Falcon — A Flexible Architecture For Accelerating Cryptography

Kevin Kiningham, Philip Levis, Mark Anderson, Dan Boneh, Mark Horowitz, Maurice Shih
Stanford University
{kkiningh, pal, mark01, dabo, horowitz, maurices}@stanford.edu

***Abstract*—Internet of Things (IoT) devices, once deployed, must remain secure for their entire lifetime, which can be as long as 20 years. Over this lifetime, devices must be able to update which ciphers they use to meet evolving security requirements. However, devices cannot rely on software updates for their cryptography because software implementations consume too much energy. At the same time, fixed function hardware accelerators such as an AES engine cannot support new ciphers.**

This paper presents Falcon, a hardware architecture for accelerating a broad range of cryptography on energy limited devices. Rather than accelerate a fixed set of current ciphers, Falcon provides a general execution engine that accelerates dominant and emerging ciphers, such as AES, Cha-Cha, SHA-256, RSA, ECC with Curve25519, as well as post-quantum ciphers such as R-LWE. For cryptography, Falcon provides the flexibility of software while reducing the energy consumption of cryptography by 5-60x compared to software. This reduction makes it feasible for IoT applications to upgrade the ciphers they use after deployment, allowing them to keep up to date with security best practices without reducing their deployment lifetime or reducing the application workload. In an application monitoring the temperature of sensitive medical supplies in hospitals, Falcon doubles the deployment lifetime (2.2x).

***Index Terms*—Security, Cryptography, Accelerator architectures, Internet of Things**

I. Introduction

The term "Internet of Things" (IoT) describes a broad range of technologies and applications that incorporate powerful computational, sensing, and actuation capabilities into physical objects. Examples include smart thermostats, door locks, industrial machinery, automobiles, light bulbs, and watches. Networking these devices provides new, valuable services and allows them to perform better. These benefits, however, are limited by the fact that the security of IoT devices is notoriously poor; security flaws in IoT devices have been used to take control of vehicles [1] and take down key parts of the Internet infrastructure [2].

Cryptography is a critical component of securing IoT applications. Cryptography, however, is typically highly compute intensive, which poses a problem for energy limited IoT devices. To make cryptography energy-efficient enough to be practical, many embedded microcontrollers for IoT devices include dedicated cryptographic accelerators. These accelerators typically support particular cryptosystems, such as AES-128 [3] or SHA-256 [4]. Some generalize slightly, by accelerating lower-level primitives, such as modular arithmetic or operating on Galois fields [5]. These dedicated accelerators can reduce the energy costs of cryptography by two orders of magnitude, making it practical, and are commonly used on a wide variety of commercially available devices.

However, fixed function accelerators cannot be easily adapted to new ciphers. While this may be acceptable for applications with short lifetimes or fixed security requirements, many IoT devices deployed today may remain in operation for 10-20 years. On a multi-decade timescale, the changes in security standards, laws, and application requirements can be immense; 20 years ago, AES, the dominant symmetric cipher used today, had only recently been published and standardized. This leaves IoT applications with a difficult choice: rely on hardware accelerated ciphers for energy efficiency and fall behind in security best practices, or keep up to date and reduce the application workload due to increased energy costs.

To resolve this tension, we propose Falcon, a *flexible* accelerator designed to reduce energy consumption for both current and future cryptosystems. Falcon allows software like programmability while still reducing the energy cost of cryptosystems by a factor of 5-60x compared to a software only implementations. This cost reduction makes it possible to dynamically upgrade crypto after deployment, improving security and allowing applications to be deployed for significantly longer. In one application, Falcon doubles the deployment lifetime.

A. Contributions

This paper makes the following three contributions:

- An analysis of existing and emerging cryptosystems that identifies common primitives (Section III)
- The design and implementation of a novel cryptographic accelerator designed to both directly accelerate common primitives as well as exploit parallelism using a SIMD design (Sections IV and V)
- The evaluation of a range of cryptosystems on a fully synthesized implementation of Falcon, considering area and energy, as well as demonstrating Falcon's advantages on the design of an application (Section VI)

II. Motivating Application

We introduce the following IoT application to motivate the design of Falcon. Hospitals must constantly monitor refrigeration units containing sensitive medical supplies, such as vaccines, to ensure they are stored at the correct temperature [6]. This is typically performed by an employee who visits each unit, reads an attached monitoring device, and records the

minimum and maximum observed temperature in a log book. These checks occur frequently (at least daily) to ensure that the supplies do not spoil.

However, monitoring a large number of units involves both a significant amount of physical labor and the potential for human error. To address this, battery powered IoT devices have been proposed. The devices record the temperature every 10 minutes, storing it in flash memory. The data is collected by a hospital employee who walks near each unit and automatically downloads the data over BLE using a mobile phone.

Initially, the connection between the IoT monitoring devices and the employee's phone is authenticated using a pre-shared secret and symmetric encryption (AES-128). After the initial deployment, the project switches to using contractors who must authenticate and pair themselves with the device each time they download data. As part of the pairing process, the devices use an elliptical curve key-exchange algorithm (ECDHE). Many years later, the project switches to R-LWE, a quantum resistant algorithm, after significant advances in quantum computers undermine the security of ECDHE.

We estimated the energy consumption of this application running on a Nordic nRF51822, a common microcontroller for BLE-enabled IoT devices. Table I summarizes our results, showing the energy required per day.

TABLE I: Increase in energy consumption over baseline in model IoT application.

