operands, amount of computation, and instruction encoding space. These are dubbed *Genric-Atomic, Specific-Block, or Specific-Procedure ISA* as describe in Figure 6.

The **generic-atomic** (GA) approach is an instruction extension that encapsulates several bitwise operations happening in series, which are not available as part of the base ISA. CrISA-X GA usually have two to four operands, with part of them serving as the destination. The instruction encoding is kept short. These instructions are algorithm-agnostic, allowing them to be applied to any type of algorithm. This versatility makes them suitable for implementing a vast spectrum of LWC implementation. Their re-usability across different contexts and applicability to a wide range of cryptographic algorithms ensures they can balance performance and flexibility in general cryptographic implementations.

The **Specific-Block** (SB) family refers to an instruction extension that encapsulates multiple bitwise operations and accomplishes large computational tasks that might executed within a multi-cycle. CrISA-X SB instructions usually involve a substantial number of

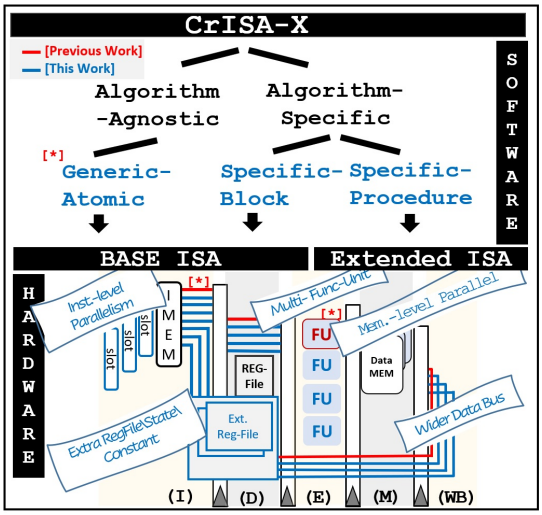

Figure: 5. *CrISA-X Classification together with Processor Extensions. Previous work (red) described in Table 2*

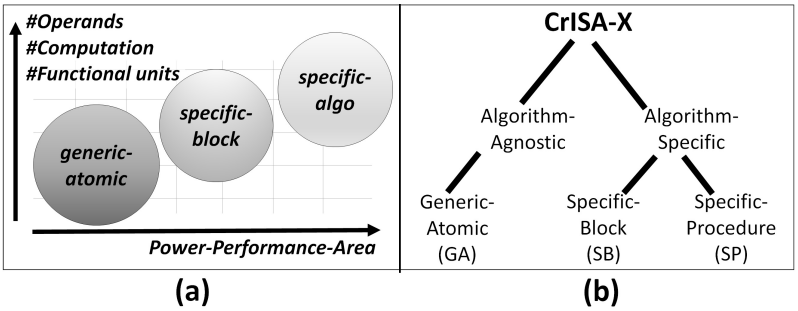

Figure: 6. *(a)CrISA-X trade-offs Power-Performance-Area(PPA) vs Number of operands, Functional-Units (FUs) and amount of computation (b) CrISA-X categories*

operands, typically ranging from four to eight, with one or more serving as the destination. The instructions use few parallel execution units to support narrower permutation building blocks that require a significant amount of computation.

The *Generic-Atomic*, GA, and *Specific-Block* approaches prioritize certain operations within the scope of individual permutations. The **Specific-Procedure** (SA) approach, on the other hand, encompasses a comprehensive sequence of operations that span across the entire permutation block as an interdependent single unit.

The **Specific-Procedure** is an instruction extension designed as tightly coupled acceleration logic that spans over more than a single cycle. The logic represents specific tasks, such as Ascon or Keccak permutation, and targets extensive computation. Instructions from this family involve multiple operands, often more than eight, and one or more serve as the destination. The length of instruction encoding is wide. The CrISA-X SA has been designed to avoid register spills to stack memory, i.e., temporary load and store operations during the permutation round. The extensive computation behind a *Specific-Procedure* instruction allows for significant performance improvement when executing cryptographic LWC. However, the hardware implementation necessitates an increased gate count and impacts the maximum frequency of the processor pipeline. To minimize the impact on the critical path, we divide the *Specific-Procedure* logic computation into two cycles and retime the signals as necessary.

Readers which are more focused on understanding bottom-lines and conclusions, and less interested in the deep technical details per each algorithm, can continue reading from Section 5.

4.3 ASCON

The *Ascon* submission presents a detailed analysis of several algorithms, including Ascon-128, Ascon-128a, Ascon-80pq, Ascon-Hash, and Ascon-Hasha, as reported in the work of Dobraunig et al [DEM+19]. However, the primary focus of the submission is on the Ascon-128 algorithm and its kernel. This kernel is composed of the pa and pb kernel permutations, where "p" denotes a single permutation, and "a" and "b" represent the number of rounds. The *Ascon* permutation, denoted as "p", comprises three layers: Round constant addition, Substitution layer, and Linear layer. This permutation operates on a 320-bit state, divided into five 64-bit words $\chi 0$, $\chi 1$, $\chi 2$, $\chi 3$, and $\chi 4$. In the round constant addition layer, $\chi 2$ input is XORed to the round-specific constant. The substitution layer is typically implemented using logical ANDs, XORs, and NOTs, while the linear layer employs shift left and XOR operations.

CrISA-X description: To demonstrate the CrISA-X instruction set, the Ascon-128 algorithm is used in a bit-interleaved manner on a 32-bit processor. Each 64-bit word of the state is divided into two 32-bit words. One word contains the bits at even positions, and the other word contains the bits at odd positions. This representation can

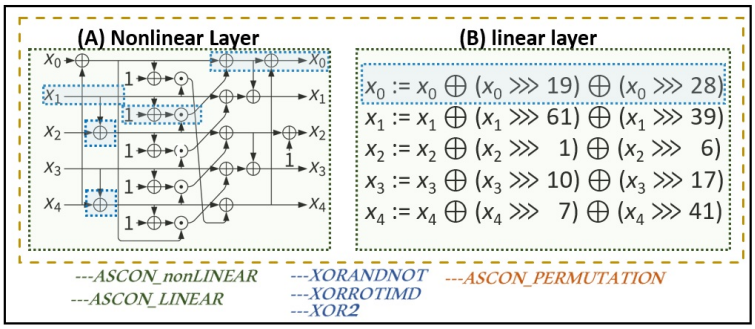

Figure: 7. *Ascon's Permutation steps*

```
XORNOTAND at, as1, as2
    at ⇐ at xor (( not as1 ) and as2)

XORROTIMD at, as, imm
    at ⇐ at xor (at≫imm or (at≫32−imm)

XOR2IMD at1, at2, imm1, imm2
    at1 ⇐ at1 xor imm1
    at2 ⇐ at1 xor imm2

XOR2 at1, as1, at2, as2
    at1 ⇐  at1 xor as1
    at2 ⇐  at2 xor as2
```

Figure: 8. *CrISA-X Generic Atomic instruction for Ascon permutation*

exploit the 32-bit register extension without paying latency overhead for accessing a 64-bit register file. However, this comes at the expense of conversions between the bit-interleaved representation and 64-bit representation whenever data is injected into or extracted from the state. CrISA-X hides this latency as part of the instruction logic as it includes bit extraction from the state as a pre-operation.

The bitwise logic for the non-linear and linear layers is illustrated in Figure 7. These layers involve logical operations performed on 64-bit words, which can be split into two operations on 32-bit chunks. The substitution box requires extensive base instructions, split between 44 XORs, 12 NOTs, and 10 ANDs, 19 shifts, and 19 ORs, spread between even and odd parts. In total, there are about 104 native instructions per round. Depending on the amount of computation it encapsulates, the various uses of the CrISA-X instruction are represented by different colors, as shown in Figure 7. It takes 16 instructions to implement the *generic-atomic* in the linear layer, while the non-linear layer requires 14 CrISA-x instructions, resulting in a total of about 30 CrISA-X instructions. The permutation layer can significantly speed up the process by a factor of 5x. Figure 8 describes the *generic-atomic* semantic use to speed up *Ascon permutation*. Immediate values can be used to specify rotation amounts. When implementing the S-box with the *specific-block* approach, only the new 8 instructions' are needed. The linear layer requires the new 10 instructions' when using the wide instruction bus approach. With this, the permutation layer can achieve a speed-up of up to 8 times faster. while implementing with *specific-procedure* brings 10x factor. Figures 9, 10 describe the appropriate semantic for block level.

```
ASCON-LINEAR ast0e, ast0o,..., ast4e, ast4o
begin
 temp$ast0e⇐ast0e xor rot32(ast0o,4)
 temp_ast0o⇐ast0o xor rot32(ast0e,5)
     ..............................
 ast0e⇐temp$ast0e xor rot32(temp$ast0e,10)
 ast0o⇐temp_ast0o xor rot32(temp_ast0o,10)
end
func: rot32(in, o f s) = (a ≫ o f s)or(a ≪ 32 − o f s)
```

Figure: 9. *CrISA-X Block-Specific for Ascon Linear layer*

```
ASCON-nonLINEAR ast0e, ast0o,..., ast4e, ast4o
begin
 tmp_e0⇐ast0e  xor ast4e
 tmp_e0⇐ast0o xor ast4o
 mid_e0⇐tmp_e0 xor
     xornotand(tmpe0, tmp_e1, tmp_e2)
mid_o0⇐tmp_o0 xor
     xornotand(tmp_o0, tmp_o1, tmp_o2
     ..............................
 ast0e⇐mid_e0 xor mid_e4
 ast0o⇐mid_o0 xor mid_o4
end
     func: xornotand(a, b, c) = a ⊕ (˜b&c)
```

Figure: 10. *CrISA-X Block-Specific for Ascon Nonlinear layer*

4.4 Elephant

The mode of Elephant is a nonce-based encrypt-then-MAC construction, where encryption is performed using counter mode and message authentication using a variant of the protected counter sum MAC function. The Elephant submission specifies a few AEAD algorithms, Dumbo, Jumbo, and Delirium [DEM+19]. To focus we choose one and specifically concentrate on Delirium: namely Delirium = Elephant-Keccak-f[200].

