Optimal software-implemented Itoh–Tsujii inversion for $\mathbb{F}_{2^m}$

Jeremy Maitin-Shepard

jbms@cs.berkeley.edu
University of California at Berkeley

Abstract

Field inversion in $\mathbb{F}_{2^m}$ dominates the cost of modern software implementations of certain elliptic curve cryptographic operations, such as point encoding/hashing into elliptic curves. [7, 6, 2] Itoh–Tsujii inversion using a polynomial basis and precomputed table-based multi-squaring has been demonstrated to be highly effective for software implementations [19, 14, 2], but the performance and memory use depend critically on the choice of addition chain and multi-squaring tables, which in prior work have been determined only by suboptimal ad-hoc methods and manual selection. We thoroughly investigated the performance/memory tradeoff for table-based linear transforms used for efficient multi-squaring. Based upon the results of that investigation, we devised a comprehensive cost model for Itoh–Tsujii inversion and a corresponding optimization procedure that is empirically fast and provably finds globally-optimal solutions. We tested this method on 8 binary fields commonly used for elliptic curve cryptography; our method found lower-cost solutions than the ad-hoc methods used previously, and for the first time enables a principled exploration of the time/memory tradeoff of inversion implementations.

1 Introduction

Field inversion in $\mathbb{F}_{2^m}$ is a key primitive required by elliptic curve cryptographic operations over binary elliptic curves. On older CPU architectures, the total cost of field multiplications typically dominates the cost of all other operations due to the inefficiency of performing polynomial multiplication without native instruction support, with the cost of field inversion only about 3–8 times the cost of field multiplication. [8, 19] The fast carry-less multiplication in modern CPU architectures, such as the `PCLMULDQ` instruction supported by Intel's Westmere and later processors, the ARMv7 `VMULL.P8` instruction, and the ARMv8 `PMUL`/`PMULL`/`PMULL2` instructions, reduces the cost of multiplication by a factor of 20 to 40, but does not significantly reduce the cost of inversion. Consequently, for applications such as point encoding/hashing into elliptic curves [7, 6, 2], field inversion can account for more than half of the total computational cost in a software implementation on modern hardware, particularly when using binary fields of large degree. A reduction in the cost of field inversion can, therefore, have a significant effect on the overall cost. For this reason, we investigated the optimal software implementation of field inversion for arbitrary binary fields $\mathbb{F}_{2^m}$.

Our contribution: Based on the Itoh–Tsujii polynomial-basis method [19, 9], apparently the fastest method for binary field inversion in software, we developed an empirically-fast algorithm for searching over the space of field inversion implementations in order to provably minimize computational cost subject to an arbitrary bound on, or cost function for, memory use. This algorithm, based on a CPU architecture-specific cost model estimated from performance

benchmarks, allows us to compute a time/memory Pareto frontier of inversion implementations for a given binary field and CPU architecture, and has allowed us to find lower-cost implementations for commonly-used binary fields than were obtained by the ad-hoc methods [19, 14] employed previously.

2 Itoh–Tsujii inversion

The inverse x^{-1} of an element $x \in \mathbb{F}_{2^m}^{\times}$ can be computed using the Itoh–Tsujii method [11] as follows: by Lagrange's theorem, $x^{2^m-1} = 1$, which implies that $x^{2^m-2} = x^{-1}$. To perform this field exponentiation efficiently, an *addition chain* is used.

Definition 1. *A sequence* $\langle a_1, b_1\rangle, \ldots, \langle a_k, b_k\rangle$ *is said to be an* addition chain *for* p *if* $a_k = p$ *and* $\{b_i, a_i - b_i\} \subseteq \{a_j \mid j \leq i\} \cup \{1\}$.

Given a valid addition chain $\langle a_i, b_i\rangle_{i\in[1,k]}$ for $m-1$, the Itoh–Tsujii method iteratively computes

$$x^{2^{a_i}-1} = (x^{2^{a_i-b_i}-1})^{2^{b_i}} \cdot x^{2^{b_i}-1} \qquad \text{for } i = 1, \ldots, k; \tag{1}$$

$$x^{-1} = x^{2^m-2} = (x^{2^{m-1}-1})^2. \tag{2}$$

The two operations required by eq. (1) are the raising of the previously-computed term $x^{2^{a_i-b_i}-1}$ to the power 2^{b_i}, which is called multi-squaring [19] or m-squaring [1, 5], and the multiplication of the newly computed $(x^{2^{a_i-b_i}-1})^{2^{b_i}}$ term by the previously computed $x^{2^{b_i}-1}$ term. Equation (2) requires only a single squaring operation.