Application	**Energy** (μJ)	**Increase**
Base Application	3100	1.0x
With ECDHE	5400	1.7x
With R-LWE	9300	3.0x

The key result of this analysis is that the changes in security requirements mean a significant increase in the total energy usage of the application. Although the nRF51822 does include a cryptographic accelerator, it only supports a single cipher, AES-128. This means that while initially cryptography was a small component of the overall energy budget, the total energy usage of the application must increase substantially after additional crypto is added. Since the hospital must collect the data each day, the updated devices must be replaced much sooner than the initial design (in this case, $1/3$rd the original lifetime.)

The goal of this research is to address this issue by designing an accelerator that allows for upgrading the ciphers used in an application, while still maintaining the efficiency of dedicated hardware.

III. Cryptographic Commonalities

In this section, we examine many different cryptographic algorithms used in IoT applications to identify the common primitives that may be accelerated. We divide these algorithms into three categories:

1) Symmetric ciphers: The workhorses of secure communication, used to ensure data confidentiality and/or integrity. Examples include AES and ChaCha.
2) Hash functions: Used to verify that data has not been tampered with, and when combined with ciphers can authenticate that the sender has a certain key. Examples include SHA256.
3) Asymmetric cryptography: Most commonly used in IoT applications for cryptographic signatures and key agreement. Asymmetric ciphers are very computationally expensive compared to symmetric ciphers or hash functions, and so are typically used sparingly.

For this work, we analyzed 38 different cryptographic algorithms, including 27 symmetric ciphers, 7 hash functions, and 4 asymmetric operations drawn from cryptographic design competitions, widely used protocols, and emerging use cases. Table II breaks down the percentage of the important operations we found. While symmetric ciphers and hash functions have a relatively wide range of operations, the dominate operations are "bit-and-byte" operations such as shifting, rotating, or permutations. On the other hand, the asymmetric operations we studied only needed modular multiplication and addition.

TABLE II: Percentage breakdown of operations found in different cryptographic algorithms.

	Symmetric			
	Block	**Stream**	**Hash**	**Asymmetric**
Multiply	17%	0%	0%	100%
Add/Sub.	61%	67%	71%	100%
Bit Shift	91%	100%	86%	0%
Rotate	41%	56%	100%	0%
Permute	44%	78%	100%	0%
Table Lookup	72%	22%	28%	0%

The bitwidth of operations also varies significantly between ciphers. Table III shows the percentage of ciphers that had operations of a particular bitwidth. For symmetric ciphers and hash functions, operations tend to use 64 or fewer bits, with the majority using 32 or fewer. This can be attributed to the fact that many of these algorithms were designed to be efficient on machines with 32-bit datapaths. Asymmetric ciphers, on the other hand, use much larger bit widths. This is due to the fact that the mathematical foundations of these algorithms are based on modular arithmetic.

TABLE III: Percentage of ciphers studied that have operations of a particular bitwidth

	Symmetric			
Bitwidth	**Block**	**Stream**	**Hash**	**Asymmetric**
≤ 16	56%	11%	14%	0%
32	72%	56%	71%	0%
64	6%	0%	28%	0%
> 64	0%	0%	0%	100%

The results of this analysis make clear that providing acceleration on the same hardware across all operations is challenging. Asymmetric ciphers in particular do not share many of the of operations that are used by symmetric ciphers and hash functions. Symmetric ciphers use a lot of wide

modular arithmetic, while symmetric operations use mostly narrow "bit-and-byte" operations. Thus, it is important that our design efficiently supports multiple datapath sizes, especially for arithmetic operations.

IV. FALCON DESIGN

A. Design Guidelines

Motivated by the cryptographic commonalities described in Section III and previous analysis [7], Falcon has four key design goals to increase the energy efficiency of a wide range of cryptosystems.

No data-dependent control flow: An accepted requirement for cryptosystems today is that they have no data-dependent branches, as these branches expose timing differences that can leak information. As a result, the architecture should support only fixed branches. Algorithms that do have data-dependent branches can be implemented with a combination of masking and redundant computation.

Wide, but flexible, data widths: Most cryptosystems use "wide" operations that operate on many bytes. For example, in the AES symmetric cipher the wide operation is a single function applied in parallel to each byte of state. In asymmetric systems such as RSA, DSA, and ECC, the wide operations are long-word modular exponentiation and multiplication. The width varies across different ciphers and even in different versions of the same cipher. For example, the size of the long-word operations needed by elliptic curves for commonly used curves can range anywhere from 163-bits to over 500 bits depending on exactly which field and curve is used. The architecture therefore needs to provide a wide, but width-configurable, data path.

Support for bit and byte operations: Symmetric ciphers, because they depend on the mixing of bits through substitutions and permutations, rely heavily on bit-level and byte-level operations. While they operate on wide (e.g., 128 or 256 bits) units of data, the operations on these wide units are not arithmetic in nature. In particular, permutations involve shifts and masks.

Small instruction size: Because many cryptosystems execute many instructions on relatively small units of data, instruction fetches can dominate data movement. This is especially true in microcontrollers, which typically do not have caches and so directly fetch instructions from RAM or flash. For example, a software implementation of RSA on a RISC-V microprocessor fetches hundreds of thousands of instructions. These fetches consume a lot of energy so need to be minimized.

B. ISA Overview

Falcon is a 256-bit wide, in-order SIMD machine with 256-bit registers and support for lane masking. Falcon's SIMD instructions support a large number of data widths; 8, 16, 32, 64, 128, and 256 bits. Additionally, it has a permutation and bitslice instruction useful for implementing the bit and byte-wise operations common in many symmetric ciphers and hash algorithms. Finally, Falcon's instruction set uses a short, simple encoding to maximize density.

C. SIMD Operation

Every Falcon instruction is SIMD: it executes an identical operation on data packed in one of 16 256-bit registers. The input and output sizes of this operation are the *lane width*. The lane width is not set on a per instruction basis, but instead defined in a global register set by the `SET_WIDTH` instruction. The `SET_WIDTH` instruction can be executed at any point, which means the lane width can change during program execution. This saves encoding space since instructions do not have to encode the width of their operands.