The delirium permutation was selected because the *keccak* permutation is widely used in other lightweight algorithms. The *Keccak-f*[] permutation is the fundamental building block for SHA3, and for two candidates in the CAESAR competition, namely KEYAK and KETJE [BDP+14, BDPA15, Dwo15, RS16]. The permutation function is computed using 25 Keccak states, marked as A0 to A24. Each state is 8 bytes and the iteration is performed over a set of 8 rounds. The state data can be saved using extended register files, resulting in fewer data movements than only exploiting register files in the baseline core. In brief, five *keccak* steps are defined by five operators symbolized by $\theta, \rho, \Pi, \chi, \iota$. θ , The most extensive ones are illustrated in Figure 11. θ step requires XOR between five states, then rotating left by a factor of one to get Di states. The last step is Di states XORs with Ai states. ρ step is for left rotate all lanes in the state by a fixed offset. For efficient implementation of the ρ step, left rotation by the *Keccak* constant tables are saved as the hardwired processor state. In the Π step, all lanes in the state are transposed in a fixed pattern. This step can be done using only MOV instruction. Next, is χ, where each bit of the lane is non-linearly combined with the bits of nearby lanes using AND, XOR, and NOT instructions. Last is ι: a simple XOR of constants into a single lane. An in-depth explanation of the *keccak* permutation can be found in the reference [BDPVA09]. The state of a 5x5x8bit *keccak* can be viewed as w-slices of 25 bits each or as five planes with

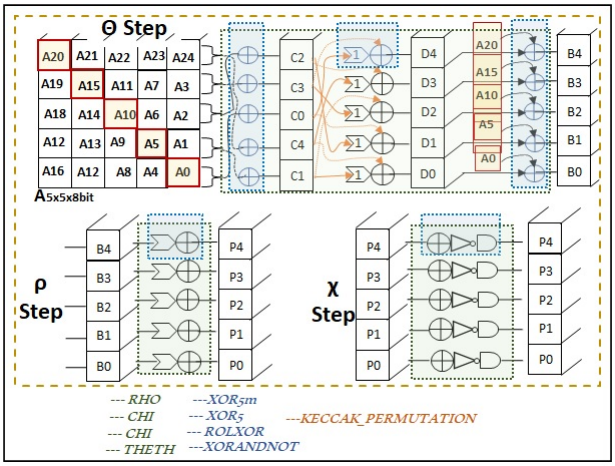

Figure: 11. *keccak* permutation θ, ρ, and χ steps

five lanes of 8 bits each. Our research employs a layered design that groups four states into a single 32-bit input. The CrISA-X instruction logic effectively hides the latency penalty by handling data separation back to an 8-bit representation as part of the instruction. All the steps involved in the *keccak* algorithm, except for Π and ι, can be efficiently performed using the CrISA-X instruction from various categories. The Π and ι steps mainly involve moving data from one register to another and do not have a significant impact on latency.

CrISA-X description: To showcase the CrISA-X instruction set, we decided to store the *keccak* in the extended register file. We wanted to work only with full 32-bit variables instead of 8-bit variables. Thus, we replaced all byte variables with uint32 variables, which resulted in a state representation that consisted of an array of seven 32-bit words. The *Keccak* permutation steps are illustrated in Figure 11. Each category of CrISA-X instructions is represented by a different color. For the *generic-atomic* instruction set, we use *XOR5* and *XOR5m*, the last one involving immediate value as inputs. These institutions perform *XOR* operation between five operands or immediate and produce the result in a single cycle. In addition, two new fusion instructions have been designed: ROLXOR, which brings rotate-left and XOR together, and XORANDNOT, which fuses XOR with the result of NOT-AND operations. Figure 12 presents the pseudo-code for *Keccak* and demonstrates the use of *specific-block* and *specific-procedure* instead of original operations. The instruction set for the *block-specific* implements the full θ, ρ, and χ *keccak* steps in a single instruction. 'Block' operation approach is possible for custom processor hardware where the state can be stored in a custom register file. Instruction logic can efficiently read the state in a lane-wise or fashion, and operate, depending on the *keccak* step. The *keccak* state is composed of 200 bits, which can be arranged in seven 32-registers. This allows for the creation of a hardware logic that can perform all permutations in one instruction, known as a *specific procedure*. The input and output for this instruction will be seven 32 registers from the extended register file. The logic of the instruction includes five Keccak operators - $\theta, \rho, \Pi, \chi, \iota$. θ executed back-to-back.

4.5 GIFT-COFB

GIFT-128 is a member of the GIFT block cipher family. It uses a substitution-permutation network (SPN) with the key length and block size set to 128 bits. The algorithm and its implementation are described in reference [BCI+19, BPP+17, MSA22]. The function *giftb128*() is the most cycle-intensive function, and it consumes 90% of all the LWC reference code and implements a 40-round block cipher. Input is a 128-bit cipher state expressed as four 32-bit slices S0, S1, S2, and S3. The round function consists of quintuple

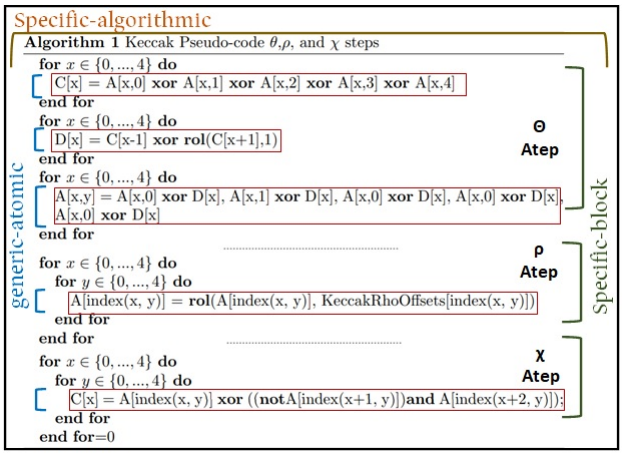

Figure: 12. *Keccak* pseudo code along with CrISA-X classification

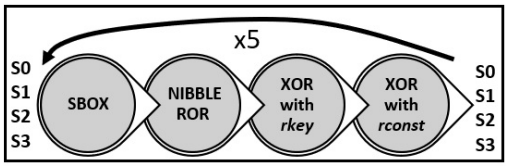

Figure: 13. Quintuple steps description

```
XORAND (XOROR) at, as1, as2
    at ⇐ at xor ( as1 and (or) as2)
ROTOR at imm1,imm2,imm3,imm4
    at ⇐ ((at≫imm1) and imm2)) or
        ((at ⇐ ((at≪imm1) and imm4))
XORSHIFTAND at,as1,as2,as3 imm
    at ⇐ (as1 xor (as2≫as2)) and imm
```

Figure: 14. *CrISA-X Generic Atomic instruction for GIFT-COFB permutation*

steps repeated eight times. Each quintuple step is a group of operations that happen in serial and are repeated five times. the steps are: SBOX, Nibble\byte\half-word rotate right, then result in XOR with round-key and last with round-constant, describe in Figure 13.

CrISA-X description: The *GIFT-128* cipher uses a 32-bit register nativity. Thus, the CrISA-X instruction is designed to take advantage of this and focuses on fusing bitwise operations into a single instruction that can be executed in a single cycle. Three new fusion instructions have been developed for the *Generic Atomic* set: XORAND, ROTOR, and XORSHFTAND. Instructions semantics are described in Figure 14. They are used as building instructions to speed-up core computation. With the *Specific-Block* set, we design interactions that take as input the four S0, S1, S2, and S3 and compute the full SBOX. The same is performed for other instructions: ROTOR, ROTXOR, and SWAPMOVE. For the *Specific-Algorithm*, we design a single interaction that calculates QUINTUPLE round in a single instruction.

4.6 Grain-128AEADv2

Grain-128AEADv2 is an authenticated encryption algorithm with support for associated data. we logically divide the cipher into two phases. The first phase is the loading phase, in

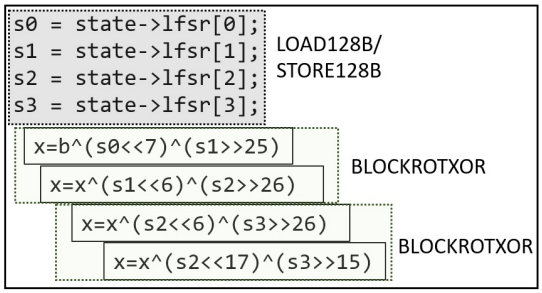

Figure: 15. *GRAIN128 extract a 32-bit word from a 64-bit register with Specific-Block*

```
ROTXOR at, as1, as2, imm1, imm2
    at⇐as1 xor(as1≫imm1)xor(as2≪imm2)
```

Figure: 16. *CrISA-X Generic Atomic instruction for Grain128 core*

which the shift registers are loaded and initialized with the key and the nonce. Second, the cipher enters the running phase, in which pre-output is generated both for encryption and authentication. A pre-output generator is constructed using a linear feedback shift register (LFSR), and a non-linear feedback shift register (NFSR), which are 128 bits each, and generates a stream of pseudo-random bits used for encryption and the authentication tag. The authentication phase consists of a shift register and an accumulator. The algorithm performs several functions, but the keystream generation, authentication, and loading phases require the most cycles. In these phases, the system works with 32-bit words and uses various bitwise operations such as shifts, ANDs, and XORs to calculate feedback bits for LFSR and NFSR and generate the new key stream.

CrISA-X description: We created fusion instructions composing multiple operations into one to speed up such operations. For the *Generic-Atomic* set, we use the ROTXOR instruction, which its semantics is describes in Figure 16. This flexible instruction can take three 32-bit operands as input, one operand as output, and two immediate values. By shifting the bits individually, we can extract a 32-bit word from a 64-bit word in many different ways. The authentication mechanism in Grain-128AEADv2 is based on a universal hash function, introduced in [WC81]. For hashing, the message is multiplied by a Toeplitz matrix where the Toeplitz matrix multiplication is implemented using XOR the shift register with the accumulator for each single bit in the byte that we are authenticating and shift in the keystream byte that has been retrieved before, for that we use *Generic-Atomic* ROTXOR, SHIFTLXOR and XORAND operations. Extending the processor data bus to 128 bits, we can load and save all state data back to memory with a *Spcific-Block* instruction called LOAD128B and STORE128B. After loading the data, it can be divided into four 32-bit variables: S0, S1, S2, and S3. These variables are saved on the processor's extended register file and available for both CrISA-X and base instructions. The algorithm extracts a 32-bit word from a 64-bit register through independent registers, so we can fuse them in a single instruction. We utilize this data group and create *Specific-Block* instructions calculating the BLOCKROTXOR as illustrated in Figure 15. For *Specific-Algorithmic*, we use the same processor extension to design a single instruction for the entire core computation of GRAIN128_NEXT_KEYSTREAM.

4.7 PHOTON-Beetle

The PHOTON-Beetle is an authenticated encryption and hash family, that uses a sponge-based mode Beetle with the PHOTON256 being the underlying permutation [BCD+19]. We focus on the primary algorithm PHOTON-Beetle-AEAD[128] bit sliced version. The

```
XOROR2(XORAND2) at1, at2, as1, as2, imm1, imm2
    at1⇐at1 xor as1, at2⇐at2 or(and) as2
```

Figure: 17. *CrISA-X Generic Atomic instruction for PHOTON-Beetle core*

PHOTON256 permutation operates on an internal state of 256 bits. Bit-slicing code slices the 256-bit data into an 8x32 words array, organized in 8 rows. The permutation is composed of 12 rounds, each applying 4 round functions: these are *AddConstant*, *SubCells*, *ShiftRows*, and *MixColumnsSerial*. *SubCells* applies a 4-bit S-Box to each of the 64 4-bit cells. *ShiftRows* rotates the position of the cells in each of the rows, and *MixColumnSerial* linearly mixes all the columns independently using serial matrix multiplication.

