Accelerated V2X provisioning with Extensible Processor Platform

Henrique S. Ogawa[1], Thomas E. Luther[1], Jefferson E. Ricardini[1], Helmiton Cunha[1], Marcos Simplicio Jr.[2], Diego F. Aranha[3], Ruud Derwig[4] and Harsh Kupwade-Patil[1]

[1] America R&D Center (ARC), LG Electronics US Inc.
{henrique1.ogawa,thomas.luther,jefferson1.ricardini,helmiton1.cunha}@lge.com
harsh.patil@lge.com
[2] University of Sao Paulo, Brazil
mjunior@larc.usp.br
[3] Aarhus University, Denmark
dfaranha@eng.au.dk
[4] Synopsys Inc., Netherlands
ruud.derwig@synopsys.com

Abstract.
With the burgeoning Vehicle-to-Everything (V2X) communication, security and privacy concerns are paramount. Such concerns are usually mitigated by combining cryptographic mechanisms with a suitable key management architecture. However, cryptographic operations may be quite resource-intensive, placing a considerable burden on the vehicle's V2X computing unit. To assuage this issue, it is reasonable to use hardware acceleration for common cryptographic primitives, such as block ciphers, digital signature schemes, and key exchange protocols. In this scenario, custom extension instructions can be a plausible option, since they achieve fine-tuned hardware acceleration with a low to moderate logic overhead, while also reducing code size. In this article, we apply this method along with dual-data memory banks for the hardware acceleration of the PRESENT block cipher, as well as for the $\mathbb{F}_{2^{255}-19}$ finite field arithmetic employed in cryptographic primitives based on Curve25519 (e.g., EdDSA and X25519). As a result, when compared with a state-of-the-art software-optimized implementation, the performance of PRESENT is improved by a factor of 17 to 34 and code size is reduced by 70%, with only a 4.37% increase in FPGA logic overhead. In addition, we improve the performance of operations over Curve25519 by a factor of ~2.5 when compared to an Assembly implementation on a comparable processor, with moderate logic overhead (namely, 9.1%). Finally, we achieve significant performance gains in the V2X provisioning process by leveraging our hardware accelerated cryptographic primitives.

Keywords: V2X · SCMS · Curve25519 · PRESENT cipher · Hardware Acceleration · Custom Extension Instructions · Dual-Data Memory Banks · Extensible Processors

1 Introduction

With the introduction of Vehicle-to-everything (V2X) communications, security and privacy concerns increase at an alarming rate. As V2X technologies become pervasive, there is a need for a V2X gateway in the vehicle [20]. Furthermore, to support the gateway with key management and cryptographic operations, an embedded Hardware Security Module (HSM) is required.

Different HSM variants have been proposed for V2X applications [27, 41], and recently, the CAR 2 CAR (C2C) Communication Consortium proposed security requirements for a V2X HSM [9]. Among the many security requirements for the V2X HSM, cryptographic operations (e.g., digital signatures and encryption) and key management are prominent features. In particular, the proposal calls for digital signatures using ECDSA (Elliptic Curve Digital Signature Algorithm), as well as ECIES (Elliptic Curve Integrated Encryption Scheme) for the encryption of one-time session keys.

Both ECDSA and ECIES are built upon elliptic curve arithmetic and a hash function, and ECIES additionally makes use of a symmetric cipher and a MAC. Supporting elliptic curves requires finite field arithmetic for operands much larger than the typical processor word size. Implementing symmetric algorithms efficiently in embedded system software can be a challenge when side-channel protection is required. It is therefore unsurprising that most HSMs make use of hardware acceleration for their cryptographic operations.

Hardware acceleration can be achieved in two different ways: either by connecting independent memory-mapped co-processor modules to the main CPU or by extending the CPU with custom instructions. It is primarily the throughput and latency requirements that determine which option is more suitable. As an example, the PRESENT cipher's round substitution and permutation operations can be implemented with a combinatorial logic datapath. Connecting such a datapath via a memory-mapped bus interface introduces significant processing latency, which is inherent to the process of moving data through the bus infrastructure.

In comparison, extensible processor platforms allow for the same datapath to become an extension of the base Arithmetic Logic Unit (ALU), which can be accessed in software just like any other instruction. Hence, in a bit-sliced implementation of PRESENT's substitution and permutation operations, dozens of regular CPU instructions can be replaced with a single custom extension instruction.

Replacing software operations by specialized instructions reduces code size, memory accesses and register usage. In addition to the reduced processing latency, the energy consumption decreases [47]. Therefore, extensible processor platforms offer means for hardware acceleration where fine-tuned improvements can be achieved with low logic overhead and reduced energy consumption.

1.1 Hardware Acceleration

Custom Extension Instructions: Several extensible processor platforms support the inclusion of custom logic via extension instructions [7, 16, 29, 36]. In these platforms, closely-coupled hardware interfaces are exposed for the connection of specialized logic modules into the main processor's pipeline. Such connections, however, impose more restrictive critical path constraints: complex custom instructions with longer datapaths can decrease the processor's maximum clock frequency. One solution is introducing pipeline registers to split the instruction datapath into multiple stages, increasing the maximum clock frequency. This approach, however, increases the number of required registers, meaning larger chip area, as well as additional latency. Optionally, an instruction datapath with several pipeline stages can be divided into multiple instructions with shorter datapaths. The throughput of these smaller collective instructions can be improved using Implicit Instruction-Level Parallelism (IILP) techniques [39].

DSP Closely-Coupled Memories: Matrix multiplications, dot products, convolutions, Finite Impulse Response (FIR) filters and Fast Fourier Transform (FFT) are some examples of compute-intensive functions often used by signal processing algorithms. Such functions are intrinsic to many embedded applications and take advantage of DSP hardware support mechanisms incorporated into modern embedded processors [23, 30]. Among these DSP mechanisms, dual-data memory banks (often referred as X and Y memory banks [10, 23, 30]) have been incorporated in order to enable the simultaneous fetching of instruction data plus two-operand data [10, 23]. This enables greater memory access bandwidth for algorithms where repeated operations on arrays is done. An addressing unit supporting variable offset, modulo and bit-reversed addressing patterns offloads array index processing from the main CPU [34]. Cryptographic functions often get implemented using DSP-analogous repetitive array-based techniques, e.g., the PRESENT cipher's encryption/decryption rounds [28] and $\mathbb{F}_{2^{255}-19}$ arithmetic operations [13]. Hence, dual-data memory banks also offer opportunities for performance enhancements in cryptographic processing.

1.2 Applications to V2X

The automotive industry is in the midst of a digital revolution. With the introduction of V2X technology, the need for stronger security primitives is imminent. In the US, Secure Credential Management System (SCMS) is the leading candidate for provisioning and managing V2X certs. Moreover, it ensures privacy-by-design by introducing the concept of a "butterfly key expansion" process. This process provisions pseudonym certificates for vehicles and thereby protects the privacy of the driver. Recently, simplifications to the butterfly key expansion process (the Unified Butterfly Key (UBK) effect) achieve significant processing and bandwidth efficiency [32, 1].

Although the processing gains were impressive, the need for hardware accelerated cryptography for real-world V2X deployment is greater than ever. The current requirement for processing V2X certs (e.g., verification of cryptographic signatures and messages) is under 10 milliseconds [24]. Therefore, to assuage the aforementioned attacks while meeting the demands of faster crypto operations, we leverage newer building blocks for cryptography to implement the UBK scheme.

Contributions

Our work presents a combination of both: custom extension instructions and dual-data memory banks to obtain hardware accelerated implementations of the PRESENT block cipher and finite-field arithmetic for Curve25519. The contributions of our work can be enumerated as follows:

1. Proposal of instruction set extensions for the acceleration of PRESENT and selected operations in $\mathbb{F}_{2^{255}-19}$, based of the profiling analysis.

2. Exploit dual-data memory banks in order to enable IILP for the implemented instruction set extensions.

3. Evaluation of the hardware accelerated crypto functions in the context of a real-world V2X certificate provisioning process (SCMS's UBK).

To the best of our knowledge, our method of combining custom instructions and DSP dual-data memory banks seems novel. Note that the related works have only explored the aforementioned approaches individually. Our tests show that the combination of both the approaches provides performance improvement, coherent software integration and constant-time programming flows, all while maintaining low to moderate logic overhead.

Additionally, we discuss a Man-in-the-middle (MitM) attack when the Regular Butterfly Key (RBK) expansion and the UBK settings coexist (co-existential attack). In defense, we propose a solution for mitigating such an attack.

Organization: This paper is organized as follows: Section 2 describes the main features of a state-of-the-art extensible processor platform supporting custom extension instructions, and a DSP engine equipped with dual-data memory banks. Next, the SCMS's UBK certificate provisioning process is introduced as the V2X application scenario. Details on the design and implementation of the Instruction Set Extensions (ISE) for PRESENT and $\mathbb{F}_{2^{255}-19}$ arithmetic are described in Section 3. Section 4 outlines the strategy for the combination of the developed ISE with the XY Memory. Section 5 details the testing frameworks for the evaluation of performance and logic overhead. Section 6 describes the achieved results, including comparison between different implementation strategies and related works.

2 Related Work

2.1 Extensible Processor Platform

The DesignWare® ARC® Processor IP portfolio from Synopsys includes a wide range of processors for embedded applications [36]. The ARC EM Processor Family, based on the 32-bit ARCv2 instruction set, features a Harvard memory-processor architecture for simultaneous instruction and memory access [36]. A broad set of DSP, security and interconnection processor components allows these processors to be configured for highly specialized embedded applications. The ARC Processor EXtension (APEX) technology enables the integration of user-defined custom instructions [36], while the ARC XY Memory DSP Option brings a DSP engine for IILP in ARC EM processors [34].

APEX Technology APEX technology allows for customization of the ARC processor implementation through user-defined instructions and auxiliary (AUX) registers [36]. The provided pipeline interfaces allow for the implementation of specialized and enlarged-width datapaths. This enables smooth software integration, reduced interfacing complexity, lower gate count and processing output latency when compared to a bus-based co-processor. Figure 1 shows the anatomy of a custom instruction in APEX technology, with respect to an overview of ARC processor's pipeline.

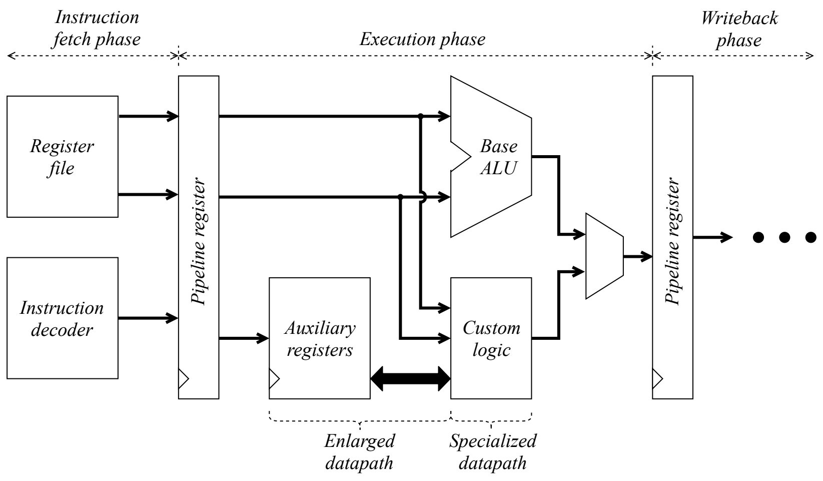

Figure 1: Anatomy of an instruction extension on an extensible processor platform [33, 11]

ARC XY Memory DSP Option Alongside the extension instructions, IILP can be achieved through the ARC XY Memory DSP Option, an IILP engine for fast and closely-coupled memory access [39]. As depicted in Figure 2, the ARC XY Memory system consists of dual-data memory banks (i.e. X and Y banks) operated by an Address Generation Unit (AGU) and an internal dedicated DMA engine, which allows the CPU to read two source operands and store the result in the same cycle [34]. This also provides increased code density since explicit array index updates can be directly leveraged with the AGU's address update mechanism [34].