The lane width determines the total number of lanes, which is 256 divided by the current lane width. This means Falcon can, with a single instruction, either execute 32 parallel operations on 8-bit lanes or a single operation on a 256-bit lane.

SIMD instructions can be prevented from executing for a particular lane by setting a lane's mask. Lane masks are set using the `SET_MASK` instruction, which sets a corresponding bit in a global 32-bit register to the least significant bit of each lane. The operation of a masked lane depends on the current lane width; if the lane width is larger than 8 bits, only the least significant bit of the mask for that lane is used. This allows simple conditional execution, albeit with the restriction that instructions on both branches are fetched.

D. Instruction Encoding

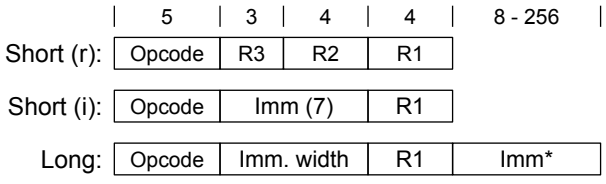


Fig. 1: Instruction encodings

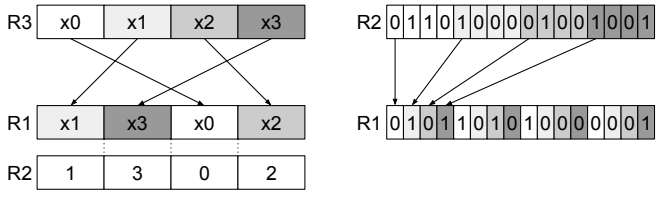
For almost all instructions, Falcon minimizes code size with a short, 16-bit encoding. In immediate form, the short encoding allows for a single destination register and a 7-bit immediate. In register form, the encoding allows one destination register (R1) and two operand registers (R2 and R3). Falcon has 16 registers. R3 has only 3 bits for its encoding, R3's encoding is extended with the most significant bit of R1. This allows for easy translation to a 2-operand form by simply replacing R3 with the least significant 3 bits of R1.

Falcon also supports a "long" encoding that supports a single destination register and a variable length immediate. This encoding is used exclusively for the load-immediate (`LDi`) instruction, which allows immediate values of up to 256-bits long. While most ciphers do not require arbitrary memory lookups (with the exception of S-Box lookups), many require special constants. Furthermore, specialized operations such as permutations require long constants to describe the permutation. The `LDi` instruction supports these operations

without requiring additional memory lookups by directly embedding the values in the instruction stream.

To reduce the number of long load immediate instructions, load behavior depends on the current lane width. If the load is smaller than the lane width then the immediate load is performed across all lanes and masked to its size. For example, an 8-bit immediate load executed when the lane width is 32 bits sets the bottom 8-bits of all lanes to the immediate value. If the load is wider than the lane then the immediate is repeated to fill 256 bits and then broken into chunks of the lane width. For example, if the lane width is 32 bits and the load is 128 bits, then the 128 bits is repeated twice. The resulting 256 bits is then split into 8 32-bit chunks, with one assigned to each lane. Lane 0 and lane 4 will have the same chunk loaded.

E. Lane Permutation and Bitslicing



(a) Lane Permutation (b) Bitslice

Fig. 2: Execution of a 4-way Permutation and Bitslice operation. Each color represents a different lane

Permutation operations are a common mechanism in symmetric ciphers and hash functions. Rather than rely on lookup tables or multiple bit operations, Falcon accelerates permutations with a `PERMUTE` instruction that allows for arbitrary permutation of values between lanes. The `PERMUTE` instruction takes three registers as arguments: a destination register (R1), a shuffle register (R2), and a data register (R3). The value from each lane in R3 is stored in the R1 of the lane specified in R2. Masked lanes do not change their value. Figure 2a shows an example of a 4 lane permutation.

Bitslicing is a technique first described for software DES and used in many cipher implementations [8]. Bitslicing reduces computation to elementary logic operations on the individual bits of the input, rather than on entire words. These logic operations are executed across every word of the input in parallel, exposing additional parallelism. Bitslicing can also eliminate table look-ups [9], which is beneficial because they frequently lead to exploitable side channels.

Falcon also provides a `BITSLICE` instruction to accelerate these operations. The `BITSLICE` instruction takes two registers as arguments: a destination register (R1) and a data register (R2). If the current lane width is W, the bits of R1 are permuted such that the ith bit of lane j is swapped with jth bit of lane i when the lane width is $256/W$. Figure 2b shows an example of a 4 lane bitslice operation. If the lane being swapped is masked, the corresponding bit is set to zero. If no masking is present, the `BITSLICE` operation is its own inverse.

```
SET_WIDTH 32
BITSLICE R0, R0
LDi      R1, \
 0x10071415 0x1D0C1C11 \
 0x010F171A 0x05121F0A \
 0x0208180E 0x201B0309 \
 0x130D1E06 0x160B0419
SET_WIDTH 8
PERMUTE  R0, R1, R0
BITSLICE R0, R0
```

```
SET_WDITH 32
LDi R1, 1   // Bit 1
SRi R1, 15
AND R1, R1, R0
SLi R1, 15
OR  R2, R2, R1
LDi R1, 1 // Bit 2
SRi R1, 6
AND R1, R1, R0
SLi R1, 5
OR  R2, R2, R1
... // 30 more
```

Fig. 3: Falcon assembly implementations for an 8-way parallel 32-bit half block DES permutation. Taking advantage of the `PERMUTATION` and `BITSLICE` instructions (left) results in 3.8% the number of instructions, 7.5% as many cycles, and 39.8% the number of bytes when compared to using a mask and shift approach (right).