CrISA-X description: The *AddConstant* step performs the XOR operation on each word out of eight with the correct constant. To make this operation faster, we utilize the *Generic-Atomic* XOR2 instruction to double the throughput. Additionally, we save the round constant as a hard wire table. This helps to minimize the new instruction encoding as we only use an index reference to the constant table. To fully utilize the capabilities of the 32-bit processor and its extensions, we used the bit-slicing code for PHOTON256, especially this can be seen for the *SubCells* step. The basic concept of bit slicing involves converting the code into a sequence of logical operations such as AND, XOR, OR, and NOT, which can be executed on a standard- processor ISA [SOM]. For the *Generic-Atomic* instruction set, we have combined and optimized several operations into a single instruction, such as XOR2, XOROR, and XORAND as described in Figure 17, that executes as a single cycle. During the *ShiftRows* steps, we use the ROTOR instruction that rotates left by immediate positions for each row per BYTE. As CrISA-X instructions use 32-bit inputs\output operands, ROTOR instruction operates on four bytes in parallel, quad the throughput. The data is separated internally as part of the instruction to hide any additional latency penalty. PHOTON 256-bit state can be divided into two groups, with each group of 128 bits mapped to four 32-bit registers. Since the input states are independent and the SBOX non-linear operation is performed per 4-bits, we build a parallel computation block to produce a 4-bit x 32 non-linearity operation in a single cycle. We utilize this data group and create *Specific-Block* SBOX instruction to accomplish this. The same concept was implemented for *ShiftRows* step where we rotate 16 bytes in parallel. For *Specific-Algorithmic*, we utilize a processor extension to create a single instruction that carries out the core computation of SBOX for the entire 256-bit input. *Specific-Algorithmic* PHOTON_PERMUTATION uses 8x32bit register as inputs\output. This greatly impacts the register file as we significantly increase the number of ports. combining *Specific-Algorithmic* instruction with wide data load and store instructions (referenced as LOAD128 and STORE128), we achieve a significant speedup in the permutation.

4.8 Romulus

The Romulus submission describes the AEAD algorithms Romulus-N/M/T and the hash function algorithm Romulus-H. The computational complexity and latency of the kernel in Skinny-128-384-plus tweakable block cipher is high hence our focus was on it. This kernel operates on an internal state of 128 bits, representing a 4x4 matrix of bytes, similar to AES. The round function is composed of five operations in the following order: *SubCells*, *AddConstants* (AC), *AddRoundTweakey* (ART), *ShiftRows* (SR), and *MixColumns* (MC). *SubCells* applies an 8-bit S-box. The *AddConstants* operation XORs some round-dependent constants to the first column of the state. *AddRoundTweakey* extracts eight bytes from the tweakey state and XORs them to the state, whereby the bytes are permuted and updated with simple LFSRs. *ShiftRows* rotates the bytes of the state row-wise to the right by 0, 1, 2, and 3 positions, similar to the *ShiftRows* Right-rotate line i by i positions. Finally,

```
SWAPMOVE at1, at2, as1, as2
    tempreg⇐(at2 xor(at1≫as2)) and as1
    at1⇐at1xor(tempreg≪as2)
    at2⇐at2xor(tempreg)

XORNOTOR at1, as1, as2
    at1 ⇐ at1 xor ( not(as1 or as2))
```

Figure: 18. *CrISA-X Generic Atomic instruction for Romulus*

MixColumns multiplies each byte-column of the state multiplied by a binary matrix. The skinny opt32 implementation relies on fix-slicing with QUADRUPLE ROUND routine [AP20]. Fix-slicing mainly consists of fixing the bits within a register (or slice) never to move and adjusting the other slices accordingly so that the proper bits are involved in the *SubBytes* operation. This implementation doesn't compute the *ShiftRows* operation. Some masks and shifts are applied during the *MixColumns* operation so that the proper bits are XORed together. Furthermore, the row permutation within *MixColumns* and the bit permutation at the end of the Sbox is omitted. The rows are synchronized with the classical after only four rounds.

CrISA-X description: The input for the *skinny128* core computation is a 128-bit state split into four 32-bit registers representing a 128-bit state. Core executing QUADRUPLE ROUND routine ten times. QUADRUPLE ROUND includes XOR and OR operation on each 32-bit state register, swap bit and mask, and add round key. We use a *Generic-Atomic* XORNOTOR instruction that fused the bitwise operation to a single instruction. We use *Generic-Atomic* XORSHFTAND instruction to swap the bits with masks. *Generic-Atomic* XOR2 instruction helps to double the throughput of adding round-key constant as this is done for each 32-bit state. Using the *Specific-Block* approach, we have implemented the QUADRUPLEROUND routine as a new CrISA-X instruction. This instruction takes the 128-bit state matrix split into four 32-bit words (quads) and the relevant tweak keys (TK1, TK2, TK3) and offsets. It implements this part of the encryption for a single block logic as a series of XORNOTOR, SWAPMOVE, followed by XOR2. The instruction we implement for *Specific-Procedure* is the new CrISA-X instruction, which includes the computation of all SKINNY128_384_PLUS and involves ten QUADRUPLE ROUND calls.

4.9 Sparkle

Sparkle is a group of cryptographic permutations, each of which operates on a different block size - 256, 384, or 512 bits [BBdS+19]. The principal member of Schwaemm is Schwaemm256-128, which takes a 256-bit nonce and a 128-bit key as input and produces a 128-bit authentication tag as output. This is the primary submission for the AEAD functionality, and thus, we will focus on that code variant. The Sparkle permutations are created using the following primary components: A non-linear layer uses the ARX-box block cipher called Alzette [DPU+16]. This layer relies only on Addition, Rotations, and XORs (ARX paradigm). It can be understood as a four-round iterated block cipher where the rounds differ in rotation amounts. After each round, the 32-bit constant (i.e., the key) is XORed to the left word. Since Alzette has a simple Feistel-like structure, the inverse computation is straightforward and reuses the same CrISA-X instructions. A linear diffusion layer utilizes a Feistel function, which involves rotating the rightmost three branches of the state, followed by a swap between the leftmost three branches and the rightmost three branches. This design of the linear layer enables the creation of a long trail by leaving half of the words in the state unaltered.

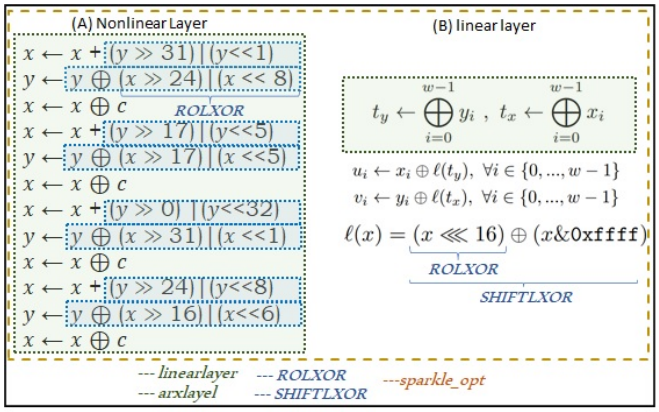

Figure: 19. Sparkle permutations with (a) linear layer (ARX-box) (b) diffusion non-linear layer

CrISA-X description: The instructions for using Sparkle permutations with CrISA-X are depicted in Figure 19. ARX-box relies on additions, rotations, and XORs, as described in the ARX diagram. These operations can easily be fused into a single instruction of the *Generic-Atomic* type. We use SHIFTLXOR and ROLXOR instructions that was defined and described earlier in the paper. By utilizing two additional Block-Specific instructions, we can perform linear and non-linear computations separately, managing the amount of computation logic for each one of those instructions. The *Specific-Procedure* instruction named "sparkle opt" involves computing all Sparkle permutations.

4.10 TinyJAMBU

TinyJAMBU is a variant of JAMBU with a 128-bit key, 128-bit state size, and 32-bit message block size. We are concentrating on the TinyJAMBU-128 algorithm for a kernel represented by the keyed permutation Pn. There are two possible number of iterations (for n): 640 or 1024 times. The TinyJAMBU permutation uses a non-linear feedback shift register (NFSR) with a feedback path that consists of four bitwise XORs and a bitwise NAND. The only non-linear operation in TinyJAMBU is this feedback path. The state-update operation in TinyJAMBU is crucial and significantly impacts performance. The function takes in three inputs: the 128-bit state, the number of rounds, and a key. The function computes the state update for each iteration by performing shifts, XORs, and bit-wise NAND operations. This process is repeated in four blocks, each dedicated to a different state. The state-update function is used during the initialization phase, key, nonce setup, processing of the associated data, and encryption and decryption steps.

CrISA-X description: The state-update function implementation involves dividing the 128-bit state into four 32-bit registers: {`state||state||state||state`}. This enables simultaneous computation of 32 rounds of the permutation, where XOR and NAND operations are performed on 32-bit words. These operations involve a word from the state, a word from the key, and four extracted words from specific positions in the state. The extraction process simplifies obtaining a single 32-bit word by combining two adjacent 32-bit state words using *Generic-Atomic* ROTXOR instruction. To speed up the state-update function, we utilize XOR5, an extension of XOR2, and NOTAND instruction. By utilizing the *Specific-Block* instruction, we can implement a single hardware logic iteration for state updates using ROTORBLOCK. This instruction expands the ROTXOR by extracting words from specific positions in the state for multiple words in parallel. The instruction can be executed 640 or 1024 times, depending on the algorithm-level call, and therefore bring significant speedup. By using the *Specific-Procedure* instruction, we have designed a larger

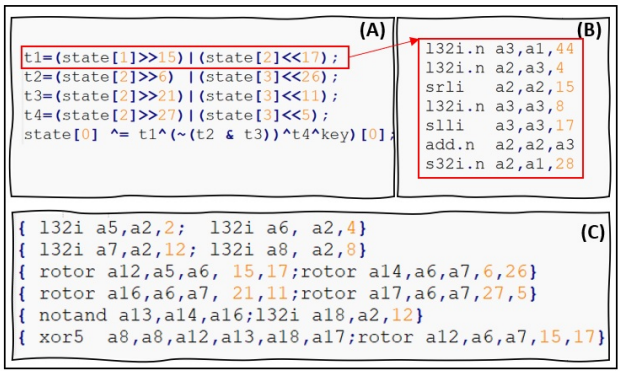

Figure: 20. *(A) TinyJAMBU state function C-code. (B) Native assembly code. (C) multi-issue code together with Generic-Atomic*

