Triads below the ISA: Optimizing Cryptographic Software in CPU Microcode

Sabrina Manickam[1], Rowan Fimmano[2], Peter Schwabe[3,4] and Yuval Yarom[1]

[1] Ruhr University Bochum, Bochum, Germany
[2] Adelaide University, Adelaide, Australia
[3] Max Planck Institute for Security and Privacy, Bochum, Germany
[4] Radboud University, Nijmegen, The Netherlands

Abstract. Cryptographic software is typically optimized for the instruction set architecture exposed by the processor. Modern processors, however, contain deeper execution layers with computational capabilities and constraints that differ from those of the instruction set architecture (ISA). This work explores the programmable microcode layer in x86 as an implementation target. We characterize the microcode domain as a distinct target architecture for cryptographic computation and adapt implementations to the state, operations, invocation interface, capacity, and scheduling resources it exposes.

We instantiate this methodology on Intel Goldmont with Keccak $f[1600]$ and X25519, two case studies with substantially different computational structures. Our Keccak implementation reduces permutation latency by 7.36% relative to the fastest native implementation in our comparison, with the improvement remaining visible in complete SHA3-256 computations. For X25519, our microcode multiplication and squaring kernels free the multiplier from fixed operand registers and restrictive carry-flag semantics, outperforming the native implementations we evaluate. The resulting complete X25519 implementation requires 7.3% fewer cycles than s2n-bignum and 9.8% fewer cycles than OpenSSL. Together, these results demonstrate that looking below the ISA exposes a distinct design space for cryptographic implementation and provide a systematic methodology for exploring it.

Keywords: Microcode · Keccak · X25519 · Cryptographic Implementation

1 Introduction

High-performance cryptographic software is closely shaped by the execution interface exposed by the processor. Implementations choose representations, operation sequences, register allocations, and algorithmic variants according to the resources and restrictions of the target architecture. On x86, this optimization normally ends at the instruction set architecture (ISA). The instructions themselves may be implemented through lower-level mechanisms, but these mechanisms are ordinarily hidden from software and are therefore not considered part of the implementation design space.

Microcode provides an opportunity to examine what happens when this boundary changes. It forms an execution layer below the x86 instruction set and exposes a computational interface that differs from the architectural one. Prior work [KKF+17, KKF+18, BES+23, WNR25] on x86 microcode has primarily studied this layer to understand processor behavior, modify existing instructions, or to introduce new functionality. In particular, reverse engineering of Intel Goldmont [ESG23, BES+23, Kro23] provides an experimentally accessible microcode environment in which these mechanisms can be studied and modified. We use this environment for a different purpose. Rather than treating microcode

as an object of reverse engineering, we use it as an experimental platform for studying cryptographic implementations below the ISA.

This perspective leads to a different question from whether cryptographic code can simply be moved into microcode. We ask how cryptographic implementations change when the execution layer below the ISA itself becomes the implementation target. The interface exposed at this level differs from the architectural interface for which existing software has been designed. Understanding these differences therefore requires reconsidering established implementation choices rather than simply reproducing them at a lower level.

We study these effects through a methodology that treats microcode as a distinct implementation target rather than as a lower level encoding of an existing x86 implementation. The methodology begins by characterizing the microcode execution environment and uses this understanding to systematically explore how cryptographic computation can be mapped to this lower level. Rather than translating an existing instruction sequence, we design specifically for the microcode domain. We use the properties of both the computation and of microcode to derive an implementation suited to the layer on which it will execute. This process exposes design choices that are difficult to observe when the ISA is treated as the lowest available abstraction and allows us to identify techniques for exploiting them systematically.

Cryptography provides a useful setting for this study because optimized implementations already make careful use of the architectural interface. Competing with these implementations provides a stronger test than demonstrating that computation can execute in microcode. More importantly, cryptographic code exposes structured dataflow and well understood implementation tradeoffs. They allow us to examine not only whether an alternative execution interface is competitive but also which properties of that interface change the implementation strategy and under which conditions those changes are useful.

We instantiate this methodology using Keccak $f[1600]$ and X25519 as two case studies. To demonstrate its feasibility, we redirect existing instructions to our custom microcode patches. The selected instructions serve no functionality required by other software in our experimental environment, allowing us to explore microcode as an alternative implementation target. These case studies span hashing and public-key cryptography and present substantially different computational structures, allowing us to study whether common implementation principles emerge when both are designed for the same underlying execution layer. More importantly, the two case studies allow us to extract these observations from their individual implementations and formulate them as reusable techniques. Our goal is therefore not only to demonstrate cryptographic computation in microcode, but also to understand how knowledge of a lower execution layer can inform the way cryptographic implementations are designed.

Our work makes the following contributions:

- We characterize programmable microcode as a distinct implementation target for cryptographic software. This characterization captures the microcode execution environment and provides a systematic way to reason about the interaction between cryptographic computations and the underlying processor mechanisms.
- We apply this characterization to Keccak $f[1600]$ and X25519 on Intel Goldmont. Through these case studies, we identify implementation techniques that exploit properties of the microcode execution layer and formulate them so that they can guide implementations beyond the individual constructions studied here.
- We evaluate these techniques through experiments that isolate their effect on the resulting implementations. The experiments show how knowledge of the underlying execution layer can change representation, scheduling, carry handling, and data movement decisions, and how these choices translate into measurable performance differences.
- We demonstrate the practical potential of this approach with complete implementations on Goldmont. Our Keccak implementation reduces permutation latency by 7.36%

relative to the fastest native implementation in our comparison, with the improvement remaining visible in complete SHA3-256 computations. For X25519, our microcode multiplication and squaring kernels outperform the native implementations we evaluate, and the resulting complete implementation requires 7.3% fewer cycles than s2n-bignum and 9.8% fewer cycles than OpenSSL.

Together, these results show that looking below the ISA provides more than a different encoding of the same computation. It provides another level at which cryptographic implementations can be designed. By understanding the properties of that level and adapting the computation to them, we expose a broader set of implementation choices and a systematic way to reason about their performance. The code artifact for this work is available at https://github.com/0xADE1A1DE/ucrypt.

2 Background

We first establish the relevant technical background required to understand this paper. This includes the microcode layer of modern x86 processors and the cryptographic primitives we target in our case studies.

2.1 Intel Microcode

Modern x86 processors do not execute all instructions directly. An internal decode layer translates architectural macro-instructions into sequences of *micro-operations* (µops) [KKF+17, BES+23]. Frequent or simple instructions are handled by hardwired decoders. More complex or infrequent instructions are instead implemented as small programs of micro-operations (µops). These programs are written by the processor manufacturer and stored in the on-die *Microcode ROM* (MSROM). When such an instruction is encountered, execution enters the *Microcode Sequencer* at a fixed entry address associated with that instruction. The sequencer then reads and issues the corresponding sequence of µops from the MSROM.

Since the microcode stored in the MSROM is fixed at fabrication, processors provide a mechanism for applying corrections after deployment. This mechanism consists of a writable *patch RAM* together with a set of *match registers*. A microcode update can populate the patch RAM and configure the match registers with an instruction's MSROM address whose execution should be redirected. When the Microcode Sequencer reaches an address contained in a match register, control is transferred from the MSROM to the corresponding location in the patch RAM. If no match occurs, execution proceeds normally from the MSROM. Figure 1 illustrates these two execution paths. Vendors use microcode updates to correct processor errata and deploy security mitigations, including mitigations for vulnerabilities such as Spectre and Meltdown [KHF+19, LSG+18]. We instead use the same mechanism to install cryptographic routines in microcode. We refer to a self-contained cryptographic routine installed in this way, such as a field multiplication or a complete permutation, as a *kernel*, and to each invocation of a kernel through its triggering instruction as a *firing*.

On the Intel platforms for which the internal microcode format has been studied, microcode is organized as sequences of *triads* [ESG23, BES+23, Kro23]. Each triad contains three µops and one *sequence word*. The sequence word controls the flow out of the triad and encodes information including the next microcode address, synchronization, and memory barriers [Kro23].

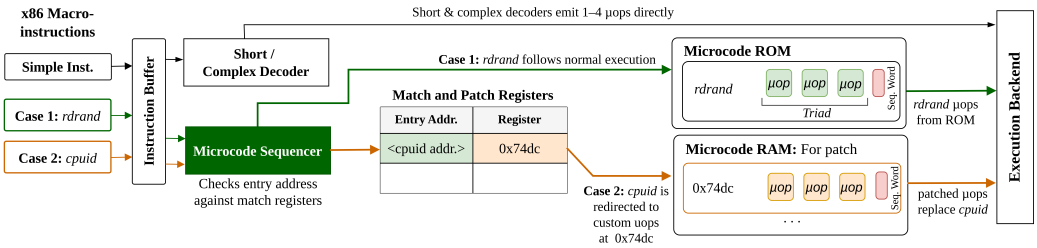

Figure 1: The x86 decode and microcode pipeline with the match and patch mechanism

2.2 Keccak

Keccak $f[1600]$ [BDPV13] is the 1600-bit permutation used by the SHA-3 and SHAKE functions standardized by NIST in FIPS 202 [Nat15]. Its state is arranged as a 5×5 array of 64-bit lanes $A[x, y]$. The permutation applies 24 rounds, each consisting of the five transformations θ, ρ, π, χ, and ι, as shown in Algorithm 1.