The permutation and bitslicing instructions can be combined to support complex bit operations without the overhead of masking and extracting individual lane values. For example, the DES permutation operation executes a complicated permutation of 32 bits every round. Since this operation is performed repeatedly, its execution time is important for the overall performance of the algorithm. In listing 3 we show the difference in code required to support the DES permutation using bitslicing and permutations, compared to the using a more traditional masking approach. Together, they cut the code size of a 32-bit half block DES permutation by 96.2%, and the number of instructions issued by 92.5%.

V. Hardware Implementation

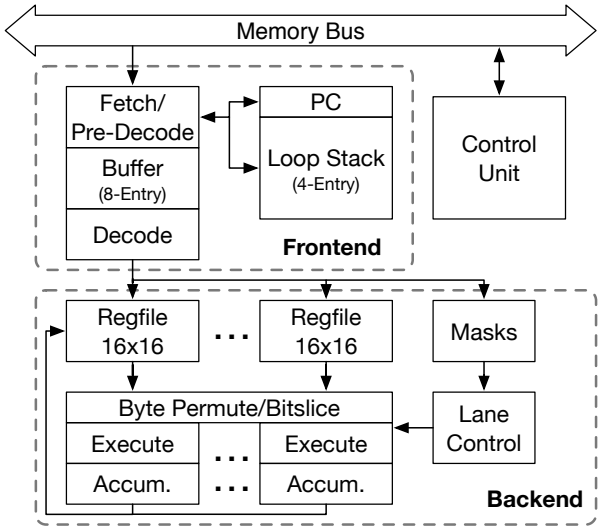


Fig. 4: Architectural overview of Falcon.

Our implementation of Falcon is designed to be run as an independent co-processor, communicating over a bus with a main processor that issues high level commands. It is

completely in-order, with no register renaming or instruction level parallelism. A hardware overview of Falcon is shown in figure 4.

A. System Interface

As is common for many cryptographic accelerators [10], Falcon operates as a co-processor that communicates with a host processor using memory mapped registers. To execute a Falcon program, the host processor first writes instructions to memory. It then passes Falcon a pointer to the program by writing it to a specific memory mapped register. Falcon then resets all internal state, checks the validity of the pointer, and starts executing instructions from the provided address.

To communicate data during the execution, Falcon's input and output data buffers are associated with a different set of memory mapped registers. Writes to these registers are buffered using a FIFO that can be read by Falcon during execution. If the FIFO is empty, Falcon will block execution until data is available. Similarly, writes by Falcon are also buffered, and can be read later by the host or peripherals attached to the main memory bus. This allows Falcon to support streaming operations, which are commonly used in many IoT applications.

Falcon can also be configured to use a scratchpad memory, intended for storing cryptographic secrets such as certificates or private keys. This is special, possibly non-volatile, storage that is inaccessible to the host, but readable by Falcon. For added isolation, the scratchpad memory is able to use a memory bus that is kept physically separate from the main memory bus. This isolation minimizes the chance that cryptographic secrets are accidentally leaked to the host processor. A high-level overview of the Falcon's memory system is given in Figure 5.

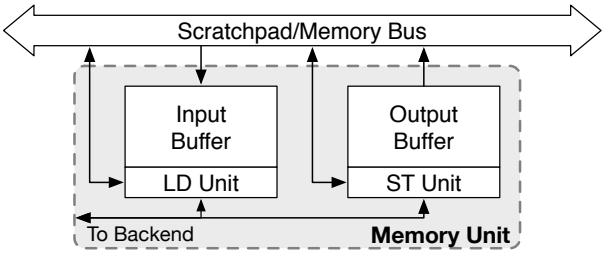


Fig. 5: The memory unit datapath. The memory bus can be configured to use a shared memory or a private scratchpad memory

B. Frontend

The frontend performs instruction fetch and decode, and was designed to remain minimal and efficient while still supporting the compact representation outlined in Section IV-A. It consists of a fetch unit, an instruction decode unit, and a "loop stack", shown in Figure 4. Each cycle, the fetch unit loads 32-bit bundles of up to 2 instructions from memory. It also co-ordinates with the loop stack to keep track of the current and next instruction, as well as manage loop state.

Fetched instruction bundles are stored in a fetch buffer until they can be fully decoded. Once a full instruction has been fetched, the decode unit decodes the instruction and passes it to the backend for execution.

1) The Loop Stack: An important design feature of Falcon is it's lack of data dependent branches. As mentioned in Section IV-A, the main motivation for disallowing data dependent branching is that it simplifies control decisions during instruction fetch. It also allows us to eliminate the logic related to frontend mis-predicts, such as squashing, since all control decisions can be predicted perfectly.

However, the lack of data dependent branching increases the code size needed for ciphers with loops. To address this, we use a hardware "loop stack" which keeps track of state needed for currently executing loops.

When a `LOOP_BEGIN` instruction is detected, the fetch unit pushes the current instruction pointer along an iteration count to the top of loop stack. When a `LOOP_END` instruction is detected, the iteration count at the top of the stack is examined. If it's greater than zero, the instruction pointer is set to the saved instruction pointer and the current iteration count is decremented. Otherwise, the loop is ended by popping the top entry off the loop stack and execution continues with the next instruction.

2) Pre-Decode and Decode Unit: Loop instruction detection is done by the pre-decoder. Loop instructions are required to be 32-bit aligned, which means pre-decoder only needs to check a small number of instruction bits on each fetch to detect a loop. This allows the pre-decode step to be very fast, and stay off the critical path of the machine.

Once an instruction bundle has been pushed into the fetch buffer, the decode unit splits the bundle one instruction at a time. Since the decoding of some instructions (such as `LDi`) depend on the current SIMD width, the decoder keeps track of the value of the most recent `SET_WIDTH` instruction. The current SIMD width is also passed with the decoded instruction to the backend for execution.