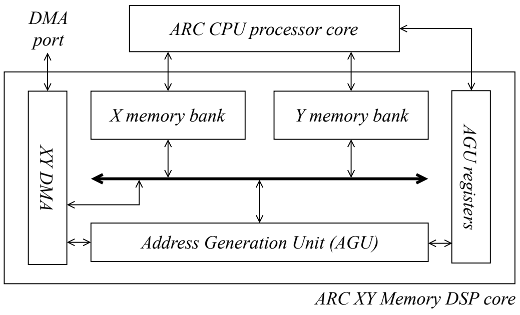

Figure 2: Overview of the ARC XY Memory DSP Core [39]

2.2 V2X Provisioning

In the past few years, the growing interest in enabling vehicular communications has led to several research and standardization efforts [24]. Given the critical nature of services based on connected vehicles, many such efforts focus on creating a security foundation for their operation. Commonly, this is addressed by means of a Vehicular Public Key Infrastructure (VPKI). Although a VPKI is similar to a regular PKI, it has different requirements, in particular 1) privacy by design and 2) the ability to cope with the resource-constrained nature of onboard equipment. Privacy is usually addressed by loading each vehicle with multiple pseudonym certificates, i.e., certificates that do not identify their owners [19]. Indeed, this is the approach employed in the Cooperative Intelligent Transportation Systems (C-ITS) and in the Security Credential Management System (SCMS), which are, respectively, the most prominent VPKI proposals in Europe and in the US [12, 8]. Reducing the computational burden on the vehicle side involves both a careful VPKI design and efficient implementations. Since this article focuses on providing high-performance hardware accelerated implementations for the SCMS architecture, in the following we describe its design in more detail.

SCMS: The SCMS was developed in cooperation with the U.S. Department of Transportation (USDOT) and the automotive industry. Its main goal is to enable a safe, secure and privacy-preserving V2X communication environment, where vehicles can trust each other's messages and the system itself. To accomplish this, SCMS's architecture includes a VPKI for issuing multiple short-lived, pseudonym certificates to authorized vehicles. Each vehicle can then use its certificates to digitally sign its messages, so their authenticity can be verified. A vehicle can also avoid tracking attempts by its peers if it periodically changes the pseudonym employed along the way: as a result, it should not be straightforward to link different messages to the same vehicle just by analyzing the corresponding certificates' contents.

In addition, a clever separation of roles during the certificate provisioning procedure prevents system authorities from linking a given vehicle to its pseudonym certificates. Privacy is, thus, preserved unless such authorities collaborate. To prevent abuse, however, a special entity called a Misbehavior Authority (MA) is expected to receive and analyze misbehavior reports by system entities. If a transgression is confirmed, the MA can collaborate with other system entities to identify the culprit, besides revoking its pseudonym certificates. This prevents malicious users or vehicles equipped with faulty components from disrupting the system for too long. Similarly, an elector-based trust management approach allows system authorities themselves to be revoked in case of misconduct.

(Unified) Butterfly Keys: When compared to other VPKIs, one of the main benefits of SCMS is its highly efficient pseudonym certificate provisioning process, called "butterfly key expansion". Two versions of this process exist in the literature: the one originally described in the SCMS article [43], hereby called RBK; and the optimized version proposed in [32], named the UBK approach. The upper part of Table 2 summarizes and compares the RBK and UBK protocols, using the notation listed in Table 1 for easy reference.

In essence, RBK requires vehicles to compute two sets of "caterpillar keys", both created using Elliptic Curve Cryptography (ECC): the signature key pair $(s, S \leftarrow s{\cdot}G)$ and the encryption signature key pair $(e, E \leftarrow e{\cdot}G)$. The vehicle then establishes a secure communication channel with a Registration Authority (RA) and sends to it: 1) the public keys S and E; 2) two pseudorandom functions, f_1 and f_2; as well as 3) long-term credentials (proving that it is authorized to request pseudonym certificates). The RA then expands each caterpillar public key into several "cocoon public keys" by applying the vehicle-provided f_1 and f_2 This leads to as many public key $(\hat{S}_i \leftarrow S + f_1(i){\cdot}G, \hat{E}_i \leftarrow E + f_2(i){\cdot}G)$ tuples as the number of pseudonym certificates the vehicle should receive. Subsequently, the RA sends the individual tuples to one or more Pseudonym Certificate Authorities (PCAs), which are the entities responsible for issuing pseudonym certificates. To preserve the vehicles' privacy, the RA-to-PCA requests are such that tuples corresponding to different vehicles are shuffled together, and no information about the vehicles' identities is provided to the PCA.

The PCA, in turn, randomizes the received signature cocoon keys $\hat{S}_i$, obtaining the butterfly keys $\mathrm{U}_i \leftarrow \hat{S}_i + r_i{\cdot}G$. Those keys are signed by the PCA together with any relevant metadata (e.g., a validity period), thus producing the corresponding pseudonym certificates $cert_i$. Finally, to prevent the RA from learning the certificates' contents (and, thus, from linking $cert_i$ to the requesting vehicle), the PCA uses $\hat{E}_i$ to encrypt its response to the RA; as a result, only the vehicle can decrypt the received package using its private key $\hat{e}_i \leftarrow e + f_2(i)$, and verify that $cert_i$ was correctly issued. In addition, in RBK this encrypted package is also signed by the PCA to prevent a malicious RA from acting as a MitM: otherwise, the RA might provide the PCA with a bogus encryption key $\widetilde{E}_i$, for which the RA knows the private key; then, it could decrypt the PCA's response, map $cert_i$ to the vehicle's identity, and then re-encrypt everything with the correct $\hat{E}_i$ to avoid suspicion.

Compared to RBK, the main distinctive feature of the UBK approach is that it involves only one caterpillar public and private key pair $(x, X \leftarrow x{\cdot}G)$ instead of two. Accordingly, a single public key X is provided by vehicles to the RA, which expands X into several cocoon public keys $\widehat{X}_i \leftarrow X + f(i){\cdot}G$ using pseudorandom function f. Similarly to RBK, the RA shuffles $\widehat{X}_i$ from different vehicles before relaying them to the PCA. Finally, the PCA creates a randomized butterfly public key $\mathrm{U}_i \leftarrow \widehat{X}_i + r_i{\cdot}G$, places it into a certificate, and encrypts the result with $\widehat{X}_i$ before responding to the RA. Unlike RBK, however, in UBK there is no need to sign the encrypted package: since UBK ties the encryption key $\widehat{X}_i$ to the certificate's key U_i, the RA cannot provide a fake encryption key $\widetilde{E}_i$ without tampering the certificate itself. In other words, as shown in [32], vehicles can indirectly assert that the PCA used the correct encryption key $\widehat{X}_i = (x + f(i)){\cdot}G$ simply by verifying that the value of U_i enclosed in the certificate satisfies $\mathrm{U}_i = (\hat{x}_i + r_i){\cdot}G$. Therefore, the UBK approach can be seen as an optimization of RBK, reducing bandwidth usage and processing costs when provisioning pseudonym certificates.

Table 1: General notation and symbols

Symbol	Meaning
G	The generator of an elliptic curve group
sig	A digital signature
$cert$	A digital certificate
$\mathrm{U}, \mathcal{U}_{rbk}, \mathcal{U}_{ubk}$	Public signature keys (stylized $\mathcal{U}_{rbk}$ and $\mathcal{U}_{ubk}$: reserved for PCA)
$\mathrm{u}, u_{rbk}, u_{ubk}$	Private keys corresponding to U, $\mathcal{U}_{rbk}$ and $\mathcal{U}_{ubk}$
S, s	Public and private caterpillar signature keys
E, e	Public and private caterpillar encryption keys
$\hat{S}, \hat{s}$	Public and private cocoon signature keys
$\hat{E}, \hat{e}$	Public and private cocoon encryption keys
X, x	Public and private unified caterpillar keys
$\widehat{X}, \hat{x}$	Public and private unified cocoon keys
β	Number of cocoon keys in certificate batch
f, f_1, f_2	Pseudorandom functions
$Enc(\mathcal{K}, str)$	Encryption of bitstring str with key $\mathcal{K}$
$Dec(\mathcal{K}, str)$	Decryption of bitstring str with key $\mathcal{K}$
$Sign(\mathcal{K}, str)$	Signature of bitstring str, using key $\mathcal{K}$
$Ver(\mathcal{K}, str)$	Verification of signature on str, using key $\mathcal{K}$

Table 2: Explicit pseudonym certificates in the RBK and UBK expansion procedures, as well as co-existence attack when vehicle believes to be running UBK, but PCA runs RBK. Operations highlighted in light gray are made unnecessary by the UBK optimization; operations in dark gray are specific to the co-existence attack.

	Vehicle	$\rightarrow$	RA	$\rightarrow$	PCA	$\rightarrow$	RA	$\rightarrow$	Vehicle
RBK	$s \xleftarrow{\$} \mathbb{Z}_q$ $S = s \cdot G$ $e \xleftarrow{\$} \mathbb{Z}_q$ $E = e \cdot G$	S, f_1 E, f_2	$\hat{S}_i \leftarrow S + f_1(i) \cdot G$ $\hat{E}_i \leftarrow E + f_2(i) \cdot G$ $(0 \leqslant i < \beta)$	$\hat{S}_i$, $\hat{E}_i$	$r_i \xleftarrow{\$} \mathbb{Z}_q$ $\mathrm{U}_i \leftarrow \hat{S}_i + r_i \cdot G$ $sig_i \leftarrow Sign(u_{rbk}, \{\mathrm{U}_i, \mathtt{meta}\})$ $cert_i \leftarrow \{\mathrm{U}_i, \mathtt{meta}, sig_i\}$ $pkg \leftarrow Enc(\hat{E}_i, \{cert_i, r_i\})$ $res \leftarrow \{pkg, Sign(u_{rbk}, pkg)\}$	res	—	res	$\hat{e}_i \leftarrow e + f_2(i)$ $Ver(\mathcal{U}_{rbk}, res)$ $\{cert_i, r_i\} \leftarrow Dec(\hat{e}_i, pkg)$ $Ver(\mathcal{U}_{rbk}, cert_i)$ $\mathrm{u}_i \leftarrow s + f_1(i) + r_i$ $\mathrm{u}_i \cdot G \overset{?}{=} \mathrm{U}_i$
UBK	$x \xleftarrow{\$} \mathbb{Z}_q$ $X \leftarrow x \cdot G$	X, f	$\widehat{X}_i \leftarrow X + f(i) \cdot G$ $(0 \leqslant i < \beta)$	$\widehat{X}_i$	$r_i \xleftarrow{\$} \mathbb{Z}_q$ $\mathrm{U}_i \leftarrow \widehat{X}_i + r_i \cdot G$ $sig_i \leftarrow Sign(u_{ubk}, \{\mathrm{U}_i, \mathtt{meta}\})$ $cert_i \leftarrow \{\mathrm{U}_i, \mathtt{meta}, sig_i\}$ $pkg \leftarrow Enc(\widehat{X}_i, \{cert_i, r_i\})$	pkg	—	pkg	$\hat{x}_i \leftarrow x + f(i)$ $\{cert_i, r_i\} \leftarrow Dec(\hat{x}_i, pkg)$ $Ver(\mathcal{U}_{ubk}, cert_i)$ $\mathrm{u}_i \leftarrow \hat{x}_i + r_i$ $\mathrm{u}_i \cdot G \overset{?}{=} \mathrm{U}_i$
Attack	$x \xleftarrow{\$} \mathbb{Z}_q$ $X \leftarrow x \cdot G$	X, f	$\widehat{X}_i \leftarrow X + f(i) \cdot G$ $z_i \xleftarrow{\$} \mathbb{Z}_q$ $\widetilde{E}_i \leftarrow z_i \cdot G$ $(0 \leqslant i < \beta)$	$\widehat{X}_i$, $\widetilde{E}_i$	$r_i \xleftarrow{\$} \mathbb{Z}_q$ $\mathrm{U}_i \leftarrow \widehat{X}_i + r_i \cdot G$ $sig_i \leftarrow Sign(u_{rbk}, \{\mathrm{U}_i, \mathtt{meta}\})$ $cert_i \leftarrow \{\mathrm{U}_i, \mathtt{meta}, sig_i\}$ $pkg \leftarrow Enc(\widetilde{E}_i, \{cert_i, r_i\})$ $res \leftarrow \{pkg, Sign(u_{rbk}, pkg)\}$	res	$\{cert_i, r_i\} \leftarrow Dec(z_i, res)$ **(now RA knows cert$_i$)** $pkg \leftarrow Enc(\widehat{X}_i, \{cert_i, r_i\})$	pkg	$\hat{x}_i \leftarrow x + f(i)$ $\{cert_i, r_i\} \leftarrow Dec(\hat{x}_i, pkg)$ $Ver(\mathcal{U}_{rbk}, cert_i)$ $\mathrm{u}_i \leftarrow \hat{x}_i + r_i$ $\mathrm{u}_i \cdot G \overset{?}{=} \mathrm{U}_i$