Algorithm 1 A round of Keccak $f[1600]$. All indices are taken modulo five, $r[x, y]$ and RC are the constants fixed by the specification, and ROL is a 64-bit left rotation.

Require: 1600 bit state $A[x, y]$, round constant RC
Ensure: updated state $A[x, y]$

θ **step**

$C[x] = A[x, 0] \oplus A[x, 1] \oplus A[x, 2] \oplus A[x, 3] \oplus A[x, 4]$ $\forall x$

$D[x] = C[x - 1] \oplus \mathrm{ROL}(C[x + 1], 1)$ $\forall x$

$A[x, y] = A[x, y] \oplus D[x]$ $\forall x, y$

ρ **and** π **steps**

$B[y,\ 2x + 3y] = \mathrm{ROL}(A[x, y],\ r[x, y])$ $\forall x, y$

χ **step**

$A[x, y] = B[x, y] \oplus \big((\neg B[x + 1, y]) \wedge B[x + 2, y]\big)$ $\forall x, y$

ι **step**

$A[0, 0] = A[0, 0] \oplus RC$

Software implementation. Efficient software implementations of Keccak use several established techniques to adapt the permutation to the resources of the target architecture, including *plane-by-plane processing*, *in-place processing*, *lane complementing*, and *round unrolling* [BDPVA14].

The 25 lanes of Keccak $f[1600]$ occupy 200 bytes before accounting for the intermediate values required during a round. Scalar x86-64 provides 16 general-purpose registers, some of which are also needed for addresses and temporary values. The complete state and all round intermediates therefore cannot remain simultaneously in those registers.

Plane-by-plane processing addresses this register pressure by organizing the computation around one five-lane plane at a time, which limits the temporary state required during the round [BDPVA14]. A direct implementation of Algorithm 1 would also require a separate 25-lane state B for the output of ρ and π. *In-place processing* avoids this additional storage. Lane $(0, 0)$ is fixed by π, while the other 24 lanes form a single cycle. The lanes can therefore be moved to their new positions using one temporary value. The rotations of ρ can be applied as the lanes are moved. The expression used by χ also requires consideration in scalar software. Each output lane contains the term $(\neg b) \wedge c$. Implementations that cannot express this term directly can use *lane complementing*, which maintains selected lanes in complemented form, reducing the number of explicit negations required during χ.

Finally, the 24 rounds may be partially or fully unrolled. Unrolling removes repeated

round control and permits computation across round boundaries to be specialized, at the cost of a larger static implementation. In particular, a fully unrolled implementation can account for the permutation of lane positions in the register assignment of the following round rather than restoring a common assignment after every round.

2.3 X25519

X25519 defines scalar multiplication on a Montgomery curve over the prime field $\mathbb{F}_p$ with $p = 2^{255} - 19$ [Ber06]. Scalar multiplication is computed using a 255-iteration Montgomery ladder operating on projective x-coordinates. Algorithm 2 shows one ladder step. Each step performs five field multiplications, four field squarings, and one multiplication by the constant 121665. After the ladder, the result is converted to affine form as $X_2 Z_2^{-1}$. The inversion uses a fixed addition chain containing a further 254 squarings and 11 multiplications. One scalar multiplication therefore performs 1287 field multiplications and 1274 field squarings, which account for most of its arithmetic.

Algorithm 2 One Montgomery ladder step for X25519 [Ber06]. All operations are in $\mathbb{F}_{2^{255}-19}$, and X_1 is the affine input coordinate.

Require: X_1, (X_2, Z_2), (X_3, Z_3)
Ensure: updated (X_2, Z_2), (X_3, Z_3)
1: $A = X_2 + Z_2, \quad AA = A^2$
2: $B = X_2 - Z_2, \quad BB = B^2$
3: $E = AA - BB$
4: $C = X_3 + Z_3, \quad D = X_3 - Z_3$
5: $DA = D \cdot A, \quad CB = C \cdot B$
6: $X_3 = (DA + CB)^2$
7: $Z_3 = X_1 \cdot (DA - CB)^2$
8: $X_2 = AA \cdot BB$
9: $Z_2 = E \cdot (AA + 121665 \cdot E)$

Software Implementation. Efficient 64-bit implementations of arithmetic modulo $2^{255} - 19$ commonly use either an unsaturated 5×51 representation or a saturated 4×64 representation [BDL+11].

In the 5×51 representation, a field element a consists of five 51-bit limbs $a_0, \ldots, a_4$ stored in 64-bit words, so that $a = \sum_{i=0}^{4} a_i 2^{51i}$. The unused bits in each word provide headroom for intermediate growth. A schoolbook multiplication $a \cdot b$ contains the 25 limb products $a_i b_j$, while squaring can exploit symmetry and requires only 15 distinct products. The products that contribute to each output limb can be collected in a separate wide accumulator, so a multiplication works on five accumulators whose arithmetic is independent. Each 51-by-51-bit product is wider than a machine word, and we write P_{lo} and P_{hi} for its low and high 64-bit halves. Several products contribute to each accumulator, so an implementation must keep both halves of the accumulator for output limb j, which we denote L_j and H_j, and propagate the carry c out of L_j into H_j.

The `amd64-51` implementation follows this organization [BDL+11]. On Goldmont, it holds each accumulator in a pair of registers and adds every product to its pair with an `ADD` for the low word followed by an `ADC` for the high word. Products whose weight exceeds 2^{255} are first multiplied by 19 and then added to a lower accumulator, using the congruence $2^{255} \equiv 19 \pmod{2^{255} - 19}$. After accumulation, the wide values are split back into radix-2^{51} limbs, and the carry out of the highest limb is folded into the lowest limb with the same congruence [BDL+11, Ber].

The saturated 4×64 representation stores a field element in four full 64-bit words. A multiplication then needs only 16 limb products, but no limb has headroom to absorb intermediate growth, so overflow must be propagated between adjacent words. Arithmetic in this

representation is therefore organized around carry chains between adjacent words, whose dependencies constrain the order of the computation. Two of the native implementations in our evaluation, `amd64-64` [BDL+11] and `s2n-bignum`, use this representation.

Carry state on x86-64. Although the five accumulators of the 5×51 representation are independent, on Goldmont, which lacks the ADX and BMI2 extensions, x86-64 provides a single carry flag through which every `ADD`–`ADC` pair must pass. Since `MUL` also overwrites the carry flag, each pair must directly follow the multiplication that produces its product. Native implementations therefore serialize the accumulator updates through one flag, even though the arithmetic of the different accumulators is independent. The mechanisms an architecture provides for holding and communicating carry state thus shape how efficiently it can execute this representation.

2.4 Target Machine

We use an Intel Celeron N3350 based on the Goldmont microarchitecture as the target platform for our study. To obtain access to the internal microcode facilities required by our experiments, we place the processor in the undocumented *Red Unlock* debug state using the publicly available procedure based on Intel TXE and INTEL-SA-00086 [EG18, ESG23]. This state enables the low-level access required to inspect and modify the microcode patching structures on Goldmont.

3 Characterizing the Microcode

In this section, we characterize features of the microcode that impact coding for cryptography. We start with a summary of information that has been previously disclosed, which we use for our implementation. We follow with the results of our investigations of the microcode domain, completing some missing information in past information and identifying new properties of the microcode.

3.1 Previously Disclosed Features

Register Availability. The x86-64 architecture features 16 general-purpose registers, which we refer to as the *architectural* registers.[1] In practice, the standard calling conventions assign special roles for some of these registers. For example, the `rsp` register is reserved as a stack pointer. Consequently, the number of registers that are actually available to the program is smaller.

In addition to the architectural registers, the microcode has access to 16 temporary registers [ESG23, ESG21]. These allow microcode to store more value in registers and avoid spilling [Cha82]. Spilling has a significant impact on the performance of cryptographic code [KEG+23]. Hence, we expect that using additional registers will improve performance.

Three-Operand µops. Most arithmetic and bitwise operations in the x86-64 ISA use two operands, where one of the operands serves as both a source and a destination. Consequently, these operations are destructive and require an additional register copy if there is a need to preserve the source operands. In contrast, µops support two source operands and a dedicated destination operand, allowing non-destructive operations. Compared to ISA operations, the microcode does not support memory addresses as operands of arithmetic and bitwise operations, requiring additional load and store µops. We note that decoding

[1] The Intel Advanced Performance Extension (APX) supports 32 architectural registers and non-destructive source registers [Int23]. At the time of writing, APX is not supported in mainstream processors.

ISA instructions to µops implicitly inserts load and store µops, when the ISA instruction operates on memory operands.

Microcode-Specific Operations. Microcode also exposes operations that are not directly available as architectural x86-64 instructions [ESG23]. One particular example is `µNOTAND`, which computes the bitwise AND of one source operand with the inverse of the other. Using this µop, e.g., as part of the computation of the χ operation of Keccak (cf. Section 4), removes one µop compared to limiting the computation to the vocabulary available in the ISA.