C. Backend

Falcon's backend executes the arithmetic and specialized cryptographic operations for each instruction. It is a SIMD design, organized into 16 logical lanes with a datapath width of 16 bits for each lane. If Falcon's lane width is less than 16, each lane can be split into two. Instruction execution is split into three stages: register read, execute, and write back. Since the frontend does not allow data dependent branching, there are no hazards between stages except for register read-after-writes (RaW), which are handled using forwarding paths. The only other source of stalls are memory access (which stalls execution until fully completed) and long-word SIMD operations, which we expand on below.

1) Execution Units: Falcon's ISA can be broken up into three sets of operations that need to be supported by the backend: byte-level permutations and bitslicing, 8-bit SIMD operations, and long-word arithmetic operations. The key design goal of the execution unit was to allow all three of

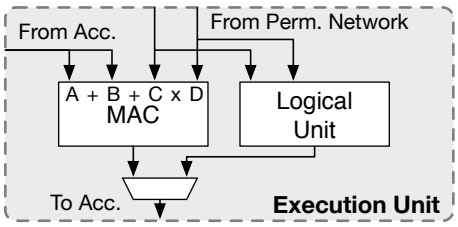


Fig. 6: Datapath of the execution unit.

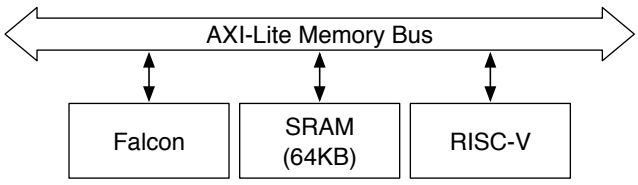


Fig. 7: Overview of experimental setup. Falcon is configured to use a shared memory instead of a scratchpad.

these operations to share as much hardware as possible while still remaining power efficient.

Each individual execution unit contains a 4-input, 16-bit MAC that can be internally segmented into two separate 8-bit MACs. This lets the execution unit support both 8-bit and 16-bit arithmetic operations in a single cycle. Logical SIMD operations are supported using the 16-bit logical unit.

To support wider SIMD widths, Falcon performs school book multiplication using the permutation network to broadcast individual digits. Each cycle, the MACs sum a 32-bit partial product plus the previous accumulation value. In order to coordinate these operations, Falcon includes a small state machine inside of the backend's control logic. This allows Falcon to support arithmetic operations of up to 256 bits in 16 cycles or less.

2) Permutation Network: Falcon contains a 8x32 permutation network to support permutations between SIMD lanes. The network is designed to efficiently support broadcasting operations needed for the long-word operations described in Section V-C1 (i.e., a single lane can be broadcast to all other lanes in less than a cycle). For arbitrary permutations, the permutation network requires 1–4 cycles depending on the current SIMD width.

We chose this design for a few reasons. First, a large permutation network proved to be significantly expensive in area and power, and by only supporting an 8x32 network we decreased the cost by a factor of 4. Second, the majority of ciphers we examined only needed 32-bit permutations rather than arbitrary byte permutations (64%). Additionally, most bit-wise or byte-wise permutations can be transformed into word-wise permutations through the use of bitslicing. As a result, we decided to use a smaller permutation network at the expense of latency for some ciphers.

VI. Evaluation

A. Experimental Setup

We implemented a parameterized version of Falcon in SystemVerilog and performed behavioral RTL simulation using Synopsys VCS to ensure correctness. We then synthesized our implementation with the Synopsys Design Compiler targeting a $180\,\mathrm{nm}$ process with power and timing related optimizations enabled. We implemented large memories using the provided memory compiler, optimizing for power. Finally, we performed place-and-route using Synopsys ICC. Although Falcon is not designed or evaluated for high throughput, we report our maximum achievable clock frequency of 120MHz for completeness.

To provide a fair baseline for evaluation, we synthesized several cipher specific accelerators (AES128, SHA2, and ChaCha20), as well as a size optimized, microcontroller class implementation of RISC-V [11], [12]. Our ChaCha20 implementation was based on the description of the quarter round serial implementation of Salsa20 given by Good [13]. We chose these ciphers for comparison because they are the most common crypto accelerators available in existing SoCs for IoT.

All designs were synthesized separately using the same optimization settings and clock frequency. In order to accurately estimate Falcon's energy consumption inside of a real SoC, we implemented a simple test harness intended to mimic an actual SoC implementation (Fig. 7). Falcon and the RISC-V core are connected over an AXI-Lite bus to a 64K SRAM macro block. Additionally, Falcon is configured to use the SRAM as it's scratch pad memory. All instructions, data, and control signals are sent over the shared AXI-Lite bus. Because we are evaluating on energy consumption rather than latency or throughput, we used a fixed clock frequency of 50MHz for a fair comparison between components. This lower clock speed also represents a more conservative estimate of total energy consumption since the proportion of leakage will be higher.

B. Area

TABLE IV: Breakdown of area and leakage for Falcon implementation in TSMC 180nm

Component	**Area**(um^2)	**(%)**	**Power**(uW)	**(%)**
Falcon	566392	100.0	2.6912	100.0
Frontend	31151	5.5	0.2002	7.5
Regfile	298632	52.7	1.3953	52.3
Execute	236982	41.8	1.1258	42.2
Comb.	317117	56.0	1.4860	55.3
Register	249275	44.0	1.2016	44.6
RISC-V	306065	100.0	1.5098	100.0
AES128	303275	100.0	1.6669	100.0
SHA2	320425	100.0	1.9601	100.0
ChaCha20	123576	100.0	1.4028	100.0

In table IV, we breakdown the area and leakage power for each component of Falcon. Since the frontend was explicitly designed to be as simple as possible, it has almost negligible impact on the total area or static power consumption of the

design. On the other hand, the register file and execution stages are very wide, consuming the vast majority of both power and area, with the register file being slightly higher for both. Compared to the RISC-V processor, Falcon uses 1.85x the area and consumes 1.78x the amount of leakage power. However, the types of devices we're targeting are typically not hyper area constrained; many SoCs we investigated include multiple such accelerators as well as large peripheral modules such as BLE radio. As such we believe this is an acceptable tradeoff for the additional flexibility.