When using a polynomial basis [19, 9], multi-squaring to the power 2^b can be performed in one of two ways:

1. **directly** as a sequence of b repeated squarings;
2. using a **lookup table**, as described in section 4, by taking advantage of the fact that squaring, and therefore multi-squaring to any fixed power, is linear in the binary coefficients. [5, 19, 14, 2] This can be faster than b repeated squarings for sufficiently large values of b.

Note that any addition chain $\langle a_i, b_i\rangle_{i\in[1,k]}$ has a complementary addition chain $\langle a_i, a_i - b_i\rangle_{i\in[1,k]}$; while for some purposes no distinction need be made between these two chains, in the case of Itoh–Tsujii inversion the distinction is necessary because the two chains lead to a different sequence of operations.

3 Optimal addition chains

Prior work has demonstrated software implementations of Itoh–Tsujii inversion with multi-squaring tables to be highly efficient.[19, 14, 2] The cost depends critically, however, on the choice of addition chain and multi-squaring tables. The key problem we investigated is the optimization of the choice of addition chain and multi-squaring tables to minimize this cost, for arbitrary binary fields $\mathbb{F}_{2^m}$.

In the case that only direct squaring is used (no precomputed multi-squaring tables), Itoh–Tsujii inversion using an addition chain of length k requires exactly k field multiplications and m squarings. Minimizing the cost is simply a matter of finding a minimum-length addition chain for $m-1$, and does not depend on the actual cost of multiplication and squaring. (There are typically many minimum-length chains for a given value of m.) There is no known polynomial-time algorithm for finding minimum-length addition chains, but there are search algorithms

based on various pruning criteria [20, 3] that are very effective in practice for reasonable values of m.[1]

The use of precomputed multi-squaring tables, crucial to achieving good performance, as shown in section 7, greatly complicates the optimization problem: because we may use either direct squaring or table-based multi-squaring for each step of the addition chain, we must consider the combined cost of the multiplications, squarings, and table lookups required; minimizing the length of the chain only minimizes the multiplication cost, which is typically less than half of the total cost. We must also consider that the table lookup cost is not fixed but indirectly depends on the total number of multi-squaring tables used, as described in section 4. Furthermore, we must consider not only the computational cost but also the memory use of the precomputed tables; it may be desirable to trade off some performance for a reduced memory footprint. The optimal solution clearly *does* depend on the actual costs of multiplication, squaring, and table lookups, which depend on the CPU architecture and their particular implementation. Our measurement of the computational costs of these operations is described in section 5.

We devised a comprehensive architecture-specific cost model based on these measurements; we also developed an A* search procedure for efficiently optimizing this cost model, as described in section 6. As the prior work on pruning criteria for finding minimum-length addition chains is not directly applicable to our more complicated cost model, we defined a novel efficiently-computable search heuristic that does not sacrifice optimality under our cost model.

4 Table-based linear field operations

Several key operations for $\mathbb{F}_{2^m}$, such as squaring, multi-squaring, square root, and half-trace, are linear in the binary coefficients. For multi-squaring (useful for inversion) and half-trace, an implementation based on a lookup table can be significantly faster than direct computation.[5, 19, 14, 2] The coefficients are split into $\lceil m/B \rceil$ blocks of B bits, and a separate table of 2^B entries is precomputed for each block position, using a total of $s_{m,B} = \lceil m/B \rceil \cdot 2^B \cdot \lceil m/W \rceil \cdot W/8$ bytes of memory, where W is the word size in bits. The linear transform can then be computed from the precomputed tables with $k = \lceil m/B \cdot \lceil m/W \rceil$ memory accesses and $k - 1$ XOR operations.

Reasonable values of B range from 4 to 16, with larger values reducing the number of memory accesses required per transform at the cost of greater memory use. Due to the nature of the cache/memory hierarchy, however, memory access latency and throughput depend significantly on the extent to which the target is cached; this effect can be sufficiently large to outweigh the reduction in memory accesses required with larger values of B. To optimize table use we must consider, therefore, the amount of cache memory expected to be available, which depends in particular on the total size of all lookup tables required concurrently.