Available Patch RAM. The writable patch RAM on Goldmont is mapped into the microcode address space at `U7c00`–`U7dff`, directly above the microcode ROM [ESG23, BES+23]. Each triad occupies four consecutive µop addresses, the fourth of which is unpopulated [ESG23]. This 512-address window holds 128 triads, i.e., at most 384 µops, each with its own sequence word.

3.2 New Information

`µMUL` Semantics. The x86-64 `MUL` instruction has register constraints. It uses `RAX` as an implicit source register, and `RAX` and `RDX` as implicit destination registers. This design places significant constraints on register allocation, and forces delays due to required register moves [ABB+20, Fal22]. The `MULX` instruction, which alleviates many of these constraints, is not supported by the Goldmont processor.

Prior work has identified the micro-op `µMUL`, which corresponds to the `MUL` instruction, but does not describe its operation [Kro23]. We experiment with `µMUL` to recover its behavior and identify that it multiplies two of its operands, stores the high bits of the result in a third operand, and overwrites one of the source operands, storing the low bits of the result into it. While `µMUL` is destructive, it does offer significantly more flexibility in register use, reducing register pressure.

Condition Flag Handling. Multi-precision arithmetic relies heavily on carry propagation. At the x86-64 level, carries are communicated through architectural flags, creating implicit dependencies between successive operations. In contrast, past work did not identify an implicit method for updating the architectural flags. Instead, past work identified an explicit update, using the `µMOVEINSERTFLGS` µop [BES+23]. Additionally, Borrello et al. [BES+23], hint that registers may be associated with condition flags and show the `µSETCC` that sets a register to 0 or 1 based on a condition flag.

We experiment with the microcode to better identify the behavior of condition flags. We first characterize the behavior of per-register flags. Specifically, we find that each temporary register has a set of condition flags that are updated when the register is used as the destination of an arithmetic or a bitwise instruction. Having multiple sets of condition flags can alleviate the implicit dependencies of condition flags in x86-64 [OGGF12], similar to the `ADCX` and `ADOX` instructions of the `ADX` extension.

Additionally, we find that the microcode does support updating the architectural flags based on the result of arithmetic and bitwise instructions. Specifically, we identify that bit 45 of the microcode instruction, also known as `MOD2`, acts similar to the `.s` modifier in the ARM instruction set, allowing selective flag updates. Unlike the `ADCX` and `ADOX` instructions, selective flag updates do not allow interleaved execution of multiple addition chains, but it does allow breaking the implicit dependencies between arithmetic instructions.

Entry Cost. To work in the microcode domain, we need to understand the cost of entering and leaving patches, i.e., the cost of invoking a microcoded kernel. To measure it, we hook a complex instruction, in our case `vmwrite` to patches of 3, 9, 15 and 27 `µNOP` instructions (one, three, five and nine triads). For each patch size, we execute 1,000

invocations per batch and repeat the measurement over 5,000 batches, reporting the median in core cycles.

The one-triad patch requires 7.95 cycles per invocation. Larger patches require 9.95, 11.95, and 15.96 cycles for 3, 5, and 9 triads, respectively. The cost therefore increases by approximately one cycle per additional triad, with a fixed overhead of about 8 cycles.

3.3 Implementation Principles

These properties lead us to design for the domain rather than translating an existing x86-64 implementation directly into micro-operations. Given the knowledge of the patch entry cost and available space of 128 triads in the patch RAM, we decide which parts of our implementation to place in microcode. The patch must contain enough work to amortize its invocation cost and should try to keep intermediate values within the microcode domain to avoid the overhead of moving them across the architectural interface.

Another aspect of the design is to make use of the available temporary registers to reduce register pressure and avoid spilling. We also design the architectural interface around the microcode implementation, so that inputs and outputs require as little movement as possible. In cases wherein a µop is available that benefits the computation, we use it to reduce the number of µops required for the computation.

These principles guide the design of our implementations, which we describe in the following sections.

4 Keccak-$f[1600]$ in Microcode

In this section, we present our implementation of Keccak $f[1600]$ in Goldmont microcode. We describe the design decisions used to adapt the permutation to the microcode execution domain and evaluate their effect on the resulting implementation. We then evaluate the complete permutation and its integration into SHA3-256.

4.1 Design Strategy

Our strategy is to execute the complete Keccak $f[1600]$ permutation within a single microcode invocation. As Section 2.2 describes, Keccak $f[1600]$ consists of 25 lanes of 64 bits each, and has 24 rounds. We aim to make use of the temporary registers to keep the complete state resident throughout the permutation, and to avoid spilling to memory. We place 13 lanes in architectural registers and 12 lanes in temporary registers, leaving the remaining registers for round intermediates and traversal state.

A completely unrolled implementation would exceed the available patch RAM. The round itself, however, is small enough to remain entirely within the patch. We therefore write the round body as a single sequence of 56 triads. So our design is to load the state once, loop the same 56-triad round body 24 times, and store the state only after the final round. Microcode supports looping using the `µUJMPCC_DIRECT` micro-op. Including the 26-triad entry and 26-triad exit sequences, the complete implementation occupies 108 triads and fits within the available patch RAM.

To enable our design, we assign each state position to a fixed register and restore this register layout at the end of every round. The same looped body can then operate repeatedly on the resident state without needing rearrangement between rounds. Within each round, we adapt established in-place Keccak techniques [BDP+12], such as combining θ, ρ, and π to place transformed lanes directly into their destination registers and evaluating χ one row at a time. Together, these choices allow one microcode invocation to execute the complete permutation while using a single looped implementation and keeping the state resident throughout.

4.2 Implementation

We now describe how we realize this strategy in microcode. We first show how the Keccak state and round intermediates are assigned to the available registers, and then describe how each round updates this resident state in place.

4.2.1 Whole State Residency and Register Allocation

Our first design decision is to keep the complete Keccak state in registers across all 24 rounds. A patch can access 16 architectural and 16 temporary registers. We assign 25 state lanes to 13 architectural registers and 12 temporary registers, and `RCX` holds the address of the working buffer. This leaves six registers for the intermediate values required within a round. The working buffer is a statically allocated array of 56 words of 64 bits in the address space of the calling process, and every memory access of the patch is addressed relative to `RCX`. It holds the 25 lanes, the five column parities, the round constant position, the saved value of `RSP`, and the 24 round constants.

Figure 2 shows the complete allocation. The green registers hold the 25 lanes $A[x, y]$. Each lane has a fixed home register. We number the lanes as $i = x + 5y$, starting from zero. Lane i is stored in the ith register of the sequence `RDI`, `RSI`, `RBX`, `RDX`, `RBP`, `R8`–`R15`, and `TMP0`–`TMP11`. Every step of the round writes its results back into the home registers of their positions, so the same mapping holds at every round boundary. The shared round body can therefore execute repeatedly without loading, storing, or rearranging the state between rounds. The state is loaded from the working buffer once, before the first round, and stored once, after the last.

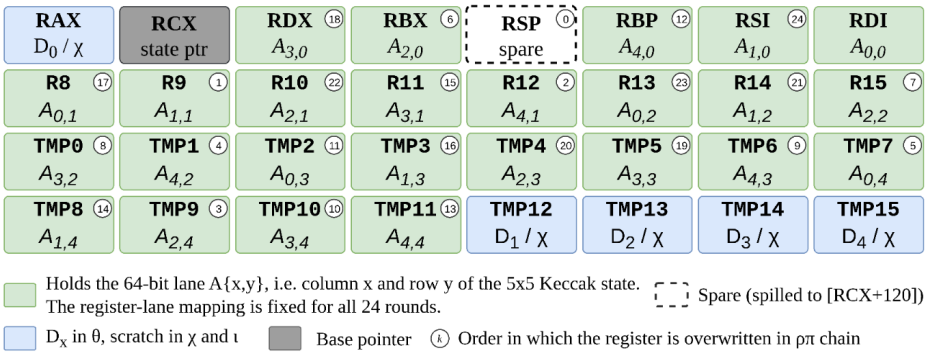

Figure 2: Register allocation for Keccak $f[1600]$

The remaining registers support the computation within a round. `RCX` points to the middle of the working buffer, so that the state, the column parities, the round constant position, and the saved value of `RSP` all lie within reach of the signed 8-bit displacement of microcode loads and stores. The round constant table lies beyond this reach and is addressed through a register index instead. `RAX` and `TMP12` through `TMP15`, shown in blue, provide five registers for round intermediates. We obtain the sixth register by repurposing `RSP`. The first triad of the patch saves its architectural value to the working buffer, and the last triad restores it before execution returns to architectural code. This requires that no interrupt is handled while the patch runs. Since the patch executes as part of a single `vmwrite` instruction, the processor delays interrupts until the patch completes, which we verified experimentally. Over 3.9 million executions of the Keccak patch, during which the core received 2.2 million interrupts, every result was correct and `RSP` always returned unchanged. A longer synthetic patch that runs for about 435 µs confirmed that interrupts arriving during its execution are delivered only after it finishes. The circled numbers in

Figure 2 indicate the order in which the state registers are overwritten during the in-place $\rho\pi$ traversal described below.