C. Energy Consumption

Since there is no standard of base settings or implementations for cipher operations, it is difficult to provide a fair comparison between designs. For example, the choice of mode for a block cipher can have a significant impact on the overall parallelism, which in turn impacts the throughput and energy consumption. Moreover, many existing implementations have been hand tuned for a specific architecture.

Nonetheless, we have chosen what we believe to be a representative sample of ciphers for evaluating Falcon's total energy consumption and implemented them in Falcon's assembly language. For AES128-ECB, ChaCha20, and SHA256, we evaluated our implementation by calculating the average energy consumed per block after running the algorithms over 1 kB of data. For AES128-ECB, we used a bitsliced implementation instead of a more traditional lookup-table based approach since lookup-tables are highly susceptible to side channels and generally not favored in modern high security software implementations. ECB mode was chosen since it is used as the core operation in other commonly used modes such as CBC or GCM. For Curve25519, we computed a shared secret key, basing our implementation on an ARM NEON implementation [14]. For RSA2048, we perform a signing operation. For R-LWE, we implement the accept operation (excluding the final SHA3 step) based on an ARM NEON implementation [15].

As a baseline, we implemented the same operations in C and optimized them for RISC-V. To match our implementation for Falcon, we implemented AES128 using bitslicing. The C implementations were compiled using GCC with -O3 and run on our RISC-V implementation using the same conditions as our energy evaluation for Falcon.

To perform energy estimation, we used Synopsys PrimeTime PX to propagate simulated activity factors onto the final placed-and-routed design and report average power usage by hardware block. The leakage power of the SRAM block was not included since it's power consumption is independent from Falcon. This average power estimate was multiplied by the total number to cycles to get a total energy estimate.

Table V shows the total energy consumed by each cipher and gives the improvement achieved by Falcon. Overall, Falcon achieves between 5x and 61x improvement across all benchmarks tested.

We breakdown power consumption by source for each cipher and normalize it to the RISC-V baseline in Figure 8.

TABLE V: Total energy consumption and improvement between the RISC-V baseline and Falcon.

	Operation	**Baseline** (nJ)	**Falcon** (nJ)	**Decrease**
Bitsliced AES	Encrypt	9036.0	147.5	61.3x
ChaCha20	Encrypt	2021.6	302.3	6.7x
SHA256	Hash	12630.0	3279.0	3.8x
Curve25519	Key-xchg	40454.0	8163.0	5.0x
RSA2048	Sign	87401.0	16840.0	5.2x
R-LWE	Accept	109502.0	19822.0	5.5x

Falcon reduces the amount of energy used for the frontend (fetch and decode) by about 20% across all ciphers tested. This is due to our SIMD design allowing us to avoid repeatedly paying the decode energy for parallel instructions. For ciphers with high amounts of internal parallelism (Bitsliced AES, ChaCha20) and ciphers with large numbers of arithmetic instructions (Curve25519, RSA, and R-LWE) most of the savings is used for execution. SHA256, on the other hand, has less parallelism and more energy is needed to move data between registers.

However, this only accounts for part of the energy savings on AES. We identify two other areas that account for the rest of the improvement. First, our implementation of AES is bitsliced using pure logical operations and no memory accesses except for instruction fetching. This means Falcon never needs to stall, and achieves close to it's theoretical maximum IPC of 1. Second, our implementation is able to take advantage of `BITSLICE` instruction along with the parallel nature of bitslicing to dramatically reduce the total number of instructions executed. For example, while the RISC-V baseline requires about 50 cycles to convert between the arithmetic and bitsliced representations, Falcon executes the same operation in a single cycle. For the entire AES algorithm Falcon executes just 13% the number of instructions as the baseline processor.

D. Application Analysis

The overarching goal of Falcon is to provide a meaningful reduction in energy consumption for an entire IoT device, such that previously infeasible operations could be incorporated in applications and the secure lifetime of a device can be extended. To examine whether the improvements given in Section VI-C are large enough to make such a difference, we evaluated Falcon's impact on the application described in Section II.

TABLE VI: Falcon energy consumption compared to a software only baseline in a model IoT application.

	Baseline		**Falcon**	
	µJ	Increase	µJ	Increase
Base Application	3100	1.00x	3100	1.00x
With Curve25519	5400	1.74x	3560	1.15x
With R-LWE	9300	3.00x	4200	1.35x

To perform our analysis, we used the estimated the power consumption of our model application from Section II. We