5 Performance evaluation

In order to measure the costs of low-level field operations (namely multiplication, squaring, and multi-squaring), and also to evaluate field inversion implementations, we developed an optimized implementation of binary field operations as an open-source C++ library. [12] Through the use of C++ templates, we were able to write much of the code generically to support different field sizes without sacrificing runtime performance; only for modular reduction was a custom implementation required for each supported field. We incorporated existing fast x86/x86-64 polynomial multiplication, squaring, and modular reduction routines for $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{193}}$, $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{239}}$, $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{409}}$, $\mathbb{F}_{2^{571}}$ [4] and for $\mathbb{F}_{2^{127}}$ [14]. These fields are sufficient for implementing

[1] As we are primarily concerned with elliptic curve cryptography, binary field sizes of interest range from about $m = 127$ to about $m = 571$.

elliptic curve operations for all NIST-recommended binary elliptic curves [13] as well as several other SEC 2-recommended binary elliptic curves [16]. $\mathbb{F}_{2^{127}}$ is needed to implement the record-breaking GLS254 curve [14] (based on the quadratic extension field $\mathbb{F}_{2^{254}}$).

As our test platforms we used an Intel Westmere i7-970 3.2 GHz CPU (with 12 MiB L3 cache) and an Intel Haswell i7-4790K 4.0 GHz CPU (with 8 MiB L3 cache). Both of these processors support the `PCLMUL` instruction for carry-less multiplication, Westmere being the first Intel architecture to support it; on the much more recent Haswell architecture, where this instruction has significantly lower cost, alternative modular reduction routines based on it are used for $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{283}}$, and $\mathbb{F}_{2^{571}}$ for a modest gain in performance.[4] All code was compiled separately for each architecture using version 3.5 of the Clang compiler at the highest optimization level.

5.1 Robust timing

We measured the execution time of all operations in CPU cycles, using the combination of `RDTSC`, `RDTSCP`, and `CPUID` instructions recommended by Intel. [15] To improve accuracy and reduce variance, we disabled turbo boost, frequency scaling, and hyper-threading, and ensured that a single non-boot CPU core was used for all benchmarks on each machine. For each operation, we estimated the benchmarking overhead and subtracted it from the measured number of cycles. Additionally, we automatically determined a per-measurement repeat count for each operation that ensured the benchmarking overhead was less than 10%.

The execution time was computed as the median of the cycle measurements; the number of cycle measurements for each operation from which the median was computed was at least 1000 and chosen automatically to ensure a sufficiently small 99% confidence interval on the median estimate (less than the larger of $1/1000$ of the estimated median or $1/10$ of a cycle). For consistency, we ensured warm-cache conditions for all estimates by discarding the first 2000 measurements.

5.2 Multiplication and squaring performance

We initially measured the execution times of field multiplication and squaring individually, but found that for small-degree fields, the measurements did not coincide with timings of sequences of repeated squarings and multiplications, as are required by field inversion. Attributing this discrepancy to compiler instruction reordering and pipelined and out-of-order execution by the CPU, we jointly estimated the cost of multiplication and squaring using linear regression on execution time measurements of b repeated squarings followed by a single multiply, for $b \in [1, 10]$, and used these estimates instead of the direct measurements for the field inversion cost model. The actual measurements and estimates are shown in table 1.

5.3 Table-based operation performance

Since table-based multi-squaring, and field inversion routines based on it, induce data-dependent memory accesses, we measured the aggregate execution time for a set of inputs guaranteed to induce a uniform memory access pattern (and then divided by the number of inputs), in order to obtain worst-case warm-cache estimates. Failure to do so would result in a large underestimate of execution time due to caching.

We also observed the performance characteristics of table operations to be significantly affected by the size of the virtual memory pages backing the tables; in particular, on the x86-64 test machines, both the base level performance and the scaling of execution times with increasing table size were significantly better with 2 MiB (huge) pages than with 4 KiB pages, as shown in fig. 1, due to the cost of translation lookaside buffer (TLB) misses. The Linux transparent huge page support (introduced in Linux version 2.6.38) results in some, but not all, memory

Table 1: Multiplication and squaring performance for $\mathbb{F}_{2^m}$. Performance was measured independently (ind.) for each operation, and also estimated jointly by linear regression from measurements of repeated squarings followed by a single multiply.