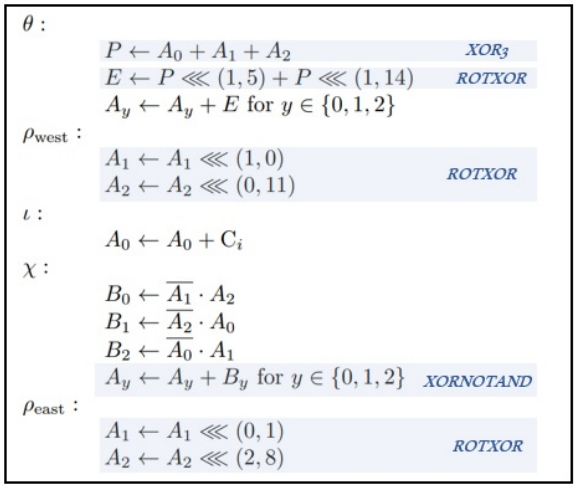

Figure: 21. Xoodoo Permutation phases along with *Generic-Atomic*

logic block that operates over multiple clock cycles. This is achieved by implementing the STATE_UPDATE function in a single computing block where the 128-bit state is used as both input and output, thus avoiding any register switching. In the code of the "state updates" function shown in Figure 20(A), one phase of the bit state extraction and update process is illustrated. This block of code repeats itself four times during a single iteration, with each phase having a different 32-bit state taken from the 128-bit input states. In Part (B) of the same figure, we can find the assembly translation for a single rotate operation made in C code. Each rotation takes about 7 instructions including loading and storing the variable to data memory. With the use of extensible processors, we can extend the instruction issuing to be able to multiply at the same cycle. By using *Generic-Atomic* set, we can execute two rotations in a single cycle. Additionally, loading data from data memory is done in parallel, further optimizing the process.

4.11 Xoodyak

Xoodyak is a versatile cryptographic object used for most symmetric-key functions such as hashing, pseudo-random bit generation, authentication, encryption, and authenticated encryption. IXoodyak uses the Xoodoo permutation The design approach of this 384-bit permutation is inspired by Keccak-p [DHVAVK18]. The Xoodoo permutation's state is represented by a matrix consisting of three rows and four columns, with each column

containing 32-bit words. This matrix can be visualized as three horizontal planes, each consisting of 128 bits, stacked on top of one another. The state can also be seen as 128 columns of three bits belonging to a different plane. Xoodoo performs 12 iterations of a round function, which consists of five steps: θ A column-parity mixing layer. χ a non-linear layer. ρ-west and ρ-east are two plane-shifting layers; and a round-constant addition. The ρ layers move bits horizontally and perform lane-wise rotations of planes and lanes by 11, 1, and 8 bits to the left. The parity-computation part of θ and the χ layers only interact with state-bits vertically, within 3-bit columns. The θ layer mainly executes XORs and left rotations by 5 and 14 bits. Finally, the non-linear layer χ applies a 3-bit S-box to each state column, which can be computed using logical ANDs, XORs, and bitwise complements. **CrISA-X description:** The extraction process simplifies obtaining a single 32-bit word by combining two adjacent 32-bit state words using *Generic-Atomic* Figure 21 describes a Pseudo-code description of the Xoodoo. We utilized the XOR3 instruction from the CrISA-X's *Generic-Atomic* set. This instruction allows us to perform the XOR operation between three operands in a single cycle, producing an output result. We also incorporat a few fusion instructions, such as ROTXOR which combines the rotate-left and XOR operations, and XORANDNOT which combines XOR with the result of NOT-AND operations. The Xoodoo permutation state is a cryptographic primitive composed of 384-bit permutations, which can be subdivided into four groups of 96 bits. The round count parameterize each group of bits. These bits are input for wide instructions, such as the *Specific-Block* and *Specific-Procedure* CrISA-X sets. The *Specific-Block* set executes each step (θ, χ, ρ-west, and ρ-east) as a single instruction in a single cycle. On the other hand, the *Specific-Procedure* set implements all steps (θ, χ, ρ-west, and ρ-east) as one computational block, the XOODOOPERMUT12 rounds, as a single instruction.

To optimize the performance of both sets, we extended the processor with a custom state that saves the round constants labeled as RC[i]. Consequently, this design choice reduces the required registers and minimizes the instruction encoding space. Moreover, along with features like multiple issues and doubling load units, this modification results in a significant boost in speed.

5 Implementation

This section details the implementation aspects of the software and processor platform used for evaluating CrISA-X instructions.

5.0.1 Software

Our design implements custom instructions for a large set of LWC from NIST and CAESAR finalist [TMC+21] to optimize latency performance. As a starting point for our optimization, we used publicly available versions of lightweight cryptography 32-bit optimized C-codes. This is the source code submitted for a given algorithm [nis22b]. Table 3 summarizes the kernel implementations and code flavor used in the LWC finalists algorithms. The NIST-specified benchmark API facilitates encryption and decryption. It includes an AEAD API for encrypting and decrypting using aead_encrypt and aead_decrypt methods. The code has been modified to support the original and the replacement kernel implementations. The original base implementation is used without alteration. We optimized the reference code with CrISA-X instruction sets. We have developed a high-level software strategy to utilize each realized CrISA-X design by the host core. We begin with a baseline implementation for each algorithm and use it as is, without modifications. The baseline implementation refers to the source code submitted for a particular algorithm. We aim for consistency by utilizing the most efficient C-code 32-bit processor implementation strategy for the base implementation, which is compatible with our replacement kernel.

Minor changes are made to create a kernel implementation that can be selected between the base and replacement code developed by us by utilizing appropriate C pre-processor directives. Maintaining code correctness is critical. We perform functional and inline checks to ensure algorithms function similarly for base and replacement code. To achieve the lowest latency performance for both the base and replacement code, we always use CLANG LLVM 15 as our software compiler. We set the highest level of speed optimization to -O3. The default optimization option for our CLANG compiler is optimized for speed rather than size. The Clang LLVM Compiler is an open-source compiler designed to provide the best possible implementation of the C family of programming languages. It uses the LLVM optimizer and code generator, which enables it to offer top-notch optimization and code generation support for many targets. Additionally, the compiler facilitates creating and encoding new instruction extensions through the backend compiler update. Cryptographic implementations are often optimized by writing them in assembly language to prevent bugs or weaknesses introduced by the compiler. However, in our case, we chose to use reference-optimized code written in C. We trust the compiler will match this code with the most optimized assembly code available. To further optimize the code at a higher level, we use the CLANG compiler with higher optimization. This generates native 32-bit codes by utilizing advanced compiler features such as optimized register reordering and allocation, loop unrolling, and utilizing processor base and extended ISA.

5.0.2 Hardware

An essential aspect of the optimization was additional custom hardware resources added to the processor architecture and used to speed up the encryption and decryption procedure as illustrated in Figure 5 and described in Sub-section 4.1. Our target platform (i.e., Host core) is based on an extended Xtensa 32-bit RISC Tiny configuration[4]. The processor core utilizes a 5-stage, in-order pipeline to execute instructions. To simplify things, we are using 16KB tightly coupled memories for both instructions and data, located very close to the CPU, which allows for single-cycle access. To demonstrate the benefits of adding CrISA-X instruction set, we did not use optimizations like the branch prediction mechanism to optimize the execution of branch instructions. The processor is designed in a modular fashion, which allows for precise implementation of instructions extension needed by the domain and extends the processor hardware with minimal area or power overheads. The processor can be tailored to specific application requirements by configuring it with various feature parameters. These options include multipliers, a floating-point unit, a multi-issue with flexible VLIW issuing stage, a wider data and instruction bus, and extended load and store units capable of supporting multiple memory accesses. Instructions can be extended using the TIE language to enhance processor performance. The TIE language enables the creation of a new data path, complete with new registers, register files, multi-cycle execution units, SIMD execution units, a VLIW data path, and custom data types and processor registers, as discussed in the above sections. The TIE Compiler facilitates this process by updating the entire compiler toolchain, including the compiler, debugger, and profiler, as well as the instruction-set simulator and system models. Furthermore, the TIE Compiler automatically inserts optimized clock-gated execution units, registers, register files, control logic, bypass logic, and other components into the processor hardware, ensuring that the process is correct by construction, as verified by EDA. The CrISA-X instruction set has been designed to efficiently execute, either as a single-cycle or two-cycle, depending on the operating frequency target, the number of operands, and the computation logic involved. For those reasons, for instructions set from the *CrISA-X Specific-Procedure or Specific-Block* classification, a register pipeline was added for some of the data paths. Our processor has been equipped with efficient forwarding units, enabling

[4]Under the tensilica academic program application agreement

additional function units to be located in the execute stage and active in parallel. The processor has also been extended to include a register file decoder and an instruction decoder, which allow it to correctly provide inputs and output operands to the functional units, ensuring the required computations are performed efficiently. To evaluate the area and the cycle-accurate execution latency, we have developed an experimental platform using the AMD Kintex7 FPGA KC705 evaluation platform. We have designed a stand-alone synthesizing using Xilinx Vivado 2019.1, which includes the processor and CrISA-X instruction set. We have used default synthesis settings and have not invested effort in synthesis or post-implementation optimization. The host core itself uses a 200MHz clock.

5.0.3 Evaluation Metrics

The study aims to improve the speed and efficiency of CrISA-X instruction sets and NIST algorithms, focusing on trade-offs between them. The study was conducted on each NIST finalist algorithm, using several quantitative measures such as clock cycle latency and code size (footprint) to evaluate the LWC algorithms. Data was collected for various NIST LWC algorithms. We have implemented specific scoping measures to manage the extensive design space effectively and minimize the required engineering effort, including:

- We studied the cycle count results for AEAD functions whose code is not inline. We done that by introduced a compiler attribute called $\_\_\_attribute\_\_\_((noinline))$. This attribute prevents the compiler from optimizing code size at the expense of execution speed by disabling the in-lining of the function. In-lining is a technique the compiler or interpreter uses to eliminate the overhead associated with function calls by inserting the function code directly at the call site. This can enhance performance by reducing the number of instructions executed and eliminating the need for stack operations and jumps.

- When using the unrolling compiler together with the -O3 optimization compilers try scheduling a loop to get the best performance. By using a CrISA-X, the compiler can almost always reduce the number of instructions required to implement a replacement kernel. This means that the loop overhead, which results from iteration over rounds within it, can become more prominent.

- Constant-time replacement kernel implementations are utilized due owing to CrISA-x, especially for the CrISA-x *Specific-Block* and *Specific-Procedure* categories. This property is easier to deliver compared to other implementation strategies.

- Some algorithms, such as GIFT-OCB and Ascon, use big-endian representation for the byte-order input and output. However, Xtensa and most other embedded microcontrollers use little-endian representation to process and store 32-bit data. Therefore, the byte order of 32-bit words injected into or extracted from the state must be reversed. Our implementation performs the injection and extraction of words using special CrISA-X 1-cycle instruction, including sequence of bytes conversion, in a byte-by-byte fashion. This has the advantage of not requiring attention to the alignment of the byte arrays in which the inputs and outputs are stored.