The main register pressure arises during θ. This transformation first computes the five column parities $C[x]$ and then derives the five mixing values $D[x]$,

$$C[x] = \bigoplus_{y=0}^{4} A[x, y], \qquad D[x] = C[x-1] \oplus \mathrm{ROL}(C[x+1], 1).$$

Keeping all five $C[x]$ and all five $D[x]$ values in registers alongside the state would require ten registers, but only six remain. The two sets, however, have different reuse. Each $C[x]$ contributes to only two D values, whereas each $D[x]$ is subsequently applied to all five lanes of one column. We therefore store the $C[x]$ values in the working buffer and retain the five $D[x]$ values in `RAX` and `TMP12` through `TMP15`. Storing $C[x]$ costs five stores and ten loads per round. Storing $D[x]$ instead would add a load to each of the 25 lane updates, because the $\rho\pi$ traversal visits the lanes in cycle order rather than column by column. `RAX` accumulates each column parity with four XORs and stores it to the buffer. Each $D[x]$ is then formed by loading $C[x+1]$ into `RSP`, rotating it by one bit, loading $C[x-1]$ directly into the register of $D[x]$, and combining the two with an XOR. Computing $C[x]$ and $D[x]$ occupies the first 20 of the 56 triads in the round body.

4.2.2 In-Place Round Computation

Once the five $D[x]$ values have been formed, the remainder of the round operates directly on the resident state. Our main objective is to avoid constructing a second 25-lane state while still restoring the register assignment of Figure 2 at the end of the round.

We first combine the application of θ with ρ and π. For each lane, we compute

$$\mathrm{ROL}\left(A[x, y] \oplus D[x],\, r[x, y]\right)$$

and write the result directly into the home register of its destination under π, which moves $A[x, y]$ to position $[y,\, 2x + 3y]$ with indices taken modulo five. The XOR completes θ, the rotation performs ρ, and the choice of destination register realizes π. Each lane therefore costs one XOR and one rotation, and the three transformations require no intermediate copy of the complete state.

The 24 lanes other than $A[0, 0]$ form a single cycle under π.

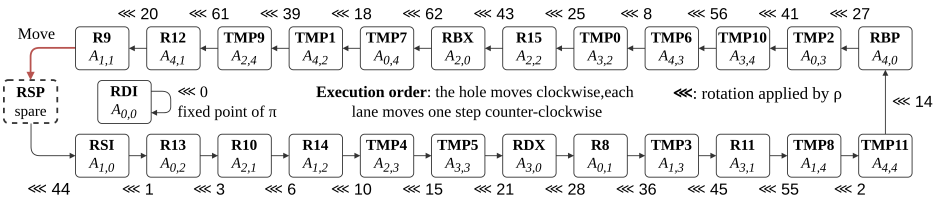

Figure 3: In-place $\rho\pi$ traversal. The 24 non-fixed lanes form a single π cycle, which is opened with one temporary move and then updated directly into destination registers.

Figure 3 shows how this cycle maps onto our register allocation. Following the arrows gives the order in which register contents are moved through the cycle, while the values beside each lane give the rotation applied by ρ. Writing a lane into its destination register would overwrite a lane that has not yet been processed, so we first copy $A[1, 1]$ from `R9` into `RSP` to start the cycle. We then traverse the cycle against the direction of π. Each step reads the lane whose destination is the register freed by the previous step, applies θ and ρ, and writes the result into that register. The first step writes the transformed $A[4, 1]$ from `R12` into `R9`, and the last step writes the transformed $A[1, 1]$ from `RSP` into `RSI`, the

home register of $A[1,0]$, which closes the cycle. Since every lane is read before its register is overwritten, the traversal needs one move, 24 XORs, and 24 rotations. The move shares a triad with the last operation of θ and with the XOR of the fixed point $A[0,0]$ described below. The 24 XORs and 24 rotations then fill the next 16 triads without an empty slot. When the cycle is complete, each transformed lane resides in the home register of its new position, so the register layout required by χ is recovered without a separate rearrangement step. The fixed point $A[0,0]$ does not participate in the cycle. ρ leaves it unrotated, so it needs only the XOR with $D[0]$, which we issue in the same triad that opens the cycle.

After $\rho\pi$, the five $D[x]$ values are dead, so we reuse their registers for χ. Writing B for the state after $\rho\pi$, χ computes for each row

$$A[x,y] \leftarrow B[x,y] \oplus ((\neg B[x+1,y]) \wedge B[x+2,y]) .$$

Updating a row one lane at a time would overwrite lanes that the terms of neighboring lanes still need. We therefore first compute all five nonlinear terms of a row into `RAX` and `TMP12` through `TMP15` while the row is still intact, and then XOR each term into its lane. The row update thus needs no register moves and no memory accesses. Goldmont provides the `μNOTAND` operation, which computes $(\neg a) \wedge b$ and therefore produces each nonlinear term of χ in a single operation. Each row costs five `μNOTAND` and five XOR operations, and χ occupies 17 triads per round. This removes the need for the lane complementing technique [BDP+12], which some architectural implementations use to reduce the number of NOT instructions in χ.

Finally, ι XORs the round constant into $A[0,0]$, which resides in `RDI`. The 24 round constants are stored consecutively in the working buffer. The current position in this table also identifies the round, so no separate round counter is required. We keep this position in the working buffer as a byte offset from `RCX`, starting at 128, the offset of the first constant. At the end of the round we load the position into `TMP14`, which χ has released, use it directly as the index of the constant load, advance it by 8 bytes, and store it back. The state then again follows the register assignment in Figure 2. In total, a round issues 18 memory operations, none of which access the state.

4.2.3 Round Repetition and Patch Interface

The last triad of the round body compares the table position with the end of the table, at byte offset 320, using an XOR, and in the same triad executes the conditional microcode branch `μUJMPCC_DIRECT_NOTTAKEN_CONDNZ_RI`. Its name describes a microcode jump on a condition code (`μUJMPCC`) to a target address encoded in the instruction (`μDIRECT`), in its not taken variant (`μNOTTAKEN`), that branches when the zero flag is clear (`μCONDNZ`) and takes a register and an immediate operand (`μRI`). Our branch experiments showed that the comparison and the branch must share a triad for the branch to observe the comparison result. While the position has not reached the end of the table, the branch returns to `0x7c68`, the first triad of the round body. After the 24th round the comparison yields zero, and execution falls through to the exit sequence.

As described in Section 2.1, entering a patch requires redirecting the microcode entry of an existing macro-instruction to patch RAM. We use the complex instruction, `vmwrite`, for this purpose. We redirect its microcode entry at `0x0cd8` to our patch at `0x7c00`. The software wrapper therefore invokes the permutation by executing `vmwrite` with the working buffer base in `RCX`.

4.3 Performance Evaluation

We evaluate the kernel in three steps. First, we compare the complete permutation against every native implementation that runs on this core. We then measure how much each

design choice of Sections 4.2.1 to 4.2.3 contributes to the result. Finally, we measure whether the advantage survives inside a complete SHA3-256 computation.

Functional Validation. We validate the installed Keccak $f[1600]$ patch through known answer tests and differential comparisons with a separate C reference. The known answer tests run the patch on the published Keccak team test vectors [BDPV] and compare all 25 output lanes against the published output states. The expected states are fixed constants, so the C reference plays no part in computing them.

We additionally compare the microcode output with the C reference 1000 pseudorandom states. Each comparison checks the complete 1600-bit output after all 24 rounds. The tests execute the installed patch on the target processor and exercise the state transfers, repeated round body, and round control.

Constant-Time Behavior. The kernel preserves the input independent control flow and addressing of Keccak $f[1600]$. The patch contains one conditional branch, which tests the public round constant cursor. The same cursor addresses the only indexed load. Every state lane is accessed through a fixed displacement, and the patch image fixes every rotation offset. The control flow and memory addresses therefore do not depend on the state values. Our constant-time reasoning assumes that the underlying micro-operations have latency independent of their operand values.

Benchmark Setup. We compare our microcode implementation against the available Keccak $f[1600]$ implementations that execute on the Goldmont N3350. From SUPERCOP [BL], we include the Keccak team's optimized implementations `opt64*`, `sseu2`, and `mmxu1` [BDP+12], and Van Keer's implementations `x86_64_asm`,`x86_64_shld`, and `simple`. We also include the x86-64 assembly from OpenSSL [Ope26], the formally verified port of that assembly in s2n-bignum [Ama26], and the 64-bit implementations `xkcp_g64` and `xkcp_g64lc` from XKCP [BDH+26]. For each implementation, we evaluate the compiler and optimization configurations used in our benchmark suite and report the configuration with the lowest median cycle count. We measure the complete 24-round permutation, including the interface required to invoke the microcode patch. All benchmarks run on core 1 with Turbo-Boost disabled and the core fixed at its 1100 MHz base frequency, which we verify under load using the `APERF/MPERF` counters.

Table 1 reports the resulting cycles per permutation. The relative cost is the native median divided by the microcode median, so values above one indicate that the microcode implementation is faster.