	Westmere cycles		Haswell cycles	
Field	Multiply Ind./Joint	Square Ind./Joint	Multiply Ind./Joint	Square Ind./Joint
$\mathbb{F}_{2^{127}}$	34 / 45	6 / 11	12 / 23	4 / 9
$\mathbb{F}_{2^{163}}$	83 / 83	29 / 33	36 / 46	17 / 26
$\mathbb{F}_{2^{193}}$	108 / 113	22 / 25	41 / 45	16 / 22
$\mathbb{F}_{2^{233}}$	108 / 114	24 / 29	41 / 45	18 / 26
$\mathbb{F}_{2^{239}}$	120 / 123	39 / 33	55 / 54	30 / 33
$\mathbb{F}_{2^{283}}$	152 / 149	36 / 37	56 / 59	24 / 30
$\mathbb{F}_{2^{409}}$	276 / 272	36 / 35	97 / 96	29 / 32
$\mathbb{F}_{2^{571}}$	418 / 426	67 / 66	152 / 175	41 / 41

regions being backed automatically by huge pages, depending on a number of factors including region alignment and physical memory fragmentation; when not taken into account, this significantly reduced the reliability of our performance measurements. For consistent performance, we therefore ensured that all lookup tables were backed by huge pages.

We estimated the cost of table-based multi-squaring for each block size $B \in \{4, 6, 8, 10, 12, 16\}$ and for varying values $n \in [1, 6]$ of the number of concurrent lookup tables in use by timing n successive table-based linear transforms (using n separate tables), and then dividing by n. Pareto-optimal points with respect to execution time and memory use are shown in fig. 2. Our implementation used a word size of $W = 128$ bits.

6 A* search procedure

Recall that in order to invert an element $x \in \mathbb{F}_{2^m}$, Itoh–Tsujii inversion requires an addition chain $\langle a_1, b_1\rangle, \ldots, \langle a_k, b_k\rangle$ where $\{b_i, a_i - b_i\} \subseteq \{a_j \mid j \leq i\} \cup \{1\}$, and $a_k = m - 1$. We modeled the cost based on the following assumptions:

1. Each step of the addition chain incurs a fixed cost of c_{mul} for the required multiplication.
2. There is a cost of $b_i \cdot c_{\text{sq}}$ for each direct multi-squaring to the power 2^{b_i}.
3. If table-based multi-squaring is used in place of direct multi-squaring, there is a cost $\tau_{B,n}$ that depends on the table block size B and the number of distinct multi-squaring tables $n = |T|$ (where T is the set of distinct values b_i for which a multi-squaring table is used).

We impose the minor restriction that the same block size B is used for all multi-squaring tables. In addition to the computational costs c_{mul}, c_{sq}, and $\tau_{B,n}$, we assume that there is a non-decreasing cost $c_{\text{mem}}(s)$ of consuming s bytes of memory with multi-squaring tables, which specifies a trade-off between computation time and memory use. Note that a hard memory constraint can be obtained by defining $c_{\text{mem}}(s)$ to be a step function to infinity.

6.1 Search formulation

To compute an optimal addition chain for $m - 1$, we use as a search space the set S of valid addition chains $A = \langle a_i, b_i, o_i\rangle_{i\in[1,k]}$ summing to values $a_k \leq m - 1$, where the extra variable $o_i \in \{\delta, \tau\}$ specifies whether direct (δ) or table-based (τ) multi-squaring is used. The successors