An UBK/RBK co-existence attack: The security of RBK and UBK is individually analyzed in their respective documentation [43, 32]. Nevertheless, in what follows we describe a novel attack that arises if: 1) both protocols co-exist at a certain point in time; 2) vehicles are led to believe they are running UBK when the PCA actually runs RBK. Even though this attack does not invalidate either RBK's or UBK's individual security claims, since it assumes the protocols are not run exactly as specified, this corresponds to a quite practical scenario. Indeed, this would be the case if some PCAs in operation decide to support only one version of the protocol, or even if some PCAs can run both RBK and UBK.

In this scenario, a malicious RA that wants to be able to track vehicles can perform the following MitM attack (see the bottom part of Table 2). First, the rogue RA announces to vehicles that it is able to issue UBK certificates. The victim, attracted by the UBK procedure's higher efficiency, follows the protocol as usual: it computes the public caterpillar key X and sends it together with the pseudorandom function f to the RA. The RA, in turn, computes the correct cocoon keys $\widehat{X}_i \leftarrow X + f(i) \cdot G$, for $0 \leqslant i < \beta$; however, the RA also computes β cocoon encryption keys $\widetilde{E}_i \leftarrow z_i \cdot G$ for arbitrary values of z_i. The RA then sends the pair $(\widehat{X}_i, \widetilde{E}_i)$ to a PCA running RBK, as if such keys were generated according to the RBK protocol. The PCA, unbeknown of the attack, simply runs the RBK protocol for generating pseudonym certificates $cert_i$, encrypts it together with the randomization factor r_i, and then signs this encrypted package. The RA, instead of acting as a proxy, simply discards this final signature from the PCA's response and recover the corresponding $cert_i$ by means of the decryption key z_i. To complete the MitM attack, the RA also re-encrypts the pair $\{cert_i, r_i\}$ with $\widehat{X}_i$, and sends the result to the requesting vehicle as if the encryption was performed by the PCA. Since the response received by the vehicle is identical to a genuine UBK package, in principle that vehicle might be believe that the certificates where indeed generated by an UBK-enabled PCA. Meanwhile, the RA learns the contents of all pseudonym certificates issued through it and, thus, can link the real identity of the vehicle to those certificates when they are used in the field. Hence, the described co-existence attack violates one fundamental property of RBK and UBK: the unlinkability of pseudonym certificates by any (non-colluding) system entity.

Countermeasures to co-existence attacks: We note that the described attack is only possible when the vehicle mistakenly believes that the PCA is running UBK, but in reality the protocol being run is RBK: in that case, the vehicle does not require a signature on the encrypted package, which is required in RBK to prevent MitM attempts by the RA. Therefore, to prevent the attack, it suffices to ensure that vehicles can verify which protocol has been actually used by the PCA. Note that, there are at least two simple approaches for accomplishing this, described in what follows.

The first solution consists of including a protocol identifier (e.g., "0" for RBK, and "1" for UBK) in the PCA's certificate. As a result, the vehicle can check whether the PCA runs UBK or RBK, and then verify the received pseudonym certificates' authenticity using the correct procedure. As long as PCA certificates for RBK do not share the same public key with any PCA certificate for UBK, vehicles cannot be tricked into accepting RBK pseudonym certificates as if they were generated using UBK, thus preventing the attack. The overhead of this approach is negligible, since it can be as small as adding a single bit to the PCAs' long term certificates.

As an alternative approach, the PCA could use the pseudonym certificate's metadata itself to inform the vehicle about which protocol was employed for its generation. The overhead in this case can once again be as small as a single bit to differentiate between UBK and RBK. Nevertheless, this approach is less efficient because: 1) there are many more short-term, pseudonym certificates in the system than long-term, PCA certificates; and 2) even though this extra bit is only useful during the issuance process, it must be transmitted afterwards when vehicles sign their own messages.

ECC and PRESENT Cipher: In the original butterfly key expansion process proposed in the SCMS proposal [43], it assumes the use of ECDSA (digital signature algorithm), ECIES (asymmetric encryption scheme) and AES (block cipher) [26, 15, 25].

In our UBK implementation, we focus on curves defined over prime fields which can be represented in the Montgomery [22] (or Twisted Edwards [3]) model, allowing faster formulas [5]. Particularly the twisted Edwards representation of Curve25519 is known as edwards25519.

The edwards25519 curve enables the use of Edwards-curve Digital Signature Algorithm (EdDSA), which is a signature scheme variant of Schnorr signatures based on elliptic curves represented in the Edwards model [4]. Like other discrete-log based signature schemes, EdDSA requires a secret value, or nonce, unique to each signature. In order to reduce the risk of random number generator failures, EdDSA calculates this nonce deterministically, as the hash of the message and the private key. Thus, the nonce is very unlikely to be repeated for different signed messages. This reduces the attack surface in terms of random number generation and improves nonce misuse resistance during the signing process. However, high quality random numbers are still needed for key generation. Given the aforementioned advantages of EdDSA over ECDSA we choose EdDSA as the underlying signature algorithm for the UBK provisioning process.

In recent efficient X25519 and EdDSA implementations, the finite field arithmetic operations were implemented in ARM-Assembly, making extensive use of the multiply-and-accumulate DSP instructions [13]. Leveraging these assembly-level optimizations, X25519 key exchange protocol required 907,240 cycles, and EdDSA key generation, sign and verification consumed 347,225, 496,039 and 1,265,078 cycles respectively. An architecture for a Curve25519 co-processor was proposed and implemented [31]. Although countermeasures for simple and differential power analysis attacks were included, a throughput of 27,500 operations per second was achieved for the point multiplication operation.

For the symmetric encryption algorithm, we propose the use of the PRESENT block cipher. The PRESENT cipher was one of the first hardware oriented proposals implemented in resource-constrained environments [6]. Inherently, PRESENT is an ideal target for hardware implementation. Although constant-time PRESENT software implementations exist, they require considerable amount of bit slicing operations [28].

At a glance, PRESENT's 4-bit S-box can be implemented as a lookup table, however this approach is vulnerable to cache memory timing attacks [28]. A constant-time and bit-sliced software implementation targeting ARM processors is presented in [28]. In this work, the 64-bit S-box layer is implemented as 14 boolean operations over the four 16-bit word inputs, and 15 boolean operations for the inverse S-box counterpart [28]. The proposed methods of interchanging permutations and S-boxes, and the decomposition of permutations resulted in substantial performance improvements for software implementations. Furthermore, side-channel countermeasures beyond the scope of timing attacks are also explored in [28].

A compact PRESENT implementation targeting FPGA platforms was proposed in [40]. Instead of using inferred BRAM blocks to implement the S-box, the authors followed a boolean decomposition method. The proposed approach resulted in a small-footprint PRESENT implementation, with 295 cycles of processing latency. The implementation required 201 flip-flops and 222 Look-up Tables (LUTs) in a Virtex-5 FPGA.

In the case of AES, an ISE for the SPARC V8 32-bit processor achieved speedups by a factor of 9 for encryption/decryption, with a moderate increase in silicon area [42]. Similarly, an ISE for the 32-bit CRISP processor was implemented for a set of bit-sliced crypto algorithms. In particular, PRESENT's performance was improved by a factor of 1.42 [14]. Furthermore, the code size was reduced by 18% compared to the implementation using only traditional instructions.

3 Instruction Set Extensions

3.1 Instruction Extensions for the PRESENT Cipher

We propose the implementation of single-cycle non-blocking extension instructions for the computation of PRESENT encryption, decryption and key update round. Since the cipher's block size is 64-bits, we make use of two AUX registers to implement a 64-bit datapath for the encryption instruction, and another two AUX registers for the decryption instruction. Likewise, four AUX registers are used in the implementation of a 128-bit datapath for the key schedule instruction. Table 3 summarizes our proposed extension instructions, alongside the logic modules required for the implementation of the respective instruction's datapath.

`enc64` instruction datapath: Our PRESENT S-box implementation uses the boolean equations for the S-box output bits that are obtained from the application of Karnaugh mapping, followed by the Quine-McCluskey logic minimization algorithm [40]. Listing 9 in the appendix contains a Verilog snippet for the implementation of the boolean S-box. The permutation layer can be implemented in hardware through simple bit-wiring, as mentioned in [40]. Listing 10 in the appendix shows our proposed Verilog snippet for

Table 3: Proposed APEX ISE for PRESENT [6]

Proposed instruction	AUX registers	Components	Module	Instances
`enc64`: block encryption	`BLK0_AR`, `BLK1_AR`	S-box	`sbox`	16
		Permutation Layer	`pLayer`	1
`keysch`: key schedule	`KEY0_AR`, `KEY1_AR`, `KEY2_AR`, `KEY3_AR`	S-box	`sbox`	2
		61-bit left rotation	`leftRot_61`	1
		5-bit input XOR	-	1
`dec64`: block decryption	`IBLK0_AR`, `IBLK1_AR`	Inverse S-box	`isbox`	16
		Inverse Permutation Layer	`ipLayer`	1

the implementation of the `pLayer` module. The S-box layer [6] is a module composed by sixteen instances of `sbox` together within a single module. The datapath for our proposed `enc64` instruction is obtained by wiring the outputs of the 128-input XOR module (*addRoundKey* [6]) to the inputs of the S-box layer module, whose outputs are then connected to the `pLayer` (Listing 10) module's inputs, as shown in Figure 3. The `blk_out` signal is the output of a single PRESENT round.

`keysch` instruction datapath: The 61-bit left rotation step of the PRESENT key schedule can be achieved similarly to the bit-wiring method used for the `pLayer` implementation. Listing 11 in the appendix shows a Verilog snippet for the implementation of the `leftRot_61` module. The datapath for the proposed `keysch` instruction is obtained by connecting the `leftRot_61` module, `sbox` modules and XOR gates as shown in Figure 3. The `key_out` output signal shown in Figure 3 is the result of a single round of the key schedule [6], and the `key_out[127:64]` output signal is the round sub-key key for a given encryption round.