- Same as above when the optimized code uses a bit of interleaving approach like in PHOTON-Beetle and ASCON. Bit interleaving 32-bit to 64-bit code is a technique for converting a 32-bit integer into a 64-bit integer by interleaving the bits of the 32-bit integer with zeros. In ASCON code, permutation is surrounded by two functions to_bit_interleaving() and from_bit_interleaving() , those too were implemented with CrISA-X instruction and execute in a single cycle approach.

5.0.4 Suggested Instruction Sets and their Verification.

The final step in CrISA-X instruction set design is to verify and test the correctness and quality of the ISA specification and implementation. Verification checks whether the ISA meets the design requirements and specifications, while testing checks whether it works as expected in real scenarios. The methods we used for this purpose include:

1. Unit-level Verification: In this phase, a unit-level operation verification is performed in isolation to ensure that each new instruction operation is correctly executed as per the defined specifications. The CrISA-X specifications are established based on the LWC algorithm flow. The unit-level procedure entails verifying that the user-defined instruction functionality and data types are implemented accurately and in accordance with the specifications. An instruction set simulation is employed in a cycle-accurate mode to accomplish this. The unit-level verification process is a crucial step in ensuring the integrity of the single computing code's functionality and the accuracy of its output. We could detect logical bugs related to the ISE HDL implementation.

2. Algorithm Simulation: In the second phase, we will verify if the CrISA-X ISA is correctly implemented at the application level. Using a software simulation environment, the reference kernels and algorithms, which are pre-existing implementations of specific LWC components, are used as test cases. These reference kernels and algorithms are executed using the CrISA-X ISA implementation, and the results are compared against expected outcomes. We used a large set of plaintext and ciphertext test patterns to observe and feed the entire encryption and decryption process. The step process ensures the application's functionality and the accuracy of its output. With this verification level, we could identify logical bugs related to the ISE functionality in the algorithm context that can arise from algorithm flow change or compiler instruction reordering.

3. Hardware Verification: In the final verification phase, we verify the CrISA-X hardware executing on processor RTL in the RTL hardware simulation environment. To make this verification real or near-real, we use extended processor RTL hardware and CrISA-X HDL integration. This phase verifies the interaction between CrISA-X instruction hardware and other processor and peripherals hardware components such as local memories, additional register and constant table, and memory buses.

4. The post-design phase involves testing implemented LWC algorithms on real FPGA hardware. This stage allows us to test the correct functionality of the CrISA-X solution in real-world conditions, ensuring that it meets the desired specifications and requirements. Additionally, real hardware testing provides valuable insights into potential issues or bottlenecks that may not have been apparent during the software simulation, enabling further optimization and refinement of the design.

6 Evaluation

6.1 Conditions

In this section, we present the result of evaluating our CrISA-X instruction designs from both hardware and software perspectives. As mentioned in Section 5. We attempt to align the replacement code as closely as possible with the original LWC code by updating the code with CrISA-X instruction. Table 3 describes a list of algorithms, Code benchmark names which flavor was used, and kernels. Each algorithm was tested with two packet sizes: 128 and 16 bytes. The results described in 6.1.2 refer to 16-byte packet sizes. The

Table 3: Algorithms List and Code name
A list of algorithms, associated code benchmark names, and the kernel code used in this evaluation.

Algorithm	Code Benchmark	Flavor	Kernel
Ascon	ascon128v12/bi32_bit_intlv	bitinterleaving 32bit	P6\P8\P12
Elephant	elephant200v2/ref	ref implementation	KeccakP200Round
GIFT	giftcofb128v1/opt32	bitslicing 32bit	giftb128
Grain	grain128aeadv2/rhys	rhys 32bit	grain128_next_keystream
PHOTON	photonbeetleaead128rate128v1/bi32	bitslicing 32bit	PHOTON_Permutation
Romulus(FS)	romulusn/opt32	bitslicing 32bit	skinny128384plus,permutetk
Sparkle	schwaemm256128v2/opt	optmize 32bit	Sparkle_opt
TinyJAMBU	tinyjambu128v2/opt	optmize 32bit	state_update
Xoodyak	xoodyakround3/ref	ref implementation	XoodooPermute12rounds

Table 4: CrISA-X Hardware
CrISA-X instruction cost in LUTs

Submission	**Processor** Area plus **CrISA-X-GA** [LUT]	**Processor** Area plus **CrISA-X-SB** [LUT]	**Processor** Area plus **CrISA-X-SP** [LUT]
Ascon	4004 (+12.6%)	5539 (+36.81%)	6614 (+47.08%)
Elephant		5259 (+33.45%)	6388 (+45.21%)
GIFT		5195 (+32.63%)	6098 (+42.6%)
Grain		5001 (+30.01%)	6066 (+42.3%)
PHOTON		5420 (+35.42%)	6259 (+44.08%)
Romulus		5453 (+35.82%)	6388 (+45.21%)
Sparkle		5033 (+30.46%)	6517 (+46.29%)
TinyJAMBU		5517 (+36.56%)	6421 (+45.49%)
Xoodyak		5291 (+33.85%)	6292 (+44.37%)

evaluation of software usage for each CrISA-X design has resulted in measuring latency in clock cycles for each algorithm. Compared to the baseline, the overhead is listed in parentheses. The performance evaluations were conducted on both an Instruction Set Simulator (ISS) toolchain and a Xilinx evaluation board. As the results of the two targets were identical, only one was mentioned.

6.1.1 Hardware- Area

Table 4 summarizes the area FPGA synthesis results for each CrISA-X instruction set design per category. For the CrISA-X *Genric-Atomic* set, we offer a single set of instructions. For the *Specific-Block* and *Specific-Procedure* sets, we suggest a set of instructions per LWC algorithm. Therefore, the area result is reported for each category appropriately. We use FPGA Look-Up Tables (LUTs) as indicators for design areas for the processor including instruction extensions. LUTs are the fundamental building blocks of logic and are used to implement most logic functions in the design. The overhead is cumulative and relative to the base five-stage processor, which consists of 3500 LUTs. According to our area synthesis results, implementing the Crisa-X *Genric-Atomic* extensions requires an additional 504 LUTs, resulting in a cumulative overhead of 12.6% compared to the baseline processor. As expected, both CrISA-X *Specific-Block* and *Specific-Procedure* have an impact on increasing the area—the former ranges from 1,501 to 2,017 LUTs and the latter from 2,566 to 3,017 LUTs. For clarity, the measurement excludes VLIW processor extension hardware for multi-issuing, which requires about 910 additional LUTs. The same is true for memories.

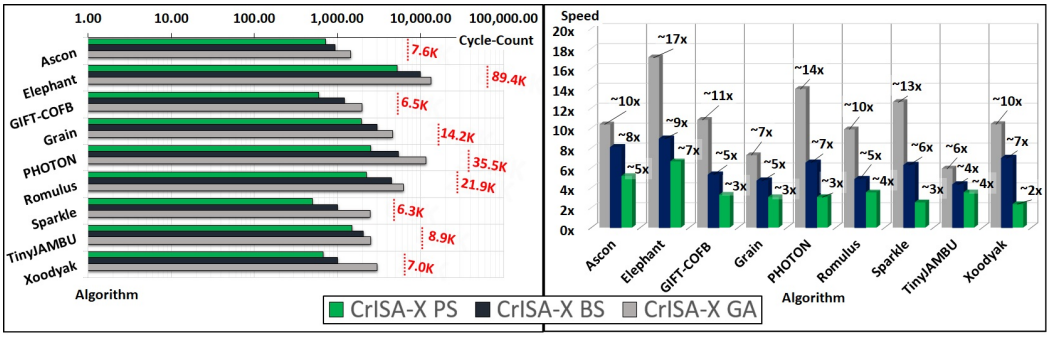

Figure: 22. Performance Analysis of Finalist AEAD Algorithms. (A) CrISA-X comparative Analysis of the Latency performance of original and replacement Code Implementations (B) CrISA-X quantifying the potential speedup in latency by a factor of X. *Conditions:16B plaintext/data associated.AEAD API used.O3-optimized code. cycle-level latency measured.*

6.1.2 Software- Latency and Code Space

Table 5 summarizes the latency in reference to CrISA-X instruction categories GA, BS, and PS. This latency result is taken for running kernel code in the context of the aead_encrypt and aead_decrypt API. We provide information regarding the execution latency of each algorithm code, reported in cycle count. The latency for encryption and decryption is specified separately. In addition to each measurement, we specify the corresponding boost factor compared to the baseline, a processor with no extensions. The results are described separately for various CrISA-X instructions categories. The measurements were taken for 16B plaintext and associated data. The code was highly optimized with -O3 optimization level for baseline and replacement codes. Figure 22 illustrates the latency results in a bar chart to emphasize the performance boost brought by CrISA-x, compared to the baseline. Note that the left (A) Sub-figure of Figure 22 is in logarithmic x-axis whereas for clarity the right Sub-figure (B) is in linear scale (y-axis).

Table 6 summarizes the Latency and Code Space (Text-Size) results. This time the data focuses on the kernel code in isolation only. We report the number of Crisa-X *Genric-Atomic* instructions used, execution latency results, and memory footprint for each kernel code. Execution latency is the time it takes to execute a program, measured in clock cycles. On the other hand, memory footprint refers to the amount of instruction space (text) measured in bytes. We calculate the relative improvement factor by comparing the results with an associated baseline. The baseline is captured on the reduced processor using only the base ISA, not processor extension, and no instruction extensions. The result is illustrated in Figure 23. Kernel speedup has been achieved by a 2x to 3.9x boost in latency performance while reducing the code space by 20% to 250%. Note that for some kernels, e.g., Ascon, GIFT, and Romulus, we utilize additional CrISA-X instructions to speed up pre-computation functions like generating big-endian representation, or byte swapping.