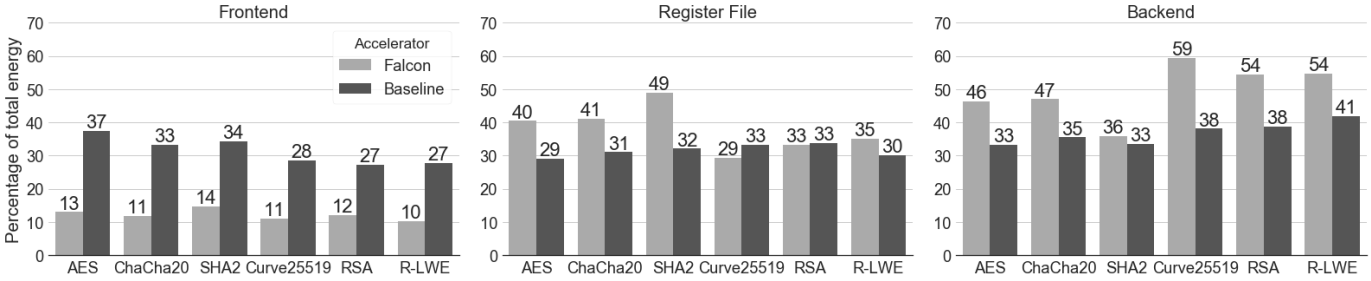


Fig. 8: Energy consumption by source for each cipher

then assumed a 5.0x improvement for Curve25519 and a 5.5x improvement for R-LWE, per our results in Section VI-C. The overall increase energy consumption for the final application using R-LWE was 3.0x for the baseline and 1.36x for Falcon. Since this application has a fixed battery capacity, this translates to a roughly doubled (2.2x) deployment time, a significant increase.

E. Future Flexibility

An important design goal of Falcon is to be flexible, even in the face of a changing security landscape. We make three arguments to support our belief as to why Falcon achieves this flexibility.

First, in our evaluation we were able to accelerate a diverse set of ciphers, each of which tested a different aspect of our design. While we were not able to evaluate all ciphers studied, the ciphers we implemented represent essentially all of the operations identified in our previous analysis.

Second, portability to existing platforms is a key requirement for the adoption of new ciphers. While ciphers could be developed that Falcon cannot accelerate, we believe it is unlikely such a cipher would gain widespread use since it would also be unlikely to be compatible with existing hardware.

Finally, we see a broad trend of new ciphers being designed to use the SIMD units included in modern ISAs. For example, the designers of ChaCha20 [16] specifically mention SIMD as an important consideration in their design. This bodes well for Falcon since our evaluation shows it performs best on ciphers with large amounts of parallelism. While none of these reasons can provide a guarantee of compatibility, taken together, we believe that Falcon provides sufficient flexibility to accelerate future cryptographic operations.

VII. Related Work

Cryptoraptor: Cryptoraptor [7] is the work that most strongly resembles Falcon. It features a reconfigurable, 4-way, 20 stage pipelined processor with a compact finite state machine encoding for control and execution units specialized for symmetric ciphers and hash functions. However, Cryptoraptor explicitly does not support asymmetric ciphers or ciphers that require multiplication. Falcon supports asymmetric ciphers, along with symmetric ciphers and hash functions, providing the full range of ciphers used on IoT devices.

Reconfigurable crypto co-processors: Other flexible cryptographic co-processors similar to Falcon have been proposed. Cryptomanic [17] and CCproc [18] are 4-wide VLIW architecture designed to support many operations common in symmetric ciphers. Cryptonite [19] is another co-processor design that targets symmetric ciphers and hash functions. It uses a simple 3-stage pipeline with a highly flexible arithmetic unit. Eslami, et. al [20] demonstrate an unnamed architecture designed to accelerate AES, TDES, and ECC, using a 256-bit wide datapath. COBRA [21] targets block ciphers using a course grained reconfigurable array structure of processing elements, similar in spirit to an FPGA.

All of these works are limited to a fairly small range of ciphers and only Cryptonite is evaluated on hash functions. Falcon adds significantly more flexibility, allowing for the support of symmetric and asymmetric ciphers, as well as hash functions. Additionally, Falcon includes as a design goal future compatibility and is able to modern ciphers such as ChaCha20 or R-LWE.

Instruction set extensions: Many processor instruction sets include instructions or extensions to accelerate a predefined set of ciphers [22]–[25]. While these can make specific ciphers extremely efficient, they are limited to a small set of supported cryptographic operations.

Other instruction extensions have been designed to accelerate a broader range of applications and have particular applicability to cryptography. In particular, bitslicing [26], permutation [27], and table look-up [28] extensions have all been proposed as more general ways to accelerate bit-and-byte operations. These are similar to Falcon's `BITSLICE` and `PERMUTE` instructions, although Falcon adds significant additional flexibility in the data width.

Likewise, SIMD instruction sets are commonly used to accelerate cryptogrograpy [9], [14]. However, while adapting SIMD hardware has proven successful in the past for throughput, previous research has not found a significant energy improvement from SIMD alone. Falcon adds features that allow it to improve overall energy consumption compared to existing architectures. Additionally, Falcon builds on previous designs by adding additional flexibility in data width, using a very short encoding scheme, and by reducing the complexity of supported control flow.

Differences between Falcon and previous work There are

several important distinctions between Falcon and the previous work we identified. First, Falcon has the design goal of supporting low power devices, instead of targeting maximum throughput. This has several important consequences, including minimizing the width of Falcon's execution units and the simplification of our encoding scheme.

Second, to the best of our knowledge, Falcon is the only crypto-specific accelerator in the literature to support asymmetric cryptography, symmetric cryptography, and hash functions in a single design. This support was a key motivator for the flexible datapath width, which was a key contribution of our design.

Finally, Falcon provides a more diverse performance evaluation compared to most previous work. Most previous work (with the exception of Cryptoraptor) only evaluated their designs on algorithms with similar structures. In our work, we evaluate Falcon using several different kinds of algorithms, including traditional round-based symmetric ciphers, modern stream ciphers, long word asymmetric operations, and post-quantum signature schemes. We also evaluate these results in the context of a deployed device, and estimate how our improvements affect the tractability of future cipher flexibility.

VIII. Conclusion

IoT devices with a long lifetime must support new ciphers as security requirements evolve, which limits the effectiveness of current fixed function accelerators. However, limited energy budgets mean these devices cannot just rely on software, instead requiring a form of hardware acceleration that could provide significant energy savings, while supporting a wide range of cryptographic operations.