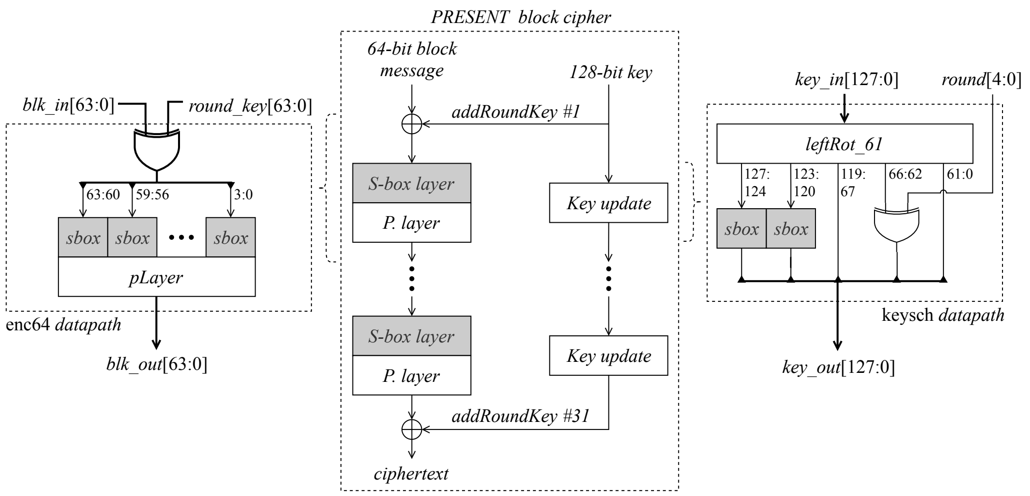

Figure 3: Description of proposed `enc64` and `keysch` instructions in PRESENT [6]

`dec64` instruction datapath: For PRESENT's inverse S-box [6] module (`isbox`) we repeat the procedure used for the `sbox` module, which is presented in the Verilog snippet of Listing 12 in the appendix. The inverse permutation layer [6] (`ipLayer`) can also be implemented in hardware through simple bit-wiring. Listing 13 in the appendix shows a Verilog snippet for the implementation of the `ipLayer` module.

The inverse S-box layer is composed of sixteen `isbox` module instances (Listing 12). The datapath of our proposed `dec64` instruction is obtained by connecting the outputs of the `ipLayer` module to the inputs of the inverse S-box layer, whose output is then wired to the 128-input XOR gate (i.e. *addRoundKey* [6]), as depicted in Figure 9 in the appendix.

APEX integration: In order to integrate the instructions proposed in Table 3 into the APEX pipeline, the datapath modules of `enc64`, `keysch` and `dec64` instructions (Figures 3 and 9) must be connected to the designated AUX registers. The AUX registers shown in Table 3 are visible to the traditional load/store instructions. The instruction operands and result can be used to transfer data to and from the extension modules, as shown in Figure 4. The `keysch` instruction operates without any source operands, as it uses the data directly from assigned AUX registers. Moreover, `keysch` returns the value currently held by `KEY2_AR`, which is the lower 32 bits of the round subkey. The `enc64` and `dec64` instructions take two source operands: `src1` and `src2`, which are the upper and lower 32-bits, respectively, of the round subkey. The 64-bit message block is read from the `BLK0_AR` and `BLK1_AR` AUX registers (or `IBLK0_AR` and `IBLK1_AR` for `dec64`). The `enc64` and `dec64` instructions do not have any instruction output values. Instead, the results are written to their respective AUX registers.

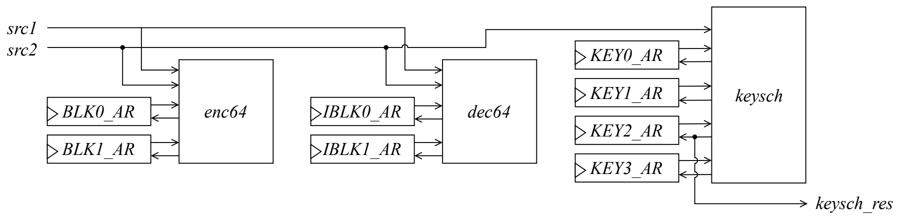

Figure 4: Integration of `enc64`, `keysch` and `dec64` to AUX registers

3.2 Instruction Extensions for $\mathbb{F}_{2^{255}-19}$ Arithmetic

Characterization of $\mathbb{F}_{2^{255}-19}$ multiplicative operations: Curve25519 arithmetic operations [2] are defined over the Galois Field $GF(p)$ (or $\mathbb{F}_p$), where $p = 2^{255} - 19$. For a 32-bit platform, the plain representation of a single 255-bit finite field element (`fe`) requires eight 32-bit words. Henceforth, a 32-bit word is referred to as `word`.

We first start by running the DesignWare MetaWare Debugger execution profiling tool over a reimplementation in software of [13] adapted to the Synopsys ARC. It is expected, based on the literature and related works, that the finite field multiplicative functions are the performance bottleneck in Curve25519 implementations. In our optimized software implementation of Curve25519, the multiplicative operations are listed in Table 4.

Table 4: List of multiplicative-based functions in $\mathbb{F}_{2^{255}-19}$ implementation [13]

Function	Description
`fe_mul_word`	Multiplication of a `fe` field element by a `word`, followed by a weak reduction [13]
`fe_sqr`	Multiplication of a `fe` field element by itself, followed by a weak reduction [13]
`fe_power`	Multiplication of a `fe` field element (with weak reduction) by itself N times (`fe_sqr` in loop)
`fe_mul`	Multiplication of two distinct `fe` field elements, followed by a weak reduction [13]

Table 5: Percentage of the execution time taken by $\mathbb{F}_{2^{255}-19}$ multiplicative operations listed on Table 4, with regards to the specified Curve25519 functions

	X25519	Ed25519 Key Gen.	Ed25519 Sign	Ed25519 Verify
Fujii *et al.* [13]	79%	81%	81%	84%
ARC SW	73%	75%	75%	81%

We profiled the field multiplicative operations using X25519 and Ed25519. The results show that around 80% of the total cycle count is consumed by the functions listed in Table 4. These percentages, shown in Table 5, give us a good indication that custom extension instructions for the multiplicative finite field arithmetic would have a considerable impact on the overall performance. We therefore focus our work on these operations.

256x256-bit multiplication: The first challenge in designing custom instructions to improve the performance of the functions listed in Table 4 is to outline an instruction datapath. Large multiplier units can be ruled out, as they require a large amount of scarce hardware resources, which are limited in embedded hardware platforms. Thus, instead of creating a separate datapath for each of the multiplicative operations listed, we propose a method to implement custom extension instructions for `fe_sqr`, `fe_power` and `fe_mul` through a unified datapath based on `fe_mul_word`.

Consider the multiplication of a field element by a `word` (`mul_word` operation) using 32x32 bit multiplication as shown by the schoolbook multiplication scheme of Figure 5. In this context, where {`a[7],...,a[0]`} are the 32-bit words composing a field element, the full resulting product (including carry) would be represented as the 288-bit sequence {`p0[8],...p0[0]`}, as follows:

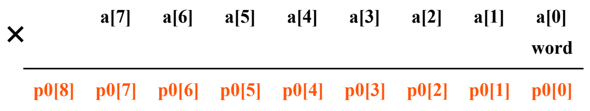
×
a[7] a[6] a[5] a[4] a[3] a[2] a[1] a[0]
word
p0[8] p0[7] p0[6] p0[5] p0[4] p0[3] p0[2] p0[1] p0[0]

Figure 5: `mul_word` - scheme of a multiplication of a `fe` field element by a `word`

Naively extending the idea in Figure 5 for the subsequent product rows, it is evident that `fe` × `fe` full multiplication would require a 512-bit accumulator. A 512-bit accumulator unit would require larger register and adder units that would not be used to their full extent during the intermediary multiplication rows computation. Figure 6 shows our proposed method on achieving a 256x256-bit full multiplication using a 288-bit accumulator-and-shifter instead of a 512-bit accumulator. The key observation is that each one of the sixteen 32-bit words composing the final full 512-bit product can be retrieved as soon as a multiplication row is obtained, i.e., a `mul_word` operation is performed. Figure 6 illustrates our proposed scheme for the 256x256-bit multiplication combining the `mul_word` operation (Figure 5) with a 288-bit accumulator-and-shifter.

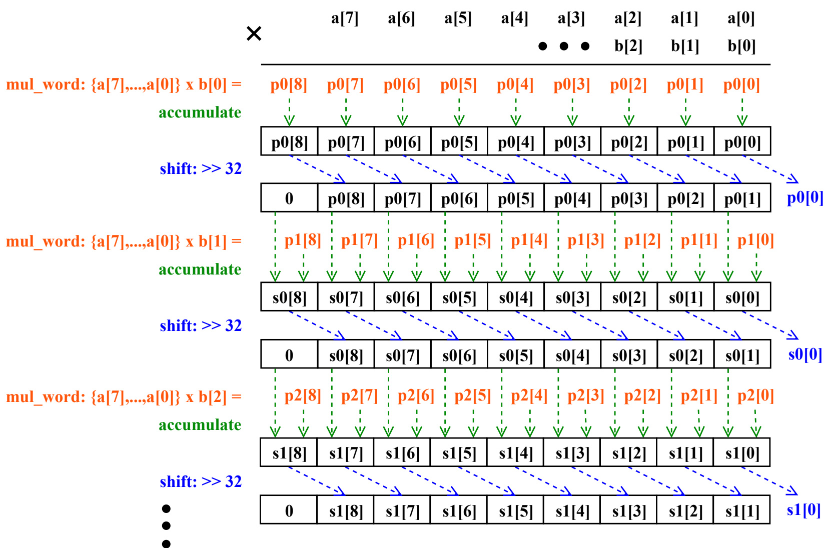
×
a[7] a[6] a[5] a[4] a[3] a[2] a[1] a[0]
b[2] b[1] b[0]
mul_word: {a[7],....,a[0]} x b[0] =
p0[8] p0[7] p0[6] p0[5] p0[4] p0[3] p0[2] p0[1] p0[0]
accumulate
p0[8] p0[7] p0[6] p0[5] p0[4] p0[3] p0[2] p0[1] p0[0]
shift: >> 32
0 p0[8] p0[7] p0[6] p0[5] p0[4] p0[3] p0[2] p0[1]
p0[0]
mul_word: {a[7],....,a[0]} x b[1] =
p1[8] p1[7] p1[6] p1[5] p1[4] p1[3] p1[2] p1[1] p1[0]
accumulate
s0[8] s0[7] s0[6] s0[5] s0[4] s0[3] s0[2] s0[1] s0[0]
shift: >> 32
0 s0[8] s0[7] s0[6] s0[5] s0[4] s0[3] s0[2] s0[1]
s0[0]
mul_word: {a[7],....,a[0]} x b[2] =
p2[8] p2[7] p2[6] p2[5] p2[4] p2[3] p2[2] p2[1] p2[0]
accumulate
s1[8] s1[7] s1[6] s1[5] s1[4] s1[3] s1[2] s1[1] s1[0]
shift: >> 32
0 s1[8] s1[7] s1[6] s1[5] s1[4] s1[3] s1[2] s1[1]
s1[0]

Figure 6: Proposed scheme for 512-bit multiplication using the `mul_word` operation and a 288-bit accumulator-and-shifter

Starting at the multiplication pivot `b[0]`, a `mul_word` operation is performed. The resulting 288-bit product row is added to the value currently held by the 288-bit accumulator (which is zero for the first operation). The accumulator's rightmost word `p0[0]` becomes the output value, and the 288-bit accumulator register is shifted 32 bits to the right. This ensures that the accumulator will never overflow. The procedure is repeated until pivot `b[7]`; at this point the collected output values {`s6[0],...,s1[0],s0[0],p0[0]`} are the lower 256 bits of the 512-bit multiplication result. The upper 256 bits are the ones being held by the 288-bit accumulator. The consecutively captured {`s14,...s7,s6[0],...,s0[0],p0[0]`} set of `words` compose the final full 512-bit product. One clear advantage of the proposed scheme is the weak reduction process [13]. It can be started right after the `b[7]` pivot is reached, being performed in parallel with the shifting retrieval process of the upper eight `words`, saving processing time and temporary registers.