6.1.3 Applicability to various LWCs.

The CrISA-X extension provides instructions from various categories with diverse levels of computing and applicability opportunities. The generic-atomic class demonstrates promising applicability to various LWC schemes. The applicability of CrISA-X to LWC schemes stems from its ability to accelerate common cryptographic sub-operations while maintaining low hardware overhead. By introducing custom instructions for these cryptographic primitives, CrISA-X GA enables processors to execute these operations more efficiently, thereby improving the overall performance of LWC schemes. Furthermore, the

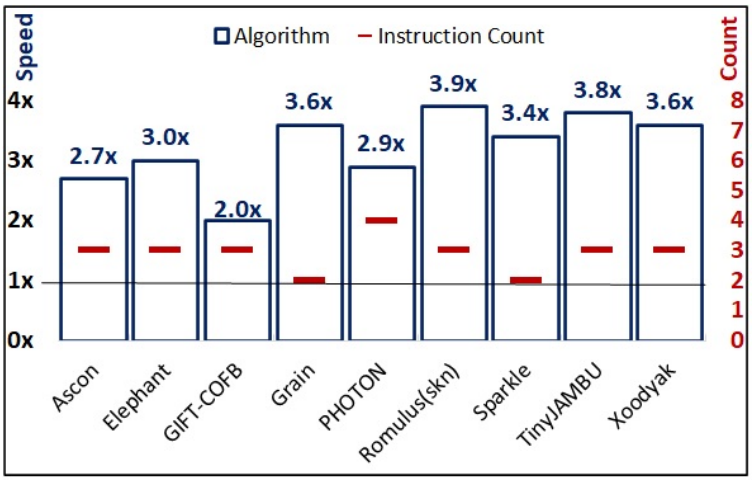

Figure: 23. Kernel Only: CrISA-X GA Speedup and instruction count, compared to baseline

Table 5: CrISA-X Latency - aead_encrypt\decrypt API
CrISA-X GA, BS, and PS performance latency results for finalist LWC algorithms compared to baseline reduce processor. "Proc." is a short for processor."Baseline." is a short for Baseline Code without ISE and "GA,BS and PS." is is refer to Crisa-X
Conditions: 16B plaintext/AD, AEAD API used, optimized -O3. Cycle-level latency measured.

Submission	Functionality	Reduce Proc. +Baseline	Extend Proc. +Baseline	Extend Proc. +GA	Extend Proc. +SB	Extend Proc. +SP
Ascon	aead_encrypt	7557(1x)	4376(1.73x)	1460(5.18x)	936(8.07x)	729(10.37x)
Ascon	aead_decrypt	7581(1x)	4532(1.67x)	1480(5.12x)	950(7.98x)	750(10.11x)
Elephant	aead_encrypt	89673(1x)	61586(1.46x)	13452(6.67x)	10014(8.95x)	5424(16.53x)
Elephant	aead_decrypt	89432(1x)	61752(1.45x)	13484(6.63x)	10021(8.92x)	5229(17.1x)
GIFT	aead_encrypt	6493(1x)	3049(2.13x)	1972(3.29x)	1221(5.32x)	601(10.8x)
GIFT	aead_decrypt	6637(1x)	2933(2.26x)	1969(3.37x)	1222(5.43x)	600(11.06x)
Grain	aead_encrypt	14189(1x)	6096(2.33x)	4648(3.05x)	3069(4.62x)	1950(7.28x)
Grain	aead_decrypt	14523(1x)	6409(2.27x)	4884(2.97x)	3009(4.83x)	1955(7.43x)
PHOTON	aead_encrypt	36127(1x)	17832(2.03x)	11621(3.11x)	5513(6.55x)	2647(13.65x)
PHOTON	aead_decrypt	35470(1x)	17199(2.06x)	11600(3.06x)	5443(6.52x)	2546(13.93x)
Romulus	aead_encrypt	21964(1x)	11325(1.94x)	6200(3.54x)	4502(4.88x)	2221(9.89x)
Romulus	aead_decrypt	21923(1x)	11374(1.93x)	6195(3.54x)	4530(4.84x)	2243(9.77x)
Sparkle	aead_encrypt	6332(1x)	3743(1.69x)	2510(2.53x)	1004(6.31x)	501(12.64x)
Sparkle	aead_decrypt	6310(1x)	3778(1.67x)	2500(2.52x)	1010(6.25x)	500(12.62x)
TinyJAMBU	aead_encrypt	8889(1x)	3888(2.29x)	2515(3.53x)	2051(4.33x)	1521(5.84x)
TinyJAMBU	aead_decrypt	8953(1x)	3843(2.33x)	2574(3.48x)	2096(4.27x)	1499(5.97x)
Xoodyak	aead_encrypt	7122(1x)	4191(1.7x)	2998(2.38x)	1005(7.09x)	676(10.54x)
Xoodyak	aead_decrypt	7031(1x)	4317(1.63x)	3001(2.34x)	1045(6.73x)	754(9.32x)

[r]With reference to [CGM+23] taking the lowest result for RV32GC +Zbkb/x between:$V_{1,2,3}$.

GA extension offers flexibility regarding its support for various LWC algorithms. Table 8 summarizes the CrISA-X GA, SB and SP instruction set per each algorithm. The table shows that the CrISA-X GA instruction set includes various instructions, such as XOR2 or any of their variants, used widely across eight LWC algorithms out of nine. The same trend is observed for ROTATE instruction classes like ROTXOR and ROTOR, which are

Table 6: CrISA-X Latency - Kernel in Isolation. Latency speedup and instruction size result of CrISA-X compared to the baseline (reduced processor).

Submmision	Code Kernel	Latency Text Size	Reduce Processor	Extended Processor CrISA-X GA	Instruction Count CrISA-X GA
Ascon	P6	latency [cycle]	428	160(2.7x)	3
	*same ratio for P8\P12	Text Size [bytes]	1010	412(2.5x)	
Elephant	KeccakP200Round	latency [cycle]	615	203(3x)	3
		Text Size [bytes]	139	59(2.4x)	
GIFT	giftb128	latency [cycle]	1400	705(2x)	3
		Text Size [bytes]	901	409(2.2x)	
Grain	grain128_next_keystream	latency [cycle]	310	85(3.6x)	2
		Text Size [bytes]	926	602(1.5x)	
PHOTON	PHOTON_Permutation	latency [cycle]	11762	4096(2.9x)	4
		Text Size [bytes]	2012	955(2.1x)	
Romulus	skinny128_384_plus	latency [cycle]	4358	1130(3.9x)	3
		Text Size [bytes]	1550	907(1.7x)	
Romulus	permute_tk	latency [cycle]	1615	533(3x)	3
		Text Size [bytes]	751	555(1.4x)	
Sparkle	Sparkle_opt	latency [cycle]	1958	577(3.4x)	2
		Text Size [bytes]	221	120(1.8x)	
TinyJAMBU	state_update	latency [cycle]	603	157(3.8x)	3
		Text Size [bytes]	265	201(1.3x)	
Xoodyak	XoodooPermute12rounds	latency [cycle]	899	252(3.6x)	3
		Text Size [bytes]	698	569(1.2x)	

widely used across seven LWC algorithms. Other GA instructions, such as XORNOTAND and ROTXOR are also used in multiple algorithms. The CrISA-X GA set exhibits wide applicability across numerous LWC schemes. In contrast, while possessing lesser applicability, the CrISA-X SB instruction class still finds common usage. Algorithms such as Elephant and Xoodyak, based on KECACCK, can share instructions for implementing the THETA, RHO, and CHI compute blocks. Another example is when the LWC algorithm processes a large data set from memory like 128bit. In such cases, LOAD and STORE 128bit are instructions that can be shared. The CrISA-X SP is built for a specific algorithm permutation and can only be shared in cases where the same permutation is used across LWC algorithms.

6.1.4 Comparable Studies

Several studies of software performance evaluation across different processor architectures exist. NIST LWC team presents the benchmarking framework and shows the evaluation results [RPM22] and in git repository[5]. The evaluation was made on several processors from different architectures and ISA sets, including some with ISA extenstion. The leading architectures evaluated are AVR ATmega, Amtel, ARM Cortex-M0\3\4, RISC-V, L106. The following list outlines the characteristics of various processor architectures:

- RISC-V 32-bit Architecture plus RV32GC with subset Z_bkb, Z_bkx - a subset of K for bit manipulation instructions sign as Zbkb/x $V32_{,0\ ,1\ ,2}$. Base ISA extended with LWC-specific ISEs.

[5]https://github.com/usnistgov/Lightweight-Cryptography-Benchmarking

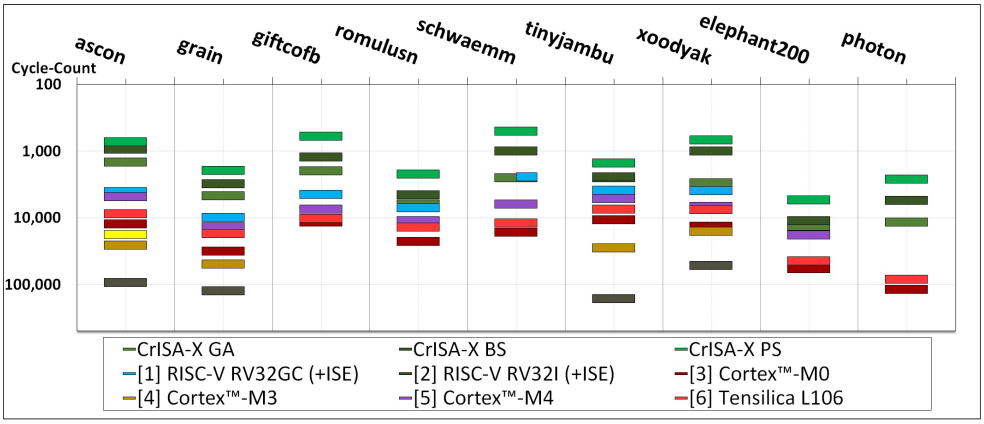

Figure: 24. Performance Comparison of Finalist LWC Algorithms Using CrISA-X Classification: GA, BS, and PS vs. Other Processors with and without ISA Extension *Conditions*: 16B plaintext/data associated. AEAD API used. O3-optimized code. cycle-level latency measured.

- RISC-V 32-bit Architecture plus RV32I base ISA extended with LWC-specific ISEs for *Ascon* only.
- Cortex-M0™ 32-bit Architecture with Armv6-M Instruction set on evaluation board Arduino MKR Zero (SAMD21G18A)
- Cortex-M3™ 32-bit Architecture with ARMv7-M Instruction set
- Cortex™-M4 32-bit Architecture with ARMv7-M plus DSP instruction set extension. on evaluation board Arduino Nano 33 BLE (nRF52840)
- Atmel AVR Atmega 8-bit Architecture
- Tensilica L106 32-bit Architecture on evaluation board NodeMCU v2 (ESP8266)

We compare our performance latency gain of CrISA-X vs other processors and ISA benchmark. Figure 24 showing Table 7 in chart bar view for easy tracking and illustrates latency performance using CrISA-X compared to other solutions and architectures. To ensure a fair comparison, we evaluate the same kernel and code variant using equal amounts of plaintext and associated data. We conduct tests for CrISA-X and the benchmark data, with 16B plaintext/data associated. We use the AEAD API and O3-optimized code. If multiple benchmark data are available for the code variant, we select the result from the benchmark set that has the best performance mean and lower latency.
We look into performance comparison for the Keccak-f[200] permutation to enhance our study's data further. We compared the reported boost from two works, one on RISC-V and one on ARM processors, to what we achieved with CrISA-X. The first work is from Lemmen [LMD20], presenting an optimized RISC-V acceleration implementation of Elephant Keccak-f[200], based on using a RISC-V, supporting RV32IMAC core. The authors used SiFive E31 core and modified the state representation and function to enable the processing of multiple blocks concurrently. Rawat [RS17] presents an optimized ARM NEON using SIMD instruction for acceleration. Figure 25 illustrates the performance comparison of CrISA-X against Lemmen and NEON SIMD from ARMv7. We compare the reported performance of those works against CrISA-X's performance in different categories.