To this end, we designed Falcon, an architecture designed to flexibly accelerate cryptography. We evaluated Falcon by synthesizing our design and measuring it's performance on key set of ciphers commonly used in IoT applications. We found that Falcon can provide between a 5x-60x improvement in energy consumption at a cost of 1.85x additional area compared to a RISC-V baseline. Since many IoT deployments are energy rather than area constrained, we believe this tradeoff is acceptable for the increased flexibility. On a model application with an evolving security model, this translates to 2.2x the overall device lifetime.

References

[1] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 447–462, IEEE Computer Society, 2010.

[2] Wikipedia, "2016 dyn cyberattack." https://en.wikipedia.org/wiki/2016_Dyn_cyberattack.

[3] "Fips pub 197, advanced encryption standard (aes)," 2001. U.S.Department of Commerce/National Institute of Standards and Technology.

[4] Anonymous, "Secure Hash Standard (SHS)," Federal Information Processing Standards Publication FIPS Pub 180-4, National Institute for Standards and Technology, Mar. 2012.

[5] M. Integrated, "Low-power, arm cortex-m4 with fpu-based soc with contactless transceiver." https://www.maximintegrated.com/en/products/digital/microcontrollers/MAX32566.html.

[6] J. Hamborsky, A. Kroger, and Wolfe, *Epidemiology and prevention of vaccine-preventable diseases*. Centers for Disease Control and Prevention, 2015.

[7] D. Chiou and G. Sayilar, "Cryptoraptor: High throughput reconfigurable cryptographic processor," in *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, pp. 154–161, IEEE Press, 2014.

[8] E. Biham, "A fast new DES implementation in software," in *FSE*, vol. 1267, pp. 260–272, Springer, 1997.

[9] E. Käsper and P. Schwabe, "Faster and timing-attack resistant AES-GCM.," in *CHES*, vol. 5747, pp. 1–17, Springer, 2009.

[10] Google LLC, "Tock-on-titan." https://github.com/google/tock-on-titan/blob/735282fd77ea8c2a6c8cfb69ed8a3ae28fded6a3/h1b/src/crypto/dcrypto.rs#L18, 2019.

[11] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume i: Base user-level ISA," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[12] C. Wolf, "PicoRV32 - a size-optimized RISC-V CPU." https://github.com/cliffordwolf/picorv32, 2019.

[13] T. Good and M. Benaissa, "Hardware results for selected stream cipher candidates," *State of the Art of Stream Ciphers*, vol. 7, pp. 191–204, 2007.

[14] D. J. Bernstein and P. Schwabe, "NEON crypto," in *Cryptographic Hardware and Embedded Systems – CHES 2012* (E. Prouff and P. Schaumont, eds.), vol. 7428 of *Lecture Notes in Computer Science*, pp. 320–339, Springer-Verlag Berlin Heidelberg, 2012.

[15] S. Streit and F. De Santis, "Post-quantum key exchange on armv8-a: A new hope for neon made simple," *IEEE Transactions on Computers*, 2017.

[16] D. J. Bernstein, "ChaCha, a variant of Salsa20," in *Workshop Record of SASC*, vol. 8, pp. 3–5, 2008.

[17] L. Wu, C. Weaver, and T. Austin, "Cryptomaniac: a fast flexible architecture for secure communication," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pp. 110–119, IEEE, 2001.

[18] D. Theodoropoulos, A. Siskos, and D. Pnevmatikatos, "Ccproc: A custom vliw cryptography co-processor for symmetric-key ciphers," in *International Workshop on Applied Reconfigurable Computing*, pp. 318–323, Springer, 2009.

[19] R. Buchty, N. Heintze, and D. Oliva, "Cryptonite–a programmable crypto processor architecture for high-bandwidth applications," in *International Conference on Architecture of Computing Systems*, pp. 184–198, Springer, 2004.

[20] Y. Eslami, A. Sheikholeslami, P. G. Gulak, S. Masui, and K. Mukaida, "An area-efficient universal cryptography processor for smart cards," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 14, no. 1, pp. 43–56, 2006.

[21] A. J. Elbirt and C. Paar, "An instruction-level distributed processor for symmetric-key cryptography," *IEEE Transactions on Parallel and distributed Systems*, vol. 16, no. 5, pp. 468–480, 2005.

[22] S. Gueron, "Intel® advanced encryption standard (AES) new instructions set," *Intel Corporation*, 2010.

[23] J. Burke, J. McDonald, and T. Austin, "Architectural support for fast symmetric-key cryptography," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 5, pp. 178–189, 2000.

[24] A. J. Elbirt, "Fast and efficient implementation of AES via instruction set extensions," in *Advanced Information Networking and Applications Workshops, 2007, AINAW'07. 21st International Conference on*, vol. 1, pp. 396–403, IEEE, 2007.

[25] S. Tillich and J. Großschädl, "Instruction set extensions for efficient AES implementation on 32-bit processors," in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 270–284, Springer, 2006.

[26] P. Grabher, J. Großschädl, and D. Page, "Light-weight instruction set extensions for bit-sliced cryptography," *Cryptographic Hardware and Embedded Systems–CHES 2008*, pp. 331–345, 2008.

[27] R. B. Lee, Z. Shi, and X. Yang, "Efficient permutation instructions for fast software cryptography," *IEEE Micro*, vol. 21, no. 6, pp. 56–69, 2001.

[28] A. M. Fiskiran and R. B. Lee, "On-chip lookup tables for fast symmetric-key encryption," in *Application-Specific Systems, Architectures Processors, 2005. ASAP 2005. 16th IEEE International Conference on*, pp. 356–363, IEEE, 2005.