Instruction set extension: The first consideration when organizing the ISE for the multiplicative operations listed in Table 4 is that we aim to design an unified instruction datapath module. This approach differs from the ISE for PRESENT (Table 3), where each one of the APEX instructions had dedicated datapaths and AUX registers. In this manner, APEX technology also offers the option to create instruction extension groups [33]. This feature allows instructions within the same group to access shared datapath modules and AUX registers. Table 6 shows the proposed custom extension instructions for the $\mathbb{F}_{2^{255}-19}$ multiplicative operations listed in Table 4. These will share hardware resources from a unified datapath construction. Additionally, the field element, which is one of the operands of the `mword` instruction, is available to the APEX instruction datapath by means of the `FEi_AR` registers, where `i = {0,...,7}`.

Table 6: List of proposed custom extension instructions for multiplicative operations on $\mathbb{F}_{2^{255}-19}$, with correspondent AUX registers and datapath modules

Proposed instruction	Description	Auxiliary registers	Datapath modules
`mword`	Multiply `fe` element (stored in AUX registers) by `word`, and accumulate the 288-bit result (`mul_word` operation, Figure 5)	`FE0_AR`, `FE1_AR`, `FE2_AR`, `FE3_AR`, `FE4_AR`, `FE5_AR`, `FE6_AR`, `FE7_AR`	`mul_word`, `adder_288`, `shift_reg`
`shacc`	Shift 288-bit accumulator by 32 bits to the right, and return the least significant 32-bit (shift: $\gg 32$ operation, Figure 6)		
`rsacc`	Reset 288-bit accumulator		

`mul_word` module: The first step towards the implementation of the `mul_word` operation is the design of the smallest arithmetic unit: the 32x32-bit (`word`-by-`word`) multiplication unit with carry-in and carry-out signals. As the target platform for this work is an FPGA device, we can make use of the DSP slices to implement the `mul32` modules.

The `mul32` module is used as the building block for the implementation of the `mul_word` module shown in Figure 7. A total of eight `mul32` modules with cascaded carry signals are used; `a` is the finite field element, and `out` is the 288-bit output. Arguably, this module could be split into pipeline stages to achieve higher throughput. However, since minimal latency is our main goal, we choose not to introduce pipeline registers, as it would result in a few extra cycles of latency.

APEX integration: For the integration of the instructions' datapath modules into the APEX pipeline, the AUX registers are directly connected to the `mul_word` module's field element input ports, as shown in Figure 8. The `word` operand of `mul_word` module is passed to the instruction datapath as the `mword` instruction's source operand (`src2`). The `mul_word`'s output value is forwarded to one of the `adder_288` module's inputs, which then sums this value with the one currently stored in the `shift_reg` module. Simultaneously, the `mword` instruction also enables the `shift_reg`'s load signal, making the `shift_reg`

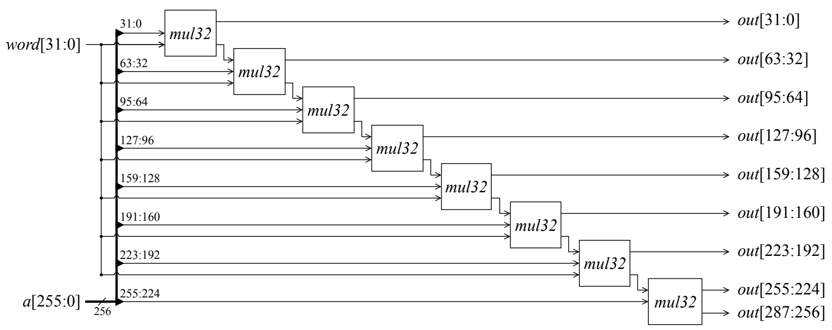

Figure 7: `mul_word` module - 256x32-bit multiplier unit from Figure 5 scheme

module store the current `adder_288`'s output value. Moreover, the `shacc` instruction, which does not have source operands, simply enables the `shift_en` signal to shift the contents of the `shift_reg`'s 288-bit internal register. With the instruction datapath shown in Figure 8, a `mul_word` operation gets executed in two clock cycles, assuming that the field element source operands are already available in the AUX registers.

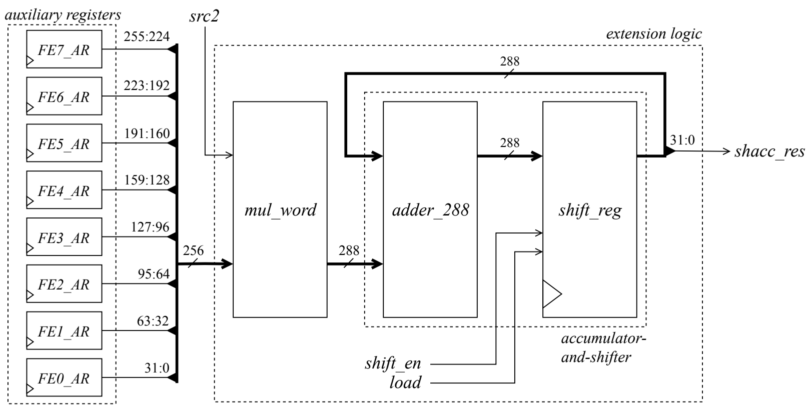

Figure 8: APEX user extension architecture: integration of `mul_word`, `adder_288` and `shift_reg` modules

4 Programming with Dual-Data Memory Banks

The primary purpose of dual-data memory banks is to provide greater bandwidth. However, as discussed in [30], [10] and [23], the main caveat is the proper assignment of data to each memory bank. This often becomes the most challenging task for obtaining optimal performance. We illustrate this problem through the code snippet of Listing 2, an example implementation of a dot-product with dual-data memory banks. We start by introducing this example using the C snippet below:

```
...
void dot_product (uint32_t *a, uint32_t *b, uint32_t *out) {
    for (int i = 0; i < N; i++) {
        out[i] = a[i]*b[i];
    }
}
...
```

This code snippet shown above can be directly translated into the Assembly language implementation of Listing 1. The traditional `mul` instruction is used in this example. The `.ab` tag specifies the post-increment address write-back mode for the `ld` and `st` instructions.

For the equivalent implementation using dual-data memory banks (i.e. XY Memory), assume that the arrays `a`, `b` and `out` are already mapped to the XY memory address space through the AGU registers `%agu_r0`, `%agu_r1` and `%agu_r2`, respectively. Also consider that the corresponding AGU pointers get incremented by 4 bytes whenever `%agu_r0`, `%agu_r1` and `%agu_r2` are accessed. In such configuration, the dot-product can be implemented using XY memory as shown in the code snippet of Listing 2.

Listing 1: Plain dot-product algorithm with traditional load/store flow

```
# r0 <- a, r1 <- b, r2 <- out
    ...
loop_in:  # loop N times
    ld.ab %r3,[%r0,4]   # fetch a[i]
    ld.ab %r4,[%r1,4]   # fetch b[i]
    mul %r3,%r3,%r4
    st.ab %r3,[%r2,4]   # write c[i]
loop_end:
    ...
```

Listing 2: Plain dot-product algorithm with XY memory flow

```
# agu_r0 <- a, agu_r1 <- b, agu_r2 <- out
    ...
loop_in:  # loop N times
    # fetch a[i] and b[i], and
    # write c[i] at the same cycle
    mul %agu_r2,%agu_r0,%agu_r1
loop_end:
    ...
    ...
```

Given the comparison above, we clarify that the ARC XY Memory DSP subsystem is capable of performing two operand fetches and a write-back in a single instruction [39]. Such operation would require at least three instructions in a traditional load/store programming scheme. However, the execution of the aforementioned instruction in a single cycle depends primarily on the correct allocation of the input arrays to the appropriate memory banks: in Listing 2, by allocating the arrays `a` and `b` to distinct memory banks, we are able to fetch `a[i]` and `b[i]` elements in parallel within a single cycle. However, if the input arrays were programmed to the same memory bank, the array elements would only be sequentially accessible, i.e. in two cycles, thus reducing the performance by a factor of two. Therefore, the optimal usage of dual-data memory banks relies fundamentally on the proper allocation of data into the available memory banks.

It is also important to notice that implementations using XY memory inherit a code overhead regarding the initialization of the AGU registers and indexing modes. However, this overhead becomes negligible whenever looping over arrays of eight elements or more, due to the additional instructions required on the traditional load/store approach.

In this section we will demonstrate how our ISE for PRESENT and $\mathbb{F}_{2^{255}-19}$ can be combined with the ARC XY Memory DSP subsystem in order to achieve significant performance improvements. This requires the instruction operands be located in the correct X and Y memory banks. For the upcoming subsections, we consider that AGU registers labeled with `%agu_x` are designated to the X memory bank, whereas registers `%agu_y` are designated to the Y memory bank.

4.1 XY Memory Programming for the PRESENT Cipher

The main objective of this subsection is to outline our approach for leveraging the ARC XY Memory DSP Option in order to achieve IILP with the developed ISE for PRESENT. We begin by demonstrating how we converted the key schedule function implemented using traditional load/store flow into the equivalent XY memory programming scheme. For the code snippet of Listings 3 and 4, assume that the `%KEY0_AR`, `%KEY1_AR`, `%KEY2_AR` and `%KEY3_AR` AUX registers are already initialized with the 128-bit key value. Further, assume that the 64-bit values resulting from each key schedule round are consecutively stored in two distinct 32-bit arrays, namely `round_keys_h` and `round_keys_l`. As such, for a given round `i`, the 64-bit round key is given by the concatenation of `{round_keys_h[i],round_keys_l[i]}`. For the code on Listing 4, also assume that the `round_keys_l` and `round_keys_h` are assigned to the X and Y memory banks. This is done through the AGU registers `%agu_x0` and `%agu_y0`. Since the `keysch` instruction returns only the value that is written back to the `%KEY2_AR` AUX register, it is necessary to manually read the value from `%KEY3_AR` in order to capture the full round key value.

Listing 3: PRESENT key schedule with traditional load/store flow

```
# r0 <- round_keys_l
# r1 <- round_keys_h
    ...
loop_in:  # loop 31 times
    keysch %r2,%lp_count
    st.ab  %r2,[%r0,4]
    lr     %r2,[%KEY3_AR]
    st.ab  %r2,[%r1,4]
loop_end:
    ...
```

Listing 4: PRESENT key schedule with XY memory flow

```
# agu_x0 <- round_keys_l
# agu_y0 <- round_keys_h
    ...
loop_in: # loop 31 times
    keysch %agu_x0,%lp_count
    lr     %agu_y0,[%KEY3_AR]
loop_end:
    ...
    ...
    ...
```

By using the `keysch` instruction alongside the XY memory, it is possible to implement the key schedule algorithm's inner-loop rounds using 50% fewer instructions. For this particular case, the XY memory allocation is not critical, as there is no instruction which fetches two operands at the same time.

At this point, let us assume that the round key values are already computed and stored in the `round_keys_l` and `round_keys_h` arrays in the X and Y memory banks. The code snippets of Listings 5 and 6 show the implementation of the encryption function's main loop (i.e. encryption rounds) according to the load/store flow and the equivalent XY memory flow. When comparing both code snippets, the idea is to demonstrate that the encryption function's inner loop can be executed in a single cycle, with the `enc64` instruction fetching the two operands simultaneously. The final 64-bit encrypted message is stored in the AUX registers `%BLK0_AR` and `%BLK1_AR`, where it can be read using the `lr` instruction.