Table 7: CrISA-X vs Other Latency.LWC Algorithm Performance Comparison using CrISA-X Classification vs other architecture and ISA.

Sub	Xtensa GA	Xtensa SB	Xtensa SP	[1]RISC-V RV32GC (+ISE)	[2]RISC-V RV32I (+ISE)	[3]Cortex M0	[4]Cortex M3	[5]*Cortex M4*	[6]Ten L106	[7]Atmel
ascon	1460	936	729	4059	17931	12337	25737	4842	8688	93452
grain	4648	3069	1950	9962	N/A	31783	49701	13664	17312	125334
gift	1972	1221	601	4440	N/A	11603	N/A	7487	10137	N/A
romulusn	6200	4502	2221	7104	N/A	22503	N/A	11145	13764	N/A
schwaemm	2500	1004	501	2424	N/A	16475	N/A	6281	11892	N/A
tinyjambu	2515	2051	1521	3891	N/A	10758	28441	5134	7395	163276
xoodyak	2998	1005	676	3921	N/A	13418	16070	6708	7546	51639
elephant	13452	10014	5424	N/A	N/A	58287	N/A	18219	44559	N/A
photon	11621	5513	2647	N/A	N/A	118408	N/A	N/A	84011	N/A

References:[1]: [CGM+23] ,[2]: [AÖ21] ,[3]: [oST22]
,[4]: [WYY22] [5]: [oST22] ,[6]: [oST22] ,[7]: [WYY22]
N/A: No data or No Applicable comparison

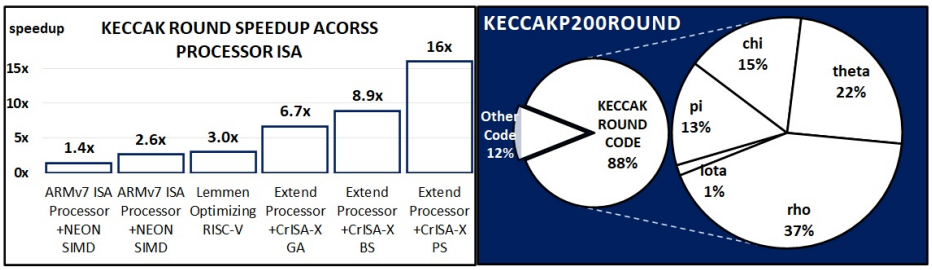

Figure: 25. Performance latency Keccak-P200 Round

7 Summary

In this study, we propose **CrISA-X methodology** as a potential enhancing ISA and processor extension for LWC algorithms. The methodology, design flow, suggested instructions and processor extensions are generic and applicable to any RISC processor and can be used as a layout for new RISC-V extensions. For example, with CrISA-X enabled, the *Ascon* algorithm family, which was announced as a finalist by NIST, can achieve a speedup of 5x to 10x depending on the selected instruction set. Other LWC algorithms achieve average speedups ranging 2x to 17x factors using the same instruction set. Figure 24 illustrates these superior performance on real hardware. Table 8 in the Appendix summarizes the instruction set for each category, while Table 9 lists the applicable processor extensions for each algorithm. Broadly speaking, the comparison of CrISA-X with software-only alternatives or other ISA extensions or processor architectures shows:

1. The CrISA-X instructions set allows a reduction in execution latency.
   (a) *Genric-Atomic* instructions: FUSED a few base instructions to one, and design as a single cycle - brings 2x-5x speedup.
   (b) *Specific-Block* instructions: SIMD a few in/out operands into mid-range computation block and design as two-cycle- brings 4x-8x speedup.

(c) *Specific-Block* instructions: SIMD a few-plus in/out operands into heavy computation block, and design as two-cycle- brings 7x-17x speedup.

2. The degree of reduction is instruction, computing and algorithm-dependent but significant in some cases and on average, and, at the same time.
3. Our results show that software-only implementations using CrISA-X can be significantly more efficient than those using the base ISA alone.
4. The hardware overhead of the CrISA-X instruction set ranges from low to mid, depending on the CrISA-X set being used.
5. The CrISA-X instruction set enables constant-time execution, reducing memory footprint. These features highlight the value of instruction set extensions in resource-constrained devices and the Lightweight Cryptography (LWC) process.

The CrISA-X provides multiple levels of computational instruction and multiple dimensions for processor extension, resulting in new instruction sets that demonstrate excellent performance. By carefully analyzing the constituent algorithms and building a set of processor extension opportunities, the CrISA-X set supports multiple algorithms and optimizes Pareto. We establish a multi-issuing instruction slot format that balances the performance of various algorithms. That is, we provide instruction extensions set for each algorithm, designing their complexity as minimally as possible and with the goal of keeping these instructions general enough (in format, number of slots and operands etc.) to be able to share them between algorithms without significantly increasing cost and area. CrISA-X and its associated LWC application codes are provided as examples and are extensively available in our public GitHub[6] repository. This repository is designed as a reference for embedding optimized ISAs and C-code algorithms and is packed with supporting examples and supplementary content for evaluation, testing, and profiling purposes. As discussed in Section 4, it can be easily ported.

Acknowledgements

We would like to express our gratitude to the anonymous reviewers for providing their valuable and constructive feedback. This work was carried out in collaboration with EnICS at the Faculty of Engineering, Bar-Ilan University, Israel, in The Secure Electronic Systems Lab (Selecsys, Lab 248), under the supervision of Dr. Levi.

Itamar Levi and Oren Ganon received partial funding for this project from the Israel Innovation Authority (IIA), Bio-Chip Consortium Grant file No. 75696, Israel, and Israel Science Foundation (ISF) grant 2569/21.

References

[ABCdS+22] Malik Alsahli, Alex Borgognoni, Luan Cardoso dos Santos, Hao Cheng, Christian Franck, and Johann Großschädl. Lightweight permutation-based cryptography for the ultra-low-power internet of things. In *International Conference on Information Technology and Communications Security*, pages 17–36. Springer, 2022.

[AÖ21] Özlem Altınay and Berna Örs. Instruction extension of rv32i and gcc back end for ascon lightweight cryptography algorithm. In *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*, pages 1–6. IEEE, 2021.

[6] https://github.com/SYSlab1/CRISA-X

[AP20] Alexandre Adomnicai and Thomas Peyrin. Fixslicing aes-like ciphers: New bitsliced aes speed records on arm-cortex m and risc-v. *Cryptology ePrint Archive*, 2020.

[BBdS+19] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Qingju Wang, and Alex Biryukov. Schwaemm and esch: lightweight authenticated encryption and hashing using the sparkle permutation family. *NIST round*, 2, 2019.

[BCD+19] Zhenzhen Bao, Avik Chakraborti, Nilanjan Datta, Jian Guo, Mridul Nandi, Thomas Peyrin, and Kan Yasuda. Photon-beetle authenticated encryption and hash family. *NIST Lightweight Compet. Round*, 1:115, 2019.

[BCI+19] 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 the NIST Lightweight Cryptography project*, 2019.

[BDP+14] G Bertoni, J Daemen, M Peeters, GV Assche, and RV Keer. Caesar submission: Ketje v2. online at http://ketje. noekeon. org/ketje-1.1. pdf, 2014.

[BDPA15] G Bertoni, J Daemen, M Peeters, and GV Assche. Caesar submission: Keyak v2. online at http://keyak. noekeon. org/keyak-2.1. pdf, 2015.

[BDPVA09] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak specifications. *Submission to nist (round 2)*, 3(30):320–337, 2009.

[BPP+17] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift: A small present: Towards reaching the limit of lightweight encryption. In *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 321–345. Springer, 2017.

[CGM+23] Hao Cheng, Johann Großschädl, Ben Marshall, Dan Page, and Thinh Pham. Risc-v instruction set extensions for lightweight symmetric cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 193–237, 2023.

[CHKP98] Hoon Choi, Seung Ho Hwang, Chong-Min Kyung, and In-Cheol Park. Synthesis of application specific instructions for embedded dsp software. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 665–671, 1998.

[CJL+20] Fabio Campos, Lars Jellema, Mauk Lemmen, Lars Müller, Daan Sprenkels, and Benoit Viguier. Assembly or optimized c for lightweight cryptography on risc-v? In *Cryptology and Network Security: 19th International Conference, CANS 2020, Vienna, Austria, December 14–16, 2020, Proceedings 19*, pages 526–545. Springer, 2020.

[CMS07] Xiaoyong Chen, Douglas L Maskell, and Yang Sun. Fast identification of custom instructions for extensible processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):359–368, 2007.

[CZM03] Nathan Clark, Hongtao Zhong, and Scott Mahlke. Processor acceleration through automated instruction set customization. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 129–140. IEEE, 2003.

[DEM+19] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. Submission to nist. *Submission to the NIST LWC Competition*, 2019.

[DFA+20] Viet B Dang, Farnoud Farahmand, Michal Andrzejczak, Kamyar Mohajerani, Duc T Nguyen, and Kris Gaj. Implementation and benchmarking of round 2 candidates in the nist post-quantum cryptography standardization process using hardware and software/hardware co-design approaches. *Cryptology ePrint Archive: Report 2020/795*, 2020.

[DHVAVK18] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. The design of xoodoo and xoofff. *IACR Transactions on Symmetric Cryptology*, pages 1–38, 2018.

[DPU+16] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. Design strategies for arx with provable bounds: Sparx and lax (full version). *Cryptology ePrint Archive*, 2016.

[Dwo15] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. 2015.

[DYSC07] ZiBin Dai, XueRong Yu, JinHai Su, and XingYuan Chen. Accelerated flexible processor architecture for crypto information. In *2007 2nd International Conference on Pervasive Computing and Applications*, pages 399–403. IEEE, 2007.

[EF96] Martyn Edwards and John Forrest. A practical hardware architecture to support software acceleration. *Microprocessors and Microsystems*, 20(3):167–174, 1996.

[FS88] Jay Fenlason and Richard Stallman. Gnu gprof. *GNU Binutils. Available online: http://www. gnu. org/software/binutils (accessed on 21 April 2018)*, 1988.

[GL23] Oren Ganon and Itamar Levi. Modular processor architecture with cryptography isa extensions. In *2023 21st IEEE Interregional NEWCAS Conference (NEWCAS)*, pages 1–2. IEEE, 2023.

[GPKT09] Vladimir Guzma, Teemu Pitkanen, Pertti Kellomaki, and Jarmo Takala. Reducing processor energy consumption by compiler optimization. In *2009 IEEE Workshop on Signal Processing Systems*, pages 063–068. IEEE, 2009.