Table 1: Performance of Keccak $f[1600]$ permutations.

Implementation	Best config	Median	Rel. cost
microcode	Clang 14-O3	**1900**	**1.000×**
opt64lcu24	GCC 11-Os	2051	1.079×
openssl	GCC 11-Os	2054	1.081×
xkcp_g64lc	GCC 11-Os	2057	1.083×
opt64lcu6	Clang 14-O3	2064	1.086×
x86_64_asm	GCC 11-Os	2072	1.091×
s2n_bignum	Clang 17-O3	2107	1.109×
xkcp_g64	GCC 11-Os	2181	1.148×
opt64u6	GCC 11-Os	2258	1.188×
simple	GCC 11-Os	2331	1.227×
sseu2	Clang 14-O3	2618	1.378×
mmxu1	GCC 13-Os	3440	1.811×
opt64lcu24shld	Clang 17-Os	10081	5.306×
x86_64_shld	GCC 13-O	10129	5.331×

The microcode kernel requires 1900 cycles per permutation. The fastest native implementation, `opt64lcu24`, requires 2051 cycles, followed closely by OpenSSL at 2054 cycles

and `xkcp_g64lc` at 2057 cycles. The microcode kernel therefore requires 7.36% fewer cycles than the fastest native implementation, corresponding to a 1.079× speed-up.

The advantage is consistent across several optimized native implementations. The SUPERCOP `opt64` implementations and XKCP share parts of the same Keccak implementation lineage, while OpenSSL provides a separately developed scalar assembly implementation and reaches similar performance. The microcode kernel outperforms all of them, suggesting that the result is not specific to a single native implementation strategy.

4.3.1 Speed-up Analysis

The preceding experiment shows that the complete kernel outperforms every native implementation, but not which design choices produce that result. To answer this question, we build four variants of the complete kernel and compare them in Table 2. Each variant undoes exactly one design choice and keeps everything else unchanged, including the register allocation, the packing rules, and the loop structure. The additional cycles that a variant requires therefore measure the contribution of that single choice to the complete kernel. We describe next the variants we compare against.

Without resident D. In θ, every lane is combined with one of five column values $D[0], \ldots, D[4]$. The complete kernel computes the five values once per round and keeps them in registers while it updates all 25 lanes. This is possible because microcode provides 32 registers, enough to hold the 25 lanes and the five D values at the same time, whereas x86-64 provides 16 general-purpose registers. The variant computes D in the same way but stores it to memory and reloads the required value before each of the 25 lane updates. This adds 30 memory operations per round.

Without `μNOTAND`. χ computes $(\neg b) \wedge c$ 25 times per round. Microcode provides the `μNOTAND` micro-operation, which computes this expression in one operation. Goldmont does not implement BMI1 and therefore lacks the corresponding `ANDN` instruction. The variant uses the identity $(\neg b) \wedge c = (b \oplus c) \wedge c$, which requires two operations. This adds 25 operations per round.

Without move-free χ. χ replaces each lane of a row with a function of that lane and next two lanes in the row, so the original values of the row must remain available until all five new values are computed. The complete kernel first writes the five `μNOTAND` results to five scratch registers and then combines them with the lanes, so χ never copies a lane. These scratch registers are the ones that held D, which become free once θ completes. The variant uses three scratch registers instead of five, so it must copy two lanes of each row before updating the row in place. This adds ten operations per round, and the variant still uses `μNOTAND`.

Without the round-constant cursor. ι combines one lane with a round constant that the kernel loads from a table in memory. In the complete kernel, the loop counter holds the byte offset of the next constant, so the kernel loads the constant directly and advances the counter by eight in each round. The variant counts rounds instead, so it must compute the table address with a shift and an addition before it can load the constant. This adds two operations per round and leaves the patch at 108 triads.

Measurement. Before timing, every variant must reproduce the published Keccak-$f[1600]$ states $f(0)$ and $f(f(0))$ on hardware, so every reported cycle count belongs to a correct permutation. We time all five kernels in one process. In each of 200 repetitions, we install each kernel in turn and time 1000 consecutive permutations, and we rewrite the patch RAM outside the timed region. Table 2 reports the results.

Results. Register residency and `μNOTAND` design choices contribute the most to our speed-up. Without resident D, the kernel requires 292 additional cycles (+15.4%), and

Table 2: Cost of removing each design choice from the Keccak kernel.

Kernel	Triads	Median	Extra cycles	Slowdown
Complete kernel	**108**	**1895**	n.a.	n.a.
Without resident D	122	2187	+292	+15.4%
Without `μNOTAND`	117	2133	+238	+12.6%
Without move-free χ	112	2036	+141	+7.4%
Without the round-constant cursor	108	1963	+68	+3.6%

without `μNOTAND`, it requires 238 additional cycles (+12.6%). Both variants are slower than the fastest native implementation, `opt64lcu24`, which requires 2051 cycles. The kernel therefore needs both choices to outperform native code, and both choices depend on what microcode exposes: residency requires more registers than the 16 architectural ones, and Goldmont provides no instruction equivalent to `μNOTAND`.

The move-free χ and the round-constant cursor contribute less, with 141 cycles (+7.4%) and 68 cycles (+3.6%). The kernel still outperforms the fastest native implementation without either of them.

The cursor shows that patch size does not predict execution time. The variant without the cursor has the same 108 triads as the complete kernel, because its two additional operations occupy otherwise empty slots. Nevertheless, it requires 68 additional cycles, about 2.8 cycles per round, because the load of the round constant must wait for the shift and the addition. In the opposite direction, the variant without resident D executes 14 additional triads per round, 336 per permutation, but requires only 292 additional cycles.

4.4 End-to-End SHA3-256

We next evaluate whether the advantage of the microcode permutation remains visible in a complete SHA3-256 computation, where the permutation executes together with absorption, padding, and digest extraction.

Each native contender uses its own complete SHA3-256 implementation, so every native result includes the library's own sponge around its own permutation. The SUPERCOP implementations use the sponge of their `keccakc512` package, which is the Keccak submission with rate 1088 and capacity 512, and we add the FIPS 202 domain separation bits to it. The `simple` implementation uses its own hash function with the padding byte changed from `0x01` to `0x06`. XKCP uses its `SHA3_256` function, and OpenSSL uses its sponge over its assembly absorb and squeeze routines. Apart from the domain separation, we do not modify any native sponge. s2n-bignum provides only the permutation, so it does not appear in this comparison.

The microcode patch implements only Keccak-f[1600], so we wrap it in a one-shot SHA3-256 sponge written in C. This sponge zeroes the state, XORs each 136-byte block into the 17 rate lanes of the patch buffer before firing the patch, pads the final block with `0x06` and `0x80`, and copies the first four lanes out as the 32-byte digest. Each firing uses the same code as the permutation benchmark of Table 1, so the cost of entering the patch is included in every measurement. Because the sponge computes only SHA3-256, the rate and the digest length are compile-time constants, and the whole message is passed in a single call. The native sponges are general-purpose library code. XKCP takes the rate, capacity, and suffix as run-time parameters and processes the message through separate initialization, absorption, and extraction routines. The SUPERCOP sponge accepts bit lengths and buffers partial input in a queue, and OpenSSL buffers input in a hash context. We quantify the effect of this difference at the end of this section.

Each native implementation is compiled using the configuration that gives its best permutation result in Table 1. Before timing, and again after the frequency warm-up, every implementation passes SHA3-256 known answer tests at 13 message lengths, including

the padding boundaries at 135, 136, and 137 bytes. During timing, the harness compares the sum of the digests of each timed batch with the expected value, so a single incorrect hash invalidates the run. All implementations run in one process and are interleaved within each repetition, so they share the same frequency and thermal state. We repeat the measurement in three separate processes and report the median of the three run medians. Table 3 reports the results for five message lengths.

Table 3: SHA3-256 performance on the Goldmont N3350 for complete hashes, in cycles per hash. Each native implementation uses its own SHA3-256 sponge around its corresponding Keccak permutation. The microcode patch implements only the permutation and runs inside the one-shot C sponge described in the text.

Implementation	Code (B)	32 B	128 B	1 KiB	4 KiB	16 KiB
Permutations per hash		1	1	8	31	121
microcode	**503 (µcode) + 2349**	**2114**	**2129**	**15848**	**60710**	**236592**
xkcp_g64lc	39883	2320	2342	17213	64695	251612
opt64lcu24	57299	2596	2589	17199	65047	253042
opt64lcu6	17801	2329	2338	17013	65154	253632
x86_64_asm	60506	2540	2534	17246	65567	254535
openssl	1909	2395	2387	17225	65803	256306
xkcp_g64	41707	2442	2464	18196	68559	266568
opt64u6	18288	2766	2758	18792	71345	277242
simple	2390	2537	2528	19215	73997	288065
sseu2	11321	3050	3056	21929	84173	327596
mmxu1	8072	4018	4014	28702	109853	428856
opt64lcu24shld	60968	10387	10398	81443	314936	1225780
x86_64_shld	63361	10538	10552	82075	317572	1231270
Fastest native / microcode		1.097×	1.098×	1.074×	1.066×	1.063×
Microcode cycles per byte		66.06	16.63	15.48	14.82	14.44

The microcode implementation is the fastest at every measured message length. For a 128 B message, which requires one permutation, the fastest native implementation requires 1.098× as many cycles as microcode. The advantage remains 1.074× at 1 KiB, 1.066× at 4 KiB, and 1.063× at 16 KiB.