Listing 5: PRESENT block encryption with traditional load/store flow

```
# r0 <- round_keys_l
# r1 <- round_keys_h
    ...
loop_in: # loop 31 times
    ld.ab %r2,[%r0,4]
    ld.ab %r3,[%r1,4]
    enc64 0,%r3,%r2
loop_end:
    ...
```

Listing 6: PRESENT block encryption with XY memory flow

```
# agu_x <- round_keys_l in X memory bank
# agu_y <- round_keys_h in Y memory bank
    ...
loop_in: # loop 31 times
    enc64 0,%agu_y0,%agu_x0
loop_end:
    ...
    ...
    ...
```

The decryption function using the `dec64` instruction follows the same logic, as shown by the code snippets of Listings 14 and 15 in the appendix. The only difference here is that the final 64-bit decrypted message is stored in AUX registers `%IBLK0_AR` and `%IBLK1_AR`, where it can be read using the `lr` instruction. For the key schedule, encryption and decryption routines shown above, the utilization of the XY memory subsystem enables the implementation of the main inner-loops using 50% to 66% fewer instructions. This reflects in a performance improvement by a factor of two to three.

4.2 XY Memory Programming for $\mathbb{F}_{2^{255}-19}$ Arithmetic

Continuing with the ARC XY Memory DSP Option, in this subsection we outline the techniques for obtaining IILP with the ISE for $\mathbb{F}_{2^{255}-19}$ arithmetic. We start by showing how to translate the `fe` $\times$ `fe` full multiplication operation from the traditional load/store implementation to the XY memory approach. For the `fe` $\times$ `fe` operation in Listings 7 and 8, consider that the 256-bit operand `a` is already held by the eight `%FEi_AR` AUX registers, where `i = {0,...,7}`. The second operand is represented by the `b` array, and the 512-bit output is returned in two separate 256-bit arrays, `out_l` and `out_h`. They contain the least significant half and most significant half of the output, respectively. For the XY memory implementation in Listing 8, the `out_l` and `out_h` arrays are contained in different X and Y memory banks, such that they can be fetched within the same cycle.

Listing 7: `fe` × `fe` operation with traditional load/store flow

```
# FEi_AR <- a, r0 <- b
# r1 <- out_l, r2 <- out_h
    ...
# 1st loop: get out_l
loop_in:  # loop 8 times
    ld.ab %r3,[%r0,4]
    mword 0,%r3
    shacc %r3,0
    st.ab %r3,[%r1,4]
loop_end:
    ...
# 2nd loop: get out_h
loop2_in:  # loop 8 times
    shacc %r3,0
    st.ab %r3,[%r2,4]
loop2_end:
    ...
```

Listing 8: `fe` × `fe` operation with XY memory flow

```
# FEi_AR <- a, agu_x0 <- b
# agu_y0 <- out_l, agu_x1 <- out_h
    ...
# 1st loop: get out_l
loop_in: # loop 8 times
    mword 0,%agu_x0
    shacc %agu_y0,0
loop_end:
    ...
# 2nd loop: get out_h
loop2_in: # loop 8 times
    shacc %agu_x1,0
loop2_end:
    ...
    ...
    ...
    ...
```

Weak reduction and non-multiplicative operations: In order to allow intermediate results to fit within 256 bits (8 words), we perform a modular reduction to $2^{256} - 38$, exactly double the true field modulus of $2^{255} - 19$. This is done to improve efficiency: the reduction to $2^{256} - 38$ can be done using only addition with carry. Full reduction is done only once at the very end and requires some bit level manipulations, which is more costly timewise.

In addition to the `fe` × `fe` and weak reduction operations, we also require `fe + fe`, `fe - fe`, and full reduction. These operations are all implemented using regular ARC processor instructions. We describe these operations in appendix section 8.

5 Testing Methodology

Test vectors from both ISO/IEC 29192-2 standard and `lightweightcrypto.com` were used for verifying the intermediate and final outputs of our PRESENT implementation [17]. Meanwhile, test vectors from RFC7748 and RFC8032 were used for verifying our X25519 and EdDSA implementations, respectively [21, 18].

For our testbed, the following ARC EM9D processor specifications were chosen: 1) ARCv2EM core, 2) DCCM and ICCM banks of size 65K each, 3) X and Y Memory banks of size 8K each, and 4) medium-sized AGU controller. All time measurements and verification procedures are performed over the DesignWare ARC xCAM [38] cycle-accurate processor model tool. For the estimation of FPGA logic overhead introduced by the developed instructions' datapaths, we synthesized the ARC's RTL processor model (containing the developed ISE) targeting a Xilinx UltraScale XCZU9EG FPGA device [46].

DesignWare ARC xCAM: The DesignWare ARC xCAM [38] tool allows the generation of cycle-accurate ARC processor models that can be directly instantiated from the MetaWare [37] application development/debugging environment. xCAM generates cycle-accurate binary processor models by translating the Verilog-RTL system's specification into C++ or SystemC models [38]. Our tests and verification steps were performed over generated xCAM models including the developed APEX instructions and the XY Memory DSP core. For our cycle-count benchmarks, we used a 32-bit timer counter (*Timer0* component [36]) embedded in the ARC EM9D core.

FPGA Synthesis: On Xilinx FPGAs, the main resources for implementing general-purpose combinatorial and sequential circuits are the Configurable Logic Blocks (CLBs) [44]. The Xilinx XCZU9EG FPGA provides the following logic primitives: 1) 548,160 CLB Flip-Flops (FF), 2) 274,080 CLB LUTs, 3) 2,250 DSP48E2 DSP slices containing 27x18-bit multipliers [45] and 4) 34,260 CARRY8 fast dedicated lookahead carry logic for arithmetic carry chains. With the CARRY8 primitives (introduced in the UltraScale CLB architecture), carry logic can be leveraged to dedicated carry-lookahead units, reducing the number of used LUTs and improving routing/timing performance [44].

6 Results

6.1 PRESENT Cipher with ISE and XY Memory

Table 7 shows the benchmark and code size for three different PRESENT implementations on the ARC EM9D processor. The first one is a C-only implementation from [28] compiled with ARC's `CCAC` compiler using `-O3` [37]. The second one is an Assembly code implementation using the developed `enc64`, `keysch` and `dec64` APEX instructions. The final one is an assembly code implementation using the same APEX instructions along with XY Memory (Listings 4, 6 and 15).

Table 7: PRESENT with 128-bit key on ARC EM9D processor: timings in cycles and size of compiled code.

	Cycles			Code size
	Encryption	Key schedule	Decryption	(bytes)
Reis *et al.* [28], C-only	1,602	3,415	2,058	2,774
With APEX	161	331	161	362
With APEX & XY Memory	57	193	59	830

With the proposed ISE for PRESENT, a speedup by a factor of 9 to 12 was achieved, while reducing code size by a factor of 7.6. The combination of APEX instructions and XY Memory results in a speedup by a factor of 17 to 34. The introduction of XY Memory to the APEX ISE does, however, result in a code size increase from 362 to 830 bytes. Even with the increase in code size, it remains 70% smaller than the C-only software implementation.

Table 8 shows the FPGA synthesis results for the instruction datapath modules detailed in Table 3. The number of required logic resources for the synthesis of the AUX registers is shown next to the `aux_regs` label. The percentages show the increase in logic resources over a synthesized baseline ARC EM9D processor on the selected FPGA device (Xilinx XCZU9EG). Overall, the ISE for PRESENT requires 4.37% additional CLB LUTs and 8.26% extra CLB registers.

Table 8: Logic overhead introduced by PRESENT extension instructions on a synthesized ARC EM9D processor targeting the Xilinx XCZU9EG FPGA device

	`enc64`	`keysch`	`dec64`	`aux_regs`
CLB LUTs (Logic)	0.39%	0.04%	0.4%	3.53%
CLB Registers (FF)	-	-	-	8.26%

6.2 Curve25519 with ISE and XY Memory

Table 9 shows the benchmark results for the $\mathbb{F}_{2^{255}-19}$ field element operations implemented on the ARC EM9D processor. The `fe_mul_word`, `fe_mul` and `fe_sqr` operations are implemented in assembly using our `mword`, `shacc` and `rsacc` APEX instructions from Table 6 as well as the XY Memory programming flow introduced in the previous section.

Table 9: Timings in cycles and code sizes for $\mathbb{F}_{2^{255}-19}$ arithmetic functions on ARC EM9D processor. The cycle counts shown are taken as the average of 256 executions

`fe_`	`add`	`sub`	`mul2`	`mul_word`	`mul`	`sqr`	`inv`
Cycles	35	35	31	51	108	103	25,613
Code size (bytes)	122	122	108	154	354	338	354

The `fe_inv` (multiplicative inverse) is implemented using 11 `fe_mul` and 254 `fe_sqr` operations. We take advantage of the fact that many of the `fe_sqr` operations use the previous `fe_sqr` results for input. By keeping these intermediate results in AUX registers we are able to achieve roughly 6% savings over calling `fe_sqr` directly.

For the high level Curve25519 operations, we integrate our field arithmetic functions in a reimplementation of the software library from [13], along with modifications to support field elements optimally allocated in XY memory. Table 10 shows the benchmark results for the cryptographic Curve25519 functions implemented in C and compiled with ARC's `CCAC` compiler using `-O3`. To improve performance of the X25519 key exchange, we split the traditional X25519 operation into two separate steps: the public function is used to generate the public key, while the shared function generates the shared secret from the other party's public key. In the X25519 function's loop we need to multiply a field element by either 9 (in the case of public) or the other party's public key (in the case of shared). When multiplying by 9, `fe_mul_word` can be used instead of `fe_mul` which performs a multiplication of two field elements. The total computation time for each party during a key exchange is the sum of these two functions, which results in an additional 2% improvement.

In terms of code size, our `fe_mul` and `fe_sqr` implementations are roughly 45% smaller than [13]. Overall, our implementation of X25519 and EdDSA is about 5% smaller. It should be noted that the bulk of the code size of the X25519 and EdDSA functions shown in Table 10 are from the high level Curve25519 library and hash function.

Table 10: Timings in cycles and code sizes for Curve25519 cryptographic functions on ARC EM9D processor

	`x25519_public`	`x25519_shared`	`EdDSA_key_expansion`	`EdDSA_sign`	`EdDSA_verify`
Cycles	353,085	368,435	138,294	205,012	525,142
Code size (bytes)	3,486		20,768	21,076	25,130

Table 11 shows the FPGA synthesis results for the instruction datapath modules from Table 6, including the AUX registers. The percentages are the increase in logic resources over a baseline ARC EM9D processor on the selected FPGA device (Xilinx XCZU9EG).

Table 11: Logic overhead introduced by $\mathbb{F}_{2^{255}-19}$ extension instructions on a synthesized ARC EM9D processor targeting the Xilinx XCZU9EG FPGA device

	CLB LUTs (Logic)	CLB Registers (FF)	DSP Blocks	CARRY8
`mul_word`	2.75%	-	32[†]	104[†]
`adder_288`	1.32%	-	-	36[†]
`shift_reg`	1.18%	6.57%	-	-
`aux_regs`	3.88%	8.17%	-	-

Overall, the ISE for $\mathbb{F}_{2^{255}-19}$ arithmetic requires 9.1% additional CLB LUTs and 14.7% extra CLB registers. Additionally, a small number of DSP blocks and CARRY8 primitives are required for the synthesis and implementation of the `mul_word` and `adder_288` modules on the XCZU9EG FPGA. When comparing with the work from [13], speedups by a factor of around 2.5 (in terms of cycles) were achieved for the protocols over Curve25519.