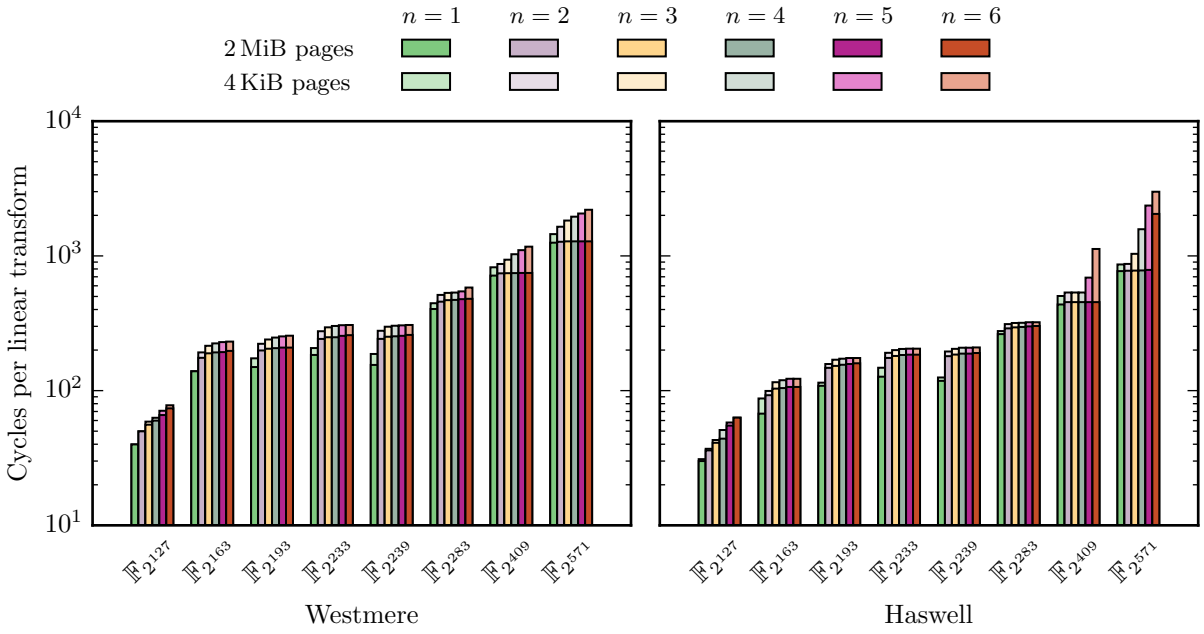

Figure 1: Effect of virtual memory page size on table-based linear field operation (i.e. multi-square) performance, for varying values of n, the total number of lookup tables used concurrently. Results are shown for a fixed block size of $B = 8$ bits. The lighter-color bar extensions show the overhead introduced by the use of 4 KiB pages, as discussed in section 5.3.

of a state are the valid extensions of that addition chain. Without loss of generality, we consider only addition chains in which $a_1, \ldots, a_k$ are monotonically increasing.

Given an addition chain $A = \langle a_i, b_i, o_i \rangle_{i \in [1,k]}$, let $T_j(A) \coloneqq \{b_i \mid i \in [1,j], o_i = \tau\}$ be the set of distinct multi-squaring tables used by the first j steps of the chain, and let $U_j^o(A) \coloneqq \{i \mid i \in [1,j], o_i = o\}$ for $o = \tau$ and $o = \delta$ be the subsets of the first j steps for which table-based and direct multi-squaring are used, respectively. Restating our cost model formally, the cost of the j-step prefix $A_{1\ldots j}$ of the chain A, depending on $n = |T_j(A)|$, is given by

$$\begin{aligned}\mathrm{cost}(A_{1\ldots j}) &\coloneqq \mathrm{cost}_n(A_{1\ldots j}) \\ &\coloneqq j \cdot c_{\mathrm{mul}} + c_{\mathrm{mem}}(n \cdot s_{m,B}) + |U_j^\tau(A)| \cdot \tau_{B,n} + \sum_{i \in U_j^\delta(A)} b_i \cdot c_{\mathrm{sq}}.\end{aligned}$$

Recall that $s_{m,B}$ is the size in bytes of a single multi-squaring table with block size B, defined in section 4. Note that the effective *marginal cost* c_k of the last action $\langle a_k, b_k, o_k \rangle$ is given by

$$\begin{aligned}c_k &\coloneqq \mathrm{cost}(A_{1\ldots k}) - \mathrm{cost}(A_{1\ldots k-1}) \\ &= c_{\mathrm{mul}} + \begin{cases} b_k \cdot c_{\mathrm{sq}} & \text{if } o_k = \delta; \\ \tau_{B,|T_k|} & \text{if } o_k = \tau \text{ and } T_k = T_{k-1}; \\ \tau_{B,|T_k|} + c_{\mathrm{mem}}(|T_k| \cdot s_{m,B}) - c_{\mathrm{mem}}(|T_{k-1}| \cdot s_{m,B}) & \\ \qquad + (\tau_{B,|T_k|} - \tau_{B,|T_{k-1}|}) \cdot |U_{k-1}^\tau| & \text{if } |T_k| = |T_{k-1}| + 1. \end{cases}\end{aligned}$$