Figure 4 shows the same comparison in cycles per byte for the strongest native

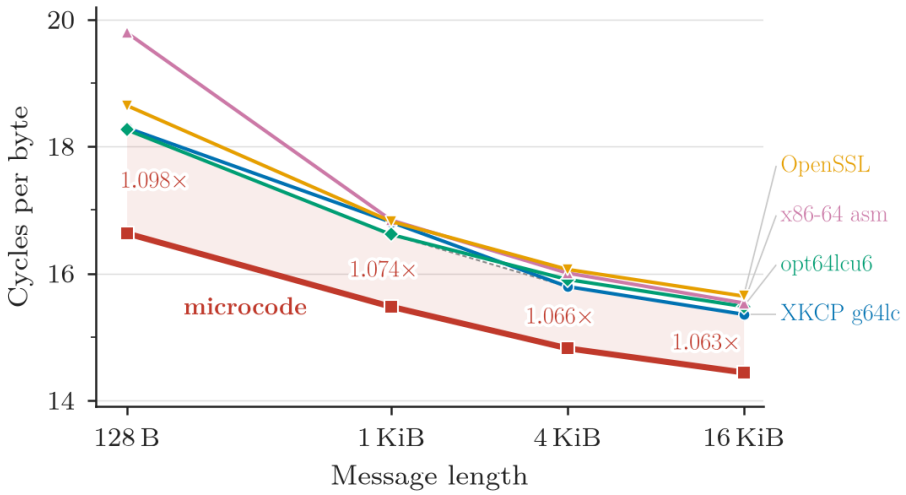

Figure 4: Complete SHA3-256 performance for the microcode implementation and the strongest native baselines. The annotations give the ratio between the fastest native implementation and microcode at each message length.

implementations. The microcode implementation remains below the native implementations across all measured message lengths. Its cost decreases from 16.63 cycles per byte at 128 B to 14.44 cycles per byte at 16 KiB. At 16 KiB, the fastest native implementation, `xkcp_g64lc`, requires approximately 15.36 cycles per byte.

To separate the contribution of the permutation from that of the sponge, we subtract the permutation time of each implementation in Table 1 from its hash time. Since this combines two separate measurements, the resulting figures are approximate. For a 128 B message, the microcode sponge adds approximately 230 cycles to the permutation, compared with approximately 275 cycles for `opt64lcu6`, 285 for `xkcp_g64lc`, 335 for OpenSSL, and 460 to 540 for `x86_64_asm` and `opt64lcu24`. The hash function of `simple` is also a one-shot sponge and adds approximately 200 cycles, so this saving comes from the structure of the sponge and not from microcode. As a result, the speed-up for short messages exceeds the 1.079× of the isolated permutation.

For long messages, the cost per absorbed block dominates, and here our sponge is at a disadvantage. Between 4 KiB and 16 KiB, each additional block costs the microcode implementation approximately 54 cycles beyond the permutation, compared with approximately 20 cycles for `xkcp_g64lc`. XKCP and the SUPERCOP packages XOR each input block into the state inside their permutation routine, and XKCP loads the state once and absorbs consecutive blocks without writing it back to memory. Our sponge instead performs this XOR in memory before each firing of the patch. As the fixed-cost saving is amortized and this per-block cost accumulates, the speed-up decreases from 1.098× at 128 B to 1.063× at 16 KiB.

The advantage of the microcode implementation does not depend on its sponge. If the microcode implementation had the fixed cost of the XKCP sponge instead of its own, a 128 B hash would require approximately 2185 cycles, which is still 1.07× faster than the fastest native implementation. The complete hash results therefore preserve the advantage of the isolated permutation, and the choice of sponge changes its size by less than three percentage points at either end of the measured range.

5 X25519 in Microcode

In this section, we apply our methodology to X25519. We first describe the implementation strategy and then show how established unsaturated 5×51 field arithmetic is mapped to the Goldmont microcode domain. We finally evaluate the resulting implementation against optimized native x25519 implementations.

5.1 Design Strategy

We implement X25519 field multiplication and squaring in microcode while retaining the surrounding Montgomery ladder and inversion logic as architectural code. These two operations provide suitable microcode targets because they dominate the cycle count of the computation. Each ladder iteration invokes five field multiplications and four squarings, while the inversion requires a further 254 squarings and 11 multiplications.

We use the established 5×51 field representation and the five-accumulator organization used by optimized implementations such as OpenSSL's `x25519_fe51_mul` [Pol25]. As in OpenSSL, the limbs are unsigned and may exceed 2^{51} between operations, and subtraction adds $2p$ so that no limb becomes negative. Our objective is not to introduce a new multiplication schedule, but to map this organization to the microcode execution environment. The additional temporary registers available to microcode allow the operands and all five 128-bit accumulators to remain resident in registers simultaneously, so the patch performs no memory accesses.

The main challenge in this mapping is carry propagation within the 128-bit accumulators. Unlike architectural `ADD`, a plain microcode `µADD` does not make its carry available to a subsequent `µADC`. Consequently, the conventional `µADD`/`µADC` idiom cannot be used directly to accumulate 128-bit partial products. We therefore implement and evaluate two microcode-specific carry strategies.

The first uses `µSETCC_CONDB` to materialize the carry produced by a low-word addition. Because this condition state is associated with the destination register, the five accumulators can maintain separate carry chains. The second uses `µADD|MOD2`, which explicitly writes the carry produced by the addition to the architectural carry flag, followed immediately by `µADC` to update the high word. Crucially, `µADD|MOD2` does not consume the previous architectural carry. Each `µADD|MOD2`/`µADC` pair therefore forms an independent two-operation carry dependency rather than extending a carry chain across successive product accumulations.

We use the same accumulator organization for squaring, with a product schedule specialized for equal operands. In either implementation, the multiplication and squaring patches reside in patch RAM simultaneously. We additionally adapt the architectural code around these kernels so that selected short-lived values are passed directly through the register assignments expected by the microcode patches.

5.2 Implementation

This section describes the microcode-specific realization of the multiplication and squaring kernels and their interface with the architectural Montgomery ladder.

5.2.1 Multiplication, Squaring, and Carry Handling

A 5×51 multiplication forms five output sums. A product $a_i b_j$ has weight $2^{51(i+j)}$, so when $i + j \geq 5$ it crosses the 255-bit boundary and is folded into output limb $i + j - 5$ using

$$2^{255} \equiv 19 \pmod{2^{255} - 19},$$

that is, it contributes $19\, a_i b_j$. Since $i + j \geq 5$ implies $j \geq 1$, we precompute $19b_j$ for $j = 1, \ldots, 4$ and multiply a_i by $19b_j$ directly, instead of multiplying each of these ten products by 19.

Following the five-accumulator organization, we maintain one 128-bit accumulator for each output limb. Each accumulator is represented by a low and high 64-bit word, so the five accumulators occupy ten registers. In our multiplication patch these values are held in microcode temporary registers, while the input limbs remain in architectural registers. Four additional temporary registers hold the precomputed values $19b_j$.

We exploit the operand semantics of the microcode multiplication operations when constructing these accumulators. The first product contributing to each output is computed with `µMUL`, which produces the high half in its destination while replacing one source with the low half. We stage this source in the register assigned to the accumulator low word. The first product therefore initializes both halves of the accumulator directly and requires no separate zeroing or accumulation operations. For the operands multiplied by 19, we instead use `µIMUL64L`, whose non-destructive semantics preserve its input operands for subsequent products.

The remaining products require a carry from the addition of the low product word into the corresponding high word. Our two implementations differ only in how this carry is transferred. In the SETCC variant, an accumulation uses

$$\begin{aligned} L_j &\leftarrow L_j + P_{\mathrm{lo}}, \quad c \leftarrow \mathrm{carry}(L_j), \\ P_{\mathrm{hi}} &\leftarrow P_{\mathrm{hi}} + c, \quad H_j \leftarrow H_j + P_{\mathrm{hi}}. \end{aligned}$$

where the second operation is implemented with `µSETCC_CONDB`. This requires four micro-operations for each additional product. The condition state remains associated with the low-word destination, allowing the five accumulator chains to coexist without routing their carries through a single architectural flag.

The second implementation uses the architectural carry state explicitly. A low-word `µADD|MOD2` writes its carry-out to the architectural carry flag, and the immediately following `µADC` incorporates this carry into the high word,

$$L_j \leftarrow L_j + P_{\mathrm{lo}}, \qquad H_j \leftarrow H_j + P_{\mathrm{hi}} + c.$$

A plain microcode `µADD` does not provide this carry to `µADC`. The `MOD2` modifier is therefore required on the first operation. Importantly, `µADD|MOD2` does not consume the previous architectural carry. Each pair consequently begins with a newly produced carry and does not introduce a carry dependency on the preceding product accumulation. This reduces the wide accumulation from four micro-operations in the SETCC implementation to two in the MOD2/ADC implementation.