[†]Showing here as absolute quantity (not relative)

6.3 SCMS/UBK

For the evaluation of a hardware accelerated UBK, we consider the customized ARC EM9D processor, which includes the specifications from Section 5 and the proposed ISE from Section 3. We assume a typical ASIC synthesis process targeting a 100MHz clock frequency [35]. Our FPGA synthesis results show that the ISE's datapaths are not part of the critical path.

In Table 12, we show timings for our implementation of a single EdDSA signature verification and for the UBK pseudonym certificate validations. Specifically, the table shows the portion of the UBK protocol that runs on the vehicle. The vehicle's UBK pseudonym certificate validation consists of EdDSA signature verification, ECIES decryption and ECC operations. These steps are shown in the last column of Table 2.

Table 12: Vehicle verification of UBK keys and signatures (considering $f_{clk} = 100$MHz)

	Single Signature Ver.	UBK Explicit	UBK Implicit	UBK Implicit Schnorr
Time (ms)	5.3	15	15.3	14.8

For the UBK pseudonym certificate validation, vehicles get their pseudonym certificates in advance, which gives the vehicle a window of time to verify them. Therefore, the performance consideration does not come with a requirement of low-latency or jitter typical of real-time scenarios. Instead the main interest is reducing the energy cost in the vehicle.

The first line of Table 12 shows the time for a single EdDSA signature verification, which is required for verifying Basic Safety Messages (BSM). Consider a typical scenario where a vehicle receives BSMs from 10 surrounding vehicles. Here the vehicle must be able to verify all message signatures within 100ms in order to avoid creating a backlog. This gives the vehicle up to 10ms to verify each message. Since our hardware accelerated implementation verifies a signature in 5.3ms, we surpass this requirement.

7 Conclusions

In this manuscript, we presented a novel hardware acceleration methodology that combines custom extension instructions with dual-data memory banks on an extensible processor platform. With the proposed methodology, the performance of PRESENT and Curve25519 were enhanced, while reducing code size and maintaining a low to moderate logic overhead. In addition, the proposed ISE facilitated constant-time cryptographic implementations.

Our hardware accelerated multiplicative functions for Curve25519 improved performance by a factor of ∼2.5 when compared to an optimized software implementation. FPGA logic overhead was moderate: 9.1% additional LUTs and 14.7% extra registers, as well as a small addition of DSP and CARRY8 blocks.

For PRESENT, we replaced the shifting and masking of the bit-sliced substitution and permutation operations with custom extension instructions composed of pure combinatorial blocks. This improved performance by a factor of 17 to 34 along with a 70% reduction in code size. Moreover, the additional FPGA logic overhead was as low as 4.37%.

Although we targeted the ARC processor, the proposed ISE can be adapted to any other extensible processor platform that supports custom extension instructions [7, 16, 29]. Furthermore, the code profiling, instruction datapath design and the XY Memory programming methods can also be extended to other cryptographic algorithms, including, but not limited to, other ECC schemes (e.g., Curve448), hash functions and MACs (Message Authentication Codes). Additionally, the hardware accelerated Curve25519 and PRESENT implementations can be used to address performance requirements for V2X applications. Specifically, our hardware accelerated functions meet the requirements for signature verification of BSMs in V2X applications. Lastly, the SCMS's UBK process can also take advantage of these hardware accelerated functions.

As a side contribution, we discuss a MitM attack when the RBK and UBK settings coexist, called as the co-existence attack. We show that this attack can be prevented by the addition of meta-data either in the PCA's certificate or the Vehicle pseudonym certificate. Both approaches lead to a negligible overhead in terms of bandwidth and processing time.

8 Acknowledgments

We thank Benedikt Brecht from VOLKSWAGEN Group of America, Inc. for the discovery of the co-existence attack. In addition, we thank S. Peter Song, Michael Frank, Guido Araujo, William Whyte, Virendra Kumar, Jonathan Petit and Abhishek Bit for their review and comments on this paper.

References

[1] Paulo S. L. M. Barreto, Marcos A. Simplicio Jr., Jefferson E. Ricardini, and Harsh Kupwade-Patil. Schnorr-based implicit certification: improving the security and efficiency of v2x communications. Cryptology ePrint Archive, Report 2019/157, 2019. https://eprint.iacr.org/2019/157.

[2] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[3] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards Curves. In *AFRICACRYPT*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer, 2008.

[4] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012.

[5] Daniel J. Bernstein and Tanja Lange. Analysis and optimization of elliptic-curve single-scalar multiplication. *Contemporary Mathematics – Finite Fields and Applications*, 461, 2008.

[6] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In *International Workshop on Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466. Springer, 2007.

[7] Cadence Design Systems. Tensilica Customizable Processors, 2019. Available at: https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable.

[8] CAMP. Security credential management system proof–of–concept implementation – EE requirements and specifications supporting SCMS software release 1.1. Technical report, Vehicle Safety Communications Consortium, may 2016.

[9] CAR 2 CAR Communication Consortium. Protection Profile V2X Hardware Security Module, 2018. Available at: https://www.car-2-car.org/fileadmin/documents/Basic_System_Profile/Release_1.3.0/C2CCC_PP_2056_HSM.pdf.

[10] Jeonghun Cho, Yunheung Paek, and David Whalley. Fast memory bank assignment for fixed-point digital signal processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 9(1):52–74, 2004.

[11] Jason Cong, Yiping Fan, Guoling Han, Ashok Jagannathan, Glenn Reinman, and Zhiru Zhang. Instruction set extension with shadow registers for configurable processors. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 99–106. ACM, 2005.

[12] ETSI. C-ITS platform, Working Group 5: Security & certification (final report, v1.0). Annex 1: Trust models for cooperative - intelligent transport system (C-ITS). Technical report, European Telecommunications Standards Institute, Aug 2016.

[13] H. Fujii and D. F. Aranha. Curve25519 for the cortex-m4 and beyond. In *Progress in Cryptology – LATINCRYPT 2017*, volume 35, pages 36–37, 2017.

[14] Philipp Grabher, Johann Großschädl, and Dan Page. Light-weight instruction set extensions for bit-sliced cryptography. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 331–345. Springer, 2008.

[15] IEEE. *IEEE Standard Specifications for Public-Key Cryptography – Amendment 1: Additional Techniques*. IEEE Computer Society, 2004.

[16] Intel Corporation. Nios II Processors, 2019. Available at: https://www.intel.com/content/www/us/en/products/programmable/processor/nios-ii.html.

[17] ISO, International Organization for Standardization. ISO/IEC 29192-2:2012 - Information technology - Security techniques - Lightweight cryptography. Accessed from: https://www.iso.org/standard/56552.html.

[18] S Josefsson and I Liusvaara. Rfc 8032–edwards-curve digital signature algorithm (eddsa). *Request for Comments, IETF*, 2017.

[19] M. Khodaei and P. Papadimitratos. The key to intelligent transportation: Identity and credential management in vehicular communication systems. *IEEE Veh. Technol. Mag.*, 10(4):63–69, Dec 2015.

[20] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2010.

[21] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security. Technical report, 2016.

[22] Peter L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[23] Alastair Murray and Björn Franke. Fast source-level data assignment to dual memory banks. In *Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pages 43–52. ACM, 2008.

[24] NHTSA. Federal Motor Vehicle Safety Standards; V2V Communication. Technical Report 8, National Highway Traffic Safety Administration, U.S. Department of Transportation (USDOT), Jan 2017.

[25] NIST. *FIPS 197: Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, November 2001.

[26] NIST. *FIPS 186-4: Digital Signature Standard (DSS)*. National Institute of Standards and Technology, July 2013.

[27] EVITA Project. E-safety vehicle intrusion protected applications, 2008. European Commission research grant FP7-ICT-224275, https://www.evita-project.org/.

[28] T. B. S. Reis, D. F.Aranha, and J. López. PRESENT runs fast - efficient and secure implementation in software. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 644–664, Cham, 2017. Springer International Publishing.

[29] RISC-V Foundation. RISC-V: The Free and Open RISC Instruction Set Architecture, 2019. Available at: https://riscv.org/.

[30] Mazen AR Saghir, Paul Chow, and Corinna G Lee. Exploiting dual data-memory banks in digital signal processors. *ACM SIGPLAN Notices*, 31(9):234–243, 1996.

[31] Pascal Sasdrich and Tim Güneysu. Implementing curve25519 for side-channel–protected elliptic curve cryptography. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 9(1):3, 2015.

[32] M. A. Simplicio, E. L. Cominetti, H. Kupwade Patil, J. E. Ricardini, and M. V. M. Silva. The unified butterfly effect: Efficient security credential management system for vehicular communications. In *2018 IEEE Vehicular Networking Conference (VNC)*, pages 1–8. IEEE, Dec 2018.

[33] Synopsys, Inc. DesignWare ARC EM APEX Databook - Version 5768-004, 2014.

[34] Synopsys, Inc. DSP Options for ARCv1 Cores, 2016. Available at: https://www.synopsys.com/dw/doc.php/ds/cc/arc_xy_adv_dsp.pdf.

[35] Synopsys, Inc. Development Platforms, 2018. Available at: https://embarc.org/platforms.html.

[36] Synopsys, Inc. DesignWare ARC EM Processor Family, 2019. Available at: https://www.synopsys.com/designware-ip/processor-solutions/arc-processors/arc-em-family.html#extensibility.

[37] Synopsys, Inc. DesignWare ARC MetaWare Development Toolkit, 2019. Available at: https://www.synopsys.com/dw/ipdir.php?ds=sw_metaware.

[38] Synopsys, Inc. DesignWare ARC xCAM, 2019. Available at: https://www.synopsys.com/dw/ipdir.php?ds=sim_xcam.

[39] Synopsys, Inc. Performance and Coding Advantages with the ARC XY Memory DSP Option, 2019. Available at: https://www.synopsys.com/designware-ip/technical-bulletin/performance-coding-advantages.html.

[40] JJ Tay, MLD Wong, MM Wong, C Zhang, and I Hijazin. Compact fpga implementation of present with boolean s-box. In *Quality Electronic Design (ASQED), 2015 6th Asia Symposium on*, pages 144–148. IEEE, 2015.

[41] Trusted Computing Group (TCG). Tpm specification 1.2 revision 116, 2011. www.trustedcomputinggroup.org.

[42] Stefan Tillich and Johann Großschädl. Instruction set extensions for efficient aes implementation on 32-bit processors. In *International workshop on cryptographic hardware and embedded systems*, pages 270–284. Springer, 2006.

[43] W. Whyte, A. Weimerskirch, V. Kumar, and T. Hehn. A security credential management system for V2V communications. In *IEEE Vehicular Networking Conference (VNC'13)*, pages 1–8, 2013.

[44] Xilinx, Inc. UltraScale Architecture Configurable Logic Block, 2017. Available at: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.

[45] Xilinx, Inc. UltraScale Architecture DSP Slice, 2018. Available at: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.

[46] Xilinx, Inc. Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit, 2019. Available at: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[47] Pan Yu and Tulika Mitra. Disjoint pattern enumeration for custom instructions identification. In *2007 International Conference on Field Programmable Logic and Applications*, pages 273–278. IEEE, 2007.