[GS22] Shubham Gupta and Sandeep Saxena. Lightweight cryptographic techniques and protocols for iot. In *Internet of Things: Security and Privacy in Cyberspace*, pages 55–77. Springer, 2022.

[KR06] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 21–30, 2006.

[LMD20] Mauk Lemmen, Bart Mennink, and Joan Daemen. *Optimizing Elephant for RISC-V*. PhD thesis, BSc thesis, Radboud University, 2020. https://www.cs. ru. nl/bachelors . . . , 2020.

[MAA+20] Dustin Moody, Gorjan Alagic, Daniel C Apon, David A Cooper, Quynh H Dang, John M Kelsey, Yi-Kai Liu, Carl A Miller, Rene C Peralta, Ray A Perlner, et al. Status report on the second round of the nist post-quantum cryptography standardization process. 2020.

[MB20] Thomas Xuan Meng and William Buchanan. Lightweight cryptographic algorithms on resource-constrained devices. *Preprints*, 2020.

[MPP21] Ben Marshall, Daniel Page, and Thinh Hung Pham. A lightweight ise for chacha on risc-v. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 25–32. IEEE, 2021.

[MSA22] Hasindu Madushan, Iftekhar Salam, and Janaka Alawatugoda. A review of the nist lightweight cryptography finalists and their fault analyses. *Electronics*, 11(24):4199, 2022.

[NIS22a] NIST. Nist requerments, available online: https://csrc.nist.gov/csrc/media/projects/lightweight-cryptography/documents/final-lwc-submission-requirements-august2018.pdf (accessed on 21 april 2023), 2022.

[nis22b] nist.gov. Nist final, available online: https://csrc.nist.gov/projects/lightweight-cryptography/finalists (accessed on 21 april 2023), 2022.

[oST22] National Institute of Standards and Technology. Benchmarking of nist lwc finalists on microcontrollers, 2022.

[PMMB22] Biagio Peccerillo, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *Journal of Systems Architecture*, 129:102561, 2022.

[PS00] Thomas Pornin and Jacques Stern. Software-hardware trade-offs: Application to a5/1 cryptanalysis. In *CHES*, pages 318–327, 2000.

[RPM20] Sebastian Renner, Enrico Pozzobon, and Jürgen Mottok. Nist lwc software performance benchmarks on microcontrollers, 2020.

[RPM22] Sebastian Renner, Enrico Pozzobon, and Jürgen Mottok. The final round: Benchmarking nist lwc ciphers on microcontrollers. In *Attacks and Defenses for the Internet-of-Things: 5th International Workshop, ADIoT 2022, Copenhagen, Denmark, September 30, 2022, Revised Selected Papers*, pages 1–20. Springer, 2022.

[RS16] Hemendra K Rawat and Patrick Schaumont. Simd instruction set extensions for keccak with applications to sha-3, keyak and ketje. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–8. 2016.

[RS17] Hemendra Rawat and Patrick Schaumont. Vector instruction set extensions for efficient computation of keccak. *IEEE Transactions on Computers*, 66(10):1778–1789, 2017.

[Sab23] Nafih Sabagh. *Msit: Modified Lightweight Encryption Algorithm for Secure Internet of Things*. PhD thesis, San Diego State University, 2023.

[Sea01] David Seal. *ARM architecture reference manual*. Pearson Education, 2001.

[SOM] Yroslav Sovyn, Ivan Opirskyy, and Olha Mykhaylova. Finding a bit-sliced representation of 4× 4 s-boxes based on typical logic processor instructions.

[SP20] Stefan Steinegger and Robert Primas. A fast and compact risc-v accelerator for ascon and friends. In *International Conference on Smart Card Research and Advanced Applications*, pages 53–67. Springer, 2020.

[SR20] Gopinath Sittampalam and Nagulan Ratnarajah. Enhanced symmetric cryptography for iot using novel random secret key approach. In *2020 2nd International Conference on Advancements in Computing (ICAC)*, volume 1, pages 398–403. IEEE, 2020.

[SZII11] Sadagopan Srinivasan, Li Zhao, Ramesh Illikkal, and Ravishankar Iyer. Efficient interaction between os and architecture in heterogeneous platforms. *ACM SIGOPS Operating Systems Review*, 45(1):62–72, 2011.

[TGMD20] Etienne Tehrani, Tarik Graba, Abdelmalek Si Merabet, and Jean-Luc Danger. Risc-v extension for lightweight cryptography. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 222–228. IEEE, 2020.

[TMC+21] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, Cagdas Calik, Lawrence Bassham, Jinkeon Kang, John Kelsey, et al. Status report on the second round of the nist lightweight cryptography standardization process. *National Institute of Standards and Technology Internal Report*, 8369(10.6028), 2021.

[TMC+23] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, Jinkeon Kang, Noah Waller, John M Kelsey, Lawrence E Bassham, and Deukjo Hong. Status report on the final round of the nist lightweight cryptography standardization process. 2023.

[VVP+16] Pawan Kumar Verma, Rajesh Verma, Arun Prakash, Ashish Agrawal, Kshirasagar Naik, Rajeev Tripathi, Maazen Alsabaan, Tarek Khalifa, Tamer Abdelkader, and Abdulhakim Abogharaf. Machine-to-machine (m2m) communications: A survey. *Journal of Network and Computer Applications*, 66:83–105, 2016.

[WC81] Mark N Wegman and J Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences*, 22(3):265–279, 1981.

[WYY22] Yuhei Watanabe, Hideki Yamamoto, and Hirotaka Yoshida. Performance evaluation of nist lwc finalists on avr atmega and arm cortex-m3 microcontrollers. *Cryptology ePrint Archive*, 2022.

[XJL+19] Yinhao Xiao, Yizhen Jia, Chunchi Liu, Xiuzhen Cheng, Jiguo Yu, and Weifeng Lv. Edge computing security: State of the art and challenges. *Proceedings of the IEEE*, 107(8):1608–1631, 2019.

[YM04] Pan Yu and Tulika Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 69–78, 2004.

A Additional Comparison and Data Tables

Table 8: CrISA-X Instruction Set
Summary of CrISA-X instruction set per algorithms.

Submission	Generic-Atomic	Specific-Block	Specific-Procedure
Ascon	XORNOTAND(1OUT,1IN\1IMD) XOR2(2OUT,2IN\2IMD) XORROT(1OUT,2IN\2IMD)	ASCON_LINR(5INOUT) ASCON_NONLINR(5INOUT)	ASCON_PERMUTATION(10INOUT)
Elephant	XORNOTAND(1OUT,1IN\1IMD) XOR#(1OUT,#IN) ROTXOR(1OUT,4IN)	THETA(4INOUT) RHO(4INOUT) CHI(4INOUT)	KECACCK_PERMUTATION(7INOUT)
GIFT	ROTOR(1OUT,4IN\4IMD) XOROR(1OUT,2IN) XORAND(1OUT\2OUT,2IN) XORROT(1OUT,2IN\2IMD) XOR2(2OUT,2IN\2IMD) XORSHFTAND(1OUT,3IN\3IMD)	SWAPMOVE(2INOUT,2INT) SBOX(4INOUT)	QUINTUPLE_ROUND(4INOUT)
Grain	ROTXOR(1OUT,4IN) XORAND(1OUT\2OUT,2IN) SHIFTLXOR(1OUT,2IN)	LOAD128B/STORE128B(4INOUT) BLOCKROTXOR(4INOUT)	GRAIN128_NEXT_KEYSTREAM(4INOUT)
PHOTON	XOR2(2IN\2OPRND OR 2OUT) XOROR(1OUT,2IN) SHIFTLXOR(1OUT,2IN) ROTOR(1OUT,2IN,2IMD)	LOAD128B/STORE128B(4INOUT) SHIFTROW(4INOUT) SBOX(8INOUT)	PHOTON_PERMUTATION(8INOUT)
RomulusN	XORSHFTAND(1OUT,3IN\3IMD) XOR2(2IN\2OPRND OR 2OUT) XOROR(1OUT,2IN)	SWAPMOVE(2INOUT,2INT) QUADRUPLEROUND(4INOUT) PERMUTE_TK_4\6\8\10(4INOUT)	SKINNY128_384_PLUS(6INOUT)
Sparkle	ROTOR(1OUT,2IN,2IMD) SHIFTLXOR(1OUT,2IN) XOR2(2IN\2OPRND OR 2OUT)	ARXLAYEL((4INOUT)) LINEARLAYER((4INOUT))	SPARKLE_OPT(8INOUT)
TinyJAMB'	ROTOR(1OUT,2IN,2IMD) XOR#(1OUT,#IN) NOTAND(1OUT,2IN)	ROTORBLOCK(5INOUT,4IMD)	STATE_UPDATE(8INOUT,8IMD)
Xoodyak	XOR#(1OUT,#IN) ROTOR(1OUT,2IN,2IMD) XORNOTAND(1OUT,1IN,1IMD) SHIFTLOR(1OUT,2IN)	THETA(3INOUT) RHO(3INOUT) CHI(3INOUT)	XOODOOPERMUT12(6INOUT)

Symbol '#' represents the number of operands involved as instruction operands.

Table 9: Processor Extension use together with CrISA-X
Summary of Processor Hardware Extension per algorithms.

Sub	Extended Regester-file	Extended Ports	Extended Instruction-Bus	Extended Data-Bus	Extended Load-Store	Extended Issue'ing Stage	Added Constants Table	Extended Functional Units	Extended Toolchain
Ascon	64reg	Yes	64bits	32bits	2L\S	Yes	N/A	{xor\or-shift, xor:xor xor-and-not }	Yes
Elephant	64reg	Yes	64bits	32bits	2L\S	Yes	Kecacck Const	{xor\or-shift, xor:...xor xor-and-note }	Yes
GIFT	64reg	Yes	64bits	32bits	2 L\S	Yes	Gift Const	{xor\or-shift, xor\or-and}	Yes
Grain	64reg	Yes	64bits	128bits	2 L\S	Yes	N/A	{xor\or-shift, xor\or-and}	Yes
PHOTON	64reg	Yes	64bits	128bits	2 L\S	Yes	N/A	{xor\or-shift\xor:xor xor\or-and, xor-and-not }	Yes
Romulus	64reg	Yes	64bits	32bits	2 L\S	Yes	N/A	{xor\or-shift, xor:xor xor-and-not }	Yes
Sparkle	64reg	Yes	64bits	32bits	2 L\S	Yes	N/A	{xor\or-shift, xor:...xor xor-and-note }	Yes
TinyJ	64reg	Yes	64bits	32bits	2 L\S	Yes	N/A	{xor\or-shift, xor:...xor xor-and-not }	Yes
Xoodyak	64reg	Yes	64bits	32bits	2 L\S	Yes	Xoodoo Const	{xor\or-shift, xor:...xor xor-and-and }	Yes