After the products have been accumulated, we perform radix reduction without introducing a serial carry chain across the five outputs. For each accumulator we compute in parallel

$$r_j = L_j \,\&\, (2^{51} - 1), \qquad q_j = (L_j \gg 51) \mid (H_j \ll 13),$$

where $\gg$ and $\ll$ are logical shifts, so that $q_j\, 2^{51} + r_j$ equals the unsigned accumulator. Each q_j is the carry from limb j into limb $j+1$, and the carry q_4 out of the top limb wraps around to limb 0 with the factor 19:

$$t_0 = r_0 + 19\, q_4, \qquad t_j = r_j + q_{j-1} \quad (1 \le j \le 4).$$

Each t_j fits in a single 64-bit word, and we apply the same split and additions to it once more, which yields output limbs below $2^{51} + 2^{15}$. This organization turns the radix carries into shifts, masks, and ordinary 64-bit additions and allows the five splits of each pass to proceed independently.

Squaring uses the same accumulator representation, carry mechanism, and reduction structure. Symmetry reduces its schedule to 15 distinct products rather than the 25 products required for general multiplication. The SETCC multiplication and squaring patches occupy 58 and 41 triads, respectively. Replacing the SETCC accumulation with `µADD|MOD2/µADC` reduces these sizes to 45 and 31 triads.

5.2.2 Interface with the Montgomery Ladder

The Montgomery ladder remains architectural code, but we arrange selected operations to match the input register assignment of the microcode kernels. Our addition and subtraction routines place their five output limbs in (`RDI`, `RSI`, `R12`, `R11`, `R14`), which is also the input layout used by multiplication and squaring. We use this directly for the chains $A \to AA$, $B \to BB$, $D \to DA$, $C \to CB$, and $DA \pm CB \to$ squaring, avoiding the five input loads that would otherwise precede each kernel invocation.

This keeps the surrounding X25519 structure unchanged while adapting its local dataflow to the register interface of the microcode kernels.

5.2.3 Patch Interface

For X25519, we use two entry points for the field multiplication and squaring patches. We redirect the microcode entry of `vmwrite` at `0x0cd8` to the multiplication patch at `0x7c00`, and the entry of `vmread` at `0x0618` to the squaring patch at `0x7cb4`. The software

wrappers therefore invoke field multiplication with `vmwrite` and field squaring with `vmread`, after placing their operands in the registers expected by the respective patches.

5.3 Performance Evaluation

We evaluate the X25519 case study at two levels. We first measure field multiplication and squaring directly. This isolates the arithmetic implemented in microcode and allows us to compare the two mechanisms we use for handling carries. We then evaluate complete X25519 computations against optimized native implementations. The resulting comparison includes the repeated cost of invoking the microcode patches as well as their interaction with the surrounding Montgomery ladder.

Functional Validation. We validate the implementation against the two test vectors and the 1000-iteration test of RFC 7748 [LHT16]. These tests run complete X25519 computations and therefore reach the installed multiplication and squaring patches together with the Montgomery ladder, conditional swaps, inversion, and encoding. Before timing complete X25519 computations, our benchmark harness applies these checks to every implementation and then checks every timed output against the expected result. It also compares each field kernel against our C implementation on 2000 random multiplications and 2000 random squarings before timing.

Constant-Time Behavior. The Montgomery ladder executes 255 iterations independently of the scalar value. Scalar bits affect only the masks used by the conditional swaps and do not select branches or memory addresses. The multiplication and squaring patches execute fixed sequences of micro-operations. Carry values affect arithmetic state but not control flow or memory accesses. The inversion similarly follows a fixed addition chain with predetermined numbers of multiplications and squarings. Our constant-time reasoning assumes that the compiler preserves these properties in the surrounding architectural code and that the architectural instructions and micro-operations used have operand independent latency.

Benchmark Setup. We compare our patches against the X25519 implementations that execute on the Goldmont N3350. For the field kernels, the comparison includes the `fe51` assembly of OpenSSL [Ope26], the C code generated by fiat-crypto [EPG+19], assembly produced by CryptOpt [KEG+23], which we generate on the N3350 itself, the `amd64-51` assembly from SUPERCOP [BL, BDL+11], and a C implementation that we wrote by hand using `__uint128_t`. For complete X25519 computations, the comparison includes the formally verified `x25519_x25519_alt` assembly of s2n-bignum [Ama26], the X25519 implementation of OpenSSL, and the `amd64-64`, `amd64-51` [BDL+11], and `donna_c64` [Lan08] implementations from SUPERCOP. OpenSSL runs its C ladder over its `fe51` assembly, which is the path it selects on processors without ADX. The remaining SUPERCOP implementations either require instructions that the N3350 lacks, such as AVX, MULX, and ADX, or serve as reference code rather than aiming at high performance. Each microcode variant runs in its own process, which installs the patches once and also times every native implementation. All benchmarks run on core 1 with turbo disabled and the core fixed at its 1100 MHz base frequency, which we verify under load using the `APERF/MPERF` counters.

5.3.1 Field Kernel Performance

Our multiplication and squaring patches use five independent accumulators for the five limbs of the 5×51 representation. We evaluate two realizations of the carry handling within this organization. The first materializes the carry from each low half accumulation using `μSETCC`. This keeps the carry associated with the corresponding accumulator in a register

and allows the five accumulation chains to remain independent. The second exploits the flag behavior exposed by the microcode execution layer. A µADD with MOD2 produces the carry state consumed directly by the corresponding µADC, eliminating the explicit µSETCC and carry addition used by the first realization. We refer to these implementations as the SETCC and ADD|MOD2 variants, respectively.

Table 4: Field operation performance on the Goldmont N3350. Values report median cycles per complete operation in a dependent chain.

Implementation	Multiplication	Squaring
microcode ADD|MOD2	**90.0**	**65.0**
openSSL fe51 assembly	97.1	72.4
microcode SETCC	103.8	72.9
fiat-crypto	127.8	107.1
cryptOpt	141.3	118.8
SUPERCOP amd64-51	145.9	118.8
hand-written __uint128_t C	140.2	104.8

The SETCC implementation is already competitive with highly optimized native arithmetic. Its multiplication requires 103.8 cycles and its squaring requires 72.9 cycles, placing squaring at essentially the same performance as OpenSSL's fe51 implementation.

Using ADD|MOD2 improves both operations substantially. Multiplication falls to 90.0 cycles and squaring to 65.0 cycles. The resulting kernels are faster than every native implementation we evaluate. Compared with OpenSSL, multiplication requires 7.3% fewer cycles and squaring requires 10.2% fewer cycles.

The difference between the two microcode variants is important. Their field representation and high level multiplication and squaring organizations are unchanged. Both exploit five independent accumulators and perform the same reduction. What changes is how the carry from a low half accumulation reaches the corresponding high half. The SETCC realization explicitly materializes this carry before adding it to the high half. The ADD|MOD2 realization communicates the carry through microcode flag state and consumes it directly with µADC.

This reduces both the dependency length and the patch footprint. The multiplication and squaring patches occupy 58 and 41 triads with SETCC, whereas the ADD|MOD2 versions require 45 and 31 triads. The result illustrates a central advantage of treating microcode as a target architecture. An operation that is ordinarily hidden below the ISA can be selected and composed according to the dependency structure of the cryptographic arithmetic.

The microcode invocation itself is included in the values in Table 4. The advantage of the ADD|MOD2 kernels therefore holds despite the additional cost of entering microcode.

5.3.2 End-to-End X25519

We next measure complete X25519 computations. Unlike the kernel experiment, this comparison includes the Montgomery ladder, conditional swaps, inversion, encoding, architectural wrappers, and every transition into and out of the microcode patches.

We evaluate 24 compiler and optimization configurations using GCC 11 through GCC 13 and Clang 14, Clang 17, and Clang 18. Each measurement computes the first RFC 7748 test vector, and each process performs 1000 rounds in which every implementation runs once. For each implementation, Table 5 reports the lowest median obtained across these configurations.

The ADD|MOD2 implementation is the fastest complete X25519 implementation in our evaluation. It requires 225,988 cycles, compared with 243,654 cycles for s2n-bignum and 250,492 cycles for OpenSSL. This corresponds to 7.3% fewer cycles than s2n-bignum, the fastest implementation optimized at ISA level for this microarchitecture, and 9.8% fewer cycles than OpenSSL.

Table 5: End-to-end X25519 performance and code size

Implementation	Code Size (Bytes)	Median cycles
this work, `ADD|MOD2`	22,083	**225,988**
s2n-bignum	21,815	243,654
this work, `SETCC`	32,334	249,405
OpenSSL	3,831	250,492
SUPERCOP `amd64-64`	8,359	271,838
SUPERCOP `donna_c64`	25,999	336,952
SUPERCOP `amd64-51`	11,237	356,432

The two microcode realizations also differ in patch footprint. The `SETCC` multiplication and squaring kernels occupy 99 triads, whereas the `ADD|MOD2` realization requires 76 triads. The latter therefore improves both execution time and patch RAM consumption while retaining the same field representation and five accumulator organization.