Appendix

Listing 9: Verilog implementation of the `sbox` module for `enc64` and `keysch` instructions

```verilog
input in3, in2, in1, in0;      // in3 is the most significant bit
output out3, out2, out1, out0; // out3 is the most significant bit

assign out3 = (~in3 & ~in1 & ~in0) | (~in3 & in1 & in0) |
              (in3 & ~in2 & in0) | (in3 & ~in2 & in1) |
              (~in3 & in2 & in1);
assign out2 = (~in2 & in1 & ~in0) | (in3 & in2 & ~in1) |
              (~in2 & ~in1 & in0) | (~in3 & ~in2 & ~in1) |
              (~in3 & in2 & in1 & in0);
assign out1 = (in3 & in2 & in0) | (in3 & ~in2 & ~in1) |
              (~in3 & in1 & ~in0) | (~in3 & ~in2 & in1) |
              (in3 & ~in2 & ~in0);
assign out0 = (~in3 & in1 & in0) | (in3 & in1 & ~in0) |
              (in3 & ~in2 & ~in0) | (~in3 & ~in2 & in0) |
              (~in3 & in2 & ~in1 & ~in0) | (in3 & in2 & ~in1 & in0);
```

Listing 10: Verilog implementation of the `pLayer` module for the `enc64` instruction

```verilog
input[63:0] in; output[63:0] out;

genvar i;
generate
  for (i=0; i<64; i=i+4)
  begin : g
    assign out[i/4] = in[i];
    assign out[i/4 + 16] = in[i+1];
    assign out[i/4 + 32] = in[i+2];
    assign out[i/4 + 48] = in[i+3];
  end
endgenerate
```

Listing 11: Verilog implementation of the `leftRot_61` module for the `keysch` instruction

```verilog
input[127:0] in; output[127:0] out;

genvar i;
generate
  for (i=127; i>=0; i=i-1)
  begin: g
    if (i-61>=0)
      assign out[i] = in[i-61];
    else
      assign out[i] = in[i+67];
  end
endgenerate
```

Listing 12: Verilog implementation of `isbox` module for `dec64` instruction

```
input in3, in2, in1, in0;      // in3 is the most significant bit
output out3, out2, out1, out0; // out3 is the most significant bit

assign out3 = (~in3 & ~in2 & in0) | (~in3 & ~in2 & in1) |
              (~in3 & in1 & in0) | (in3 & in2 & in1) |
              (~in3 & in2 & ~in1 & ~in0) | (in3 & ~in2 & ~in1 & ~in0);
assign out2 = (~in3 & ~in2 & ~in1) | (~in3 & ~in1 & ~in0) |
              (~in2 & in1 & ~in0) | (in3 & ~in1 & in0) |
              (~in3 & in2 & in1 & in0);
assign out1 = (~in3 & in1 & ~in0) | (in3 & ~in2 & ~in0) |
              (in3 & ~in2 & in1) | (in3 & in2 & in0) |
              (~in3 & ~in2 & ~in1 & in0);
assign out0 = (~in3 & ~in2 & ~in0) | (~in2 & ~in1 & ~in0) |
              (~in3 & in2 & in0) | (in2 & ~in1 & in0) |
              (in3 & ~in2 & in1 & in0) | (in3 & in2 & in1 & ~in0);
```

Listing 13: Verilog implementation of the `ipLayer` module for `dec64` instruction

```
input[127:0] in; output[127:0] out;

genvar i;
generate
  for (i=0; i<16; i=i+1)
  begin : g
    assign out[4*i]     = in[i];
    assign out[4*i + 1] = in[i + 16];
    assign out[4*i + 2] = in[i + 32];
    assign out[4*i + 3] = in[i + 48];
  end
endgenerate
```

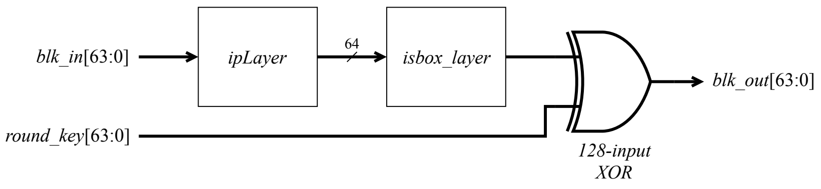

Figure 9: Datapath of proposed `dec64` instruction for PRESENT decryption

Listing 14: PRESENT block decryption with traditional load/store flow

```
# r0 <- round_keys_l
# r1 <- round_keys_h
    ...
loop_in:  # loop 31 times
    ld.ab %r2,[%r0,4]
    ld.ab %r3,[%r1,4]
    dec64 0,%r3,%r2
loop_end:
    ...
```

Listing 15: PRESENT block decryption with XY memory flow

```
# agu_x <- round_keys_l
# agu_y <- round_keys_h
    ...
loop_in:  # loop 31 times
    dec64 0,%agu_x0,%agu_y0
loop_end:
    ...
    ...
    ...
```

Non-multiplicative $\mathbb{F}_{2^{255}-19}$ Operations

The non-multiplicative $\mathbb{F}_{2^{255}-19}$ operations are implemented using standard ARC instructions along with the AGU to efficiently load and store the field elements from memory. The basic method of implementation for these operations is as follows: 1) Set up the AGU to read in the field elements one word at a time. 2) Set up the AGU to write the result. 3) Perform the operation, storing the intermediate results in regular processor registers. 4)

Perform weak reduction on the intermediate results and write the reduced output to XY memory. (The main benefit of storing the intermediate results in regular CPU registers is that it saves cycles by not having to set another AGU base address register, which must be done using the `sr` instruction.)

Further details on weak reduction: The weak reduction itself is easily explained: assume we have a 288 bit result in an array of nine words `P[8],...,P[0]` where `P[8]` is the most significant. Multiply `P[8]` by 38 and add it to `P[7,...,0]`. Since the field elements are 256-bits long, addition, subtraction and multiplication by two will only carry at most one bit to `P[8]`. Which means if `P[8] == 1`, then add 38 to `P[7,...,0]`. Although simple, there are two corner cases to consider when `P[7..0] >=` $2^{256} - 38$:

If `P[8] == 0`, then we would not add 38 and simply take the result in `P[7..0]` which would not be fully reduced `mod` $2^{256} - 38$. This can be safely ignored, as all of our field arithmetic operations are designed to work with 256-bit inputs. If `P[8] == 1`, then adding 38 to `P[7..0]` will result in `P[7..1]` being 0 and `P[0]` being a very small value. In this case, it is necessary to add an additional 38 in order to achieve the correct reduction. As the weak reduction is an integral part of the field operation, it makes more sense to discuss the details of its implementation in that context. This is done below for the `fe_add` operation.

`fe_add`: The `fe_add` operation takes two 256-bit inputs in XY memory and stores the resulting sum to XY memory reduced to $2^{256} - 38$. For this, we make use of the ARC processor's add with carry instruction:

```
adc a,b,c  # (a = b + c + carry)
```

In Listing 16, `P0 - P8` are arbitrary processor registers that correspond with the intermediate 288 bit results that become the input to the weak reduction.

The addition of the first word uses `add` since there is no previous operation that needs to be carried. The "`.f`" indicates that the carry flag should be set if the results of the addition overflows the 32-bit destination. The subsequent additions use `adc` which will then add the value of the previous operation's carry flag to the result. Technically, the most significant word `P[8]` should be 1 in the case of the final `adc` producing a carry, or 0 otherwise. This could be done easily using `adc P8,0,0`. However, in preparation for the weak reduction it is much more desirable to have 38 in the case of a carry as this is the value that needs to be added. This is accomplished using a conditional move instruction (`mov.c`): In the case of a carry, `P8` will be set to 38, otherwise it will contain the previously set value of 0.

Listing 16: `fe + fe` operation with XY memory

```
# agu_x0 <- first field element in X bank
# agu_y0 <- second field element in Y bank
    ...
    mov   P8,0
    add.f P0,%agu_x0,%agu_y0
    adc.f P1,%agu_x0,%agu_y0
    adc.f P2,%agu_x0,%agu_y0
    adc.f P3,%agu_x0,%agu_y0
    adc.f P4,%agu_x0,%agu_y0
    adc.f P5,%agu_x0,%agu_y0
    adc.f P6,%agu_x0,%agu_y0
    adc.f P7,%agu_x0,%agu_y0
    mov.c P8,38
    ...
```

The code in Listing 17 immediately follows the previous addition and is a straightforward implementation of the weak reduction already discussed: `P8` will contain either 0 or 38 which is added to the least significant word of the intermediate sum in P0. The following add with carry operations will propagate any additional carry as well as write the result to the destination XY memory. The `jcc` instruction is a conditional jump instruction (in this case, jump if carry not set): If the final adc did not generate a carry, then we are done and will return to the caller. If carry was set, then an additional 38 is added to `P[0]` and written to the output. As this case is very unlikely to ever occur in actual operation, we are not concerned about efficiency.

Listing 17: Weak reduction operation after addition

```
# agu_xy0 and r1 <- destination field element in either X or Y bank
    ...
    add.f %agu_xy0,P0,P8
    adc.f %agu_xy0,P1,0
    adc.f %agu_xy0,P2,0
    adc.f %agu_xy0,P3,0
    adc.f %agu_xy0,P4,0
    adc.f %agu_xy0,P5,0
    adc.f %agu_xy0,P6,0
    adc.f %agu_xy0,P7,0
    jcc   [%blink]
    # Handle the corner case when final carry overflows and 38 needs
    # to be added again
    ld_s  %r1,[%r0,0]
    add   %r1,%r1,38
    j_s.d [%blink]
    st_s  %r1,[%r0,0]
    ...
```

`fe_sub`: The `fe_sub` operation takes two 256-bit inputs in XY memory and stores the resulting difference to XY memory reduced to $2^{256} - 38$. For this, we make use of the ARC processor's subtract with carry instruction:

```
sbc a,b,c   # (a = b - c - carry)
```

The implementation follows the same pattern as `fe_add`, except that all of the `add/adc` become `sub/sbc`.

`fe_mul2`: Several times in the high level elliptic curve functions we need to do a field multiplication by two. Using the existing field multiplication by a digit function (`fe_mul_word`) for this is inefficient. Using the `fe_add` function to add the field element to itself would accomplish the task. However, this presents a problem when using the AGU to read the same value twice from the same memory bank, since this introduces an extra cycle of latency for every read. Alternatively, the rotate left through carry instruction allows for an efficient implementation of multiplication by two using a single input operand:

```
rlc b,c # (b = c << 1; b = b OR carry)
```

For the first word's left shift the `asl` instruction is used since there is no initial carry. For subsequent shifts `rlc` is used. The weak reduction is identical to that in `fe_add`.

`fe_rdc`: As previously discussed, the weak reduction that is done at the end of each field operation reduces the result modulo $2^{256} - 38$ in order to keep the intermediate results within 256 bits. As the final step in a series of calculations, a final modular reduction to $2^{255} - 19$ needs to be done. For this, we look at bit 256 of the intermediate output: If the bit is set, clear it and add 19 to the result. This code snippet is shown in listing 18

As we are only looking at bit 256, there is the corner case where the input value before reduction is in the range of $2^{255} - 1$ and $2^{255} - 19$. This would correspond with an elliptic curve point of 0-18.

Listing 18: Full reduction operation

```
# agu_x0 <- input field element in X bank (r0)
# agu_x1 <- output field element in X bank (r1)
    ...
    ld    %r0,[%r0,28]
    asr   %r1,%r0,31
    and   %r1,%r1,19  #r1 = (a[7] >> 31) * 19
    bclr  %r0,%r0,31  #r0 = a[7] & 0x7fffffff

    add.f %agu_x1,%agu_u1,%r1
    adc.f %agu_x1,%agu_x0,0
    adc.f %agu_x1,%agu_x0,0
    adc.f %agu_x1,%agu_x0,0
    adc.f %agu_x1,%agu_x0,0
    adc.f %agu_x1,%agu_x0,0
    adc.f %agu_x1,%agu_x0,0
    j_s.d [%blink]
    adc   %agu_x1,%r0,0
    ...
```