The architectural code size reflects a different tradeoff. Our field operation interfaces are inlined into the surrounding X25519 implementation, so operand movement around repeated microcode invocations contributes substantially to the generated x86 code. OpenSSL instead places its field operations in assembly functions that occur once and are reached through calls, resulting in a considerably smaller architectural code footprint.

The x86 size of our implementation is also sensitive to compiler treatment of the inversion sequence. At their respective fastest configurations, the `ADD|MOD2` and `SETCC` implementations occupy 22,083 and 32,334 bytes. When both are compiled with Clang 17 at `-O`, they each occupy 8,473 bytes, while requiring 231,636 and 253,896 cycles respectively. The two variants share the same x86 source and differ only in their patches, so the size difference in the best configurations comes from compiler unrolling rather than from the surrounding X25519 implementation.

6 Conclusion

We study cryptographic implementation below the ISA, treating programmable microcode as a distinct implementation target rather than as a lower-level encoding of existing x86 software. On Intel Goldmont, we characterized the properties of the microcode that matter for cryptographic code and used them to implement Keccak $f[1600]$ and the field arithmetic of X25519. Our Keccak permutation is 1.079× faster than the fastest native implementation, and the advantage persists in complete SHA3-256 computations at every measured message length. Our complete X25519 implementation requires 7.3% fewer cycles than s2n-bignum and 9.8% fewer cycles than OpenSSL. Controlled comparisons show that these gains depend on how the computation is mapped to the registers, operations, and carry mechanisms of the execution layer rather than on execution in microcode alone.

Although systematic, our methodology is manual. We probed µop semantics with targeted experiment, made use of extra registers, packed operations into triads, and chose kernel boundaries against the patch RAM capacity and invocation cost. As future work, we propose to automate this process. A semantic model of the µops would enable a code generator that takes a straight-line description of the computation, such as the output of fiat-crypto [EPG+19], allocates registers across both register files, selects among µop realizations, and packs the result into triads. Such a generator could need to search over candidate schedules using measurements on the target, as CryptOpt does at the ISA level [KEG+23]. The same model would allow generated patches to be checked for equivalence with a reference implementation, replacing validation by testing alone. Such a toolchain would make it practical to extend this study to further primitives and would further establish that the ISA is not the lowest level at which cryptographic implementations can be designed.

Acknowledgments

We would like to thank Anirban Chakraborty and Xinrui Wang for their suggestions and Amin Abdulrahman for verifying the artifact.

This work has been supported by an ARC Discovery Project number DP210102670; by the Dutch Ministry of Education, Culture, and Science through Gravitation project "Challenges in Cyber Security – 024.006.037"; and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and through project number 560392681.

References

[ABB+20] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *IEEE SP*, pages 965–982, 2020. doi:10.1109/SP40000.2020.00028. 7

[Ama26] Amazon Web Services. s2n-bignum. https://github.com/awslabs/s2n-bignum, 2026. Snapshot distributed with SUPERCOP 20260330. Keccak-f[1600] from `x86/sha3/sha3_keccak_f1600.S`, a formally verified (HOL Light) port of Andy Polyakov's OpenSSL code. 12, 19

[BDH+26] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. XKCP: the eXtended Keccak Code Package. https://github.com/XKCP/XKCP, 2026. Commit 78477d2e, 12 June 2026. 12

[BDL+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *CHES*, pages 124–142, 2011. doi:10.1007/978-3-642-23951-9_9. 5, 6, 19

[BDP+12] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview. Version 3.2, May 2012. URL: https://keccak.team/files/Keccak-implementation-3.2.pdf. 8, 11, 12

[BDPV] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak-f[1600] intermediate values. https://github.com/XKCP/XKCP/blob/master/tests/TestVectors/KeccakF-1600-IntermediateValues.txt. XKCP test vectors, accessed 30 June 2026. 12

[BDPV13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 313–314. Springer, Berlin, Heidelberg, May 2013. doi:10.1007/978-3-642-38348-9_19. 4

[BDPVA14] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The making of Keccak. *Cryptologia*, 38(1):26–60, January 2014. doi:10.1080/01611194.2013.856818. 4

[Ber] Daniel J. Bernstein. lib25519: A microlibrary for the X25519 and Ed25519 primitives. URL: https://lib25519.cr.yp.to/. 5

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Berlin, Heidelberg, April 2006. doi:10.1007/11745853_14. 5

[BES+23] Pietro Borrello, Catherine Easdon, Martin Schwarzl, Roland Czerny, and Michael Schwarz. CustomProcessingUnit: Reverse engineering and customization of Intel microcode. In *WOOT*, pages 285–297, May 2023. doi:10.1109/SPW59333.2023.00031. 1, 3, 7

[BL] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. SUPERCOP release 20260330. URL: https://bench.cr.yp.to/supercop/supercop-20260330.tar.xz. 12, 19

[Cha82] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *CC*, pages 98–105, 1982. doi:10.1145/800230.806984. 6

[EG18] Mark Ermolov and Maxim Goryachy. IntelTXE-PoC: Intel Management Engine JTAG proof of concept. https://github.com/ptresearch/IntelTXE-PoC, 2018. Positive Technologies. Archived repository, accessed September 2026. 6

[EPG+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In *IEEE SP*, pages 1202–1219, 2019. doi:10.1109/SP.2019.00005. 19, 21

[ESG21] Mark Ermolov, Dmitry Sklyarov, and Maxim Goryachy. uCodeDisasm: Intel Atom Goldmont microcode disassembler. https://github.com/chip-red-pill/uCodeDisasm, 2021. GitHub repository, accessed October 2026. 6

[ESG23] Mark Ermolov, Dmitry Sklyarov, and Maxim Goryachy. Undocumented x86 instructions to control the CPU at the microarchitecture level in modern Intel processors. *Journal of Computer Virology and Hacking Techniques*, 19(3):351–365, September 2023. doi:10.1007/s11416-022-00438-x. 1, 3, 6, 7

[Fal22] Chris Fallin. Cranelift, part 4: A new register allocator. https://cfallin.org/blog/2022/06/09/cranelift-regalloc2/, 2022. 7

[Int23] Intel Corporation. *Intel Advanced Performance Extensions (Intel APX) Architecture Specification*, 2023. Document number 355828. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-performance-extensions-apx.html. 6

[KEG+23] Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom. CryptOpt: Verified compilation with randomized program search for cryptographic primitives. In *PLDI*, pages 1268–1292, 2023. doi:10.1145/3591272. 6, 19, 21

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00002. 3

[KKF+17] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. Reverse engineering X86 processor microcode. In *USENIX Security*, pages 1163–1180, 2017. 1, 3

[KKF+18] Benjamin Kollenda, Philipp Koppe, Marc Fyrbiak, Christian Kison, Christof Paar, and Thorsten Holz. An exploratory analysis of microcode as a building block for system defenses. In *CCS*, pages 1649–1666, October 2018. doi:10.1145/3243734.3243861. 1

[Kro23] Alexander Krog. lib-micro. https://github.com/zanderdk/lib-micro, 2023. 1, 3, 7

[Lan08] Adam Langley. curve25519-donna. https://github.com/agl/curve25519-donna, 2008. 64-bit C implementation, distributed with SUPERCOP 20260330 as `donna_c64`. 19

[LHT16] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security. RFC 7748, January 2016. doi:10.17487/RFC7748. 19

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 973–990. USENIX Association, August 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp. 3

[Nat15] National Institute of Standards and Technology. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical Report FIPS PUB 202, U.S. Department of Commerce, August 2015. doi:10.6028/NIST.FIPS.202. 4

[OGGF12] Erdinc Ozturk, James Guilford, Vinodh Gopal, and Wajdi Feghali. New instructions supporting large integer arithmetic on Intel architecture processors. White Paper 327831-001, Intel Corporation, August 2012. URL: https://github.com/intel/intel-ipsec-mb/wiki/doc/ia-large-integer-arithmetic-paper.pdf. 7

[Ope26] OpenSSL: General purpose TLS and crypto library. https://github.com/openssl/openssl, 2026. Commit 43e1e98a (4.1.0-dev), June 2026. Keccak-f[1600] from `crypto/sha/asm/keccak1600-x86_64.pl`, written by Andy Polyakov. 12, 19

[Pol25] Andy Polyakov. OpenSSL 3.5.0: Curve25519 base-2^{51} field multiplication. `crypto/ec/curve25519.c`, function `fe51_mul`, 2025. Base-2^{51} implementation originally introduced in OpenSSL commit cfc32a1efb464205885e18e503bcb7051c307008. URL: https://github.com/openssl/openssl/blob/openssl-3.5.0/crypto/ec/curve25519.c. 16

[WNR25] Yubo Wang, Ruslan Nikolaev, and Binoy Ravindran. reInstruct: Toward OS-aware CPU microcode reprogramming. In *KISV*, pages 10–16, October 2025. doi:10.1145/3765889.3767043. 1