Table-based multi-squaring to the power 2^{a_i} is never advantageous relative to direct multi-squaring if $b_i \cdot c_{\mathrm{sq}} \leq \tau_{B,|T|}$, and therefore we exclude such sub-optimal cases from the search

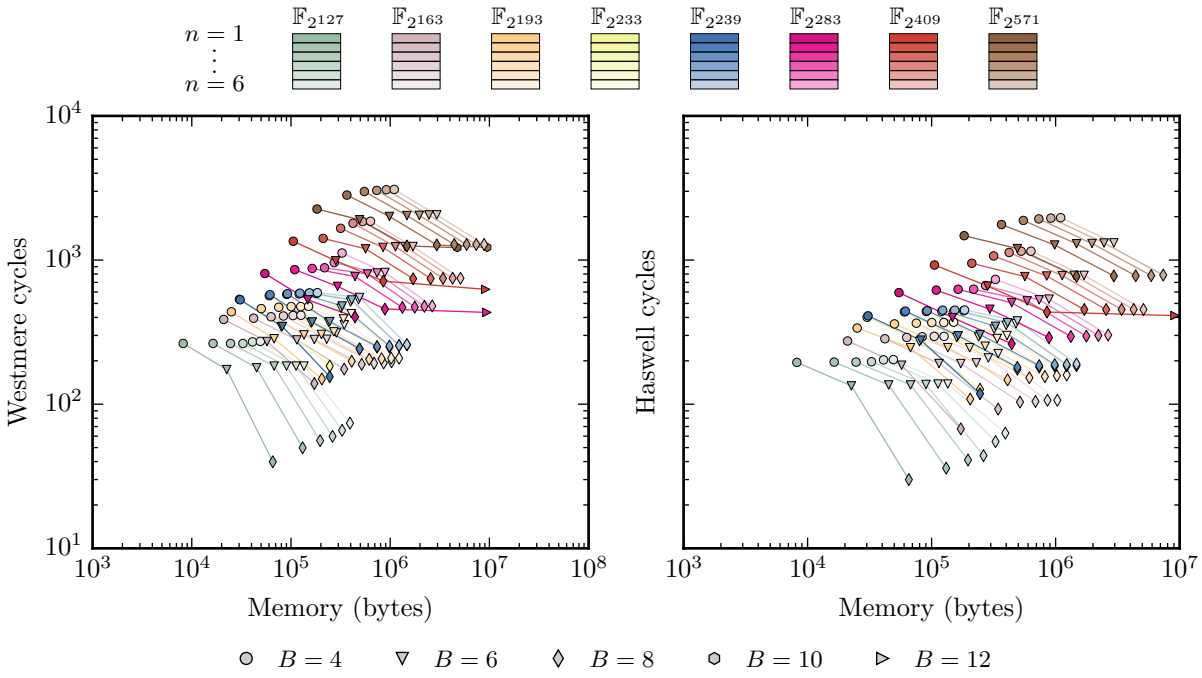

Figure 2: Table-based linear field operation (i.e. multi-square) time/memory Pareto frontiers for varying values of n, the total number of lookup tables used concurrently. Different choices of block size B lead to different trade-offs between CPU cycles per linear transform, shown on the vertical axis, and total memory required by n tables, shown on the horizontal axis. 2 MiB (huge) pages were used in all cases. No Pareto-optimal points with block size $B = 16$ were found.

space. Furthermore, in an optimal addition chain, if a multi-squaring table for the power 2^{b_i} is used at one step of the chain, i.e. $o_i = \tau$, then it must be the case that $c_{\text{sq}} \cdot b_i > \tau_{B,|T|}$, and thus there is no advantage in using direct multi-squaring for the same power at another step; therefore we also exclude this case from the search space.

6.2 Search algorithm

While the search space S is acyclic as defined, we can consider two equal-length addition chains $A = \langle a_i, b_i, o_i \rangle_{i \in [1,k]}$ and $A' = \langle a'_i, b'_i, o'_i \rangle_{i \in [1,k]}$ to be equivalent if the following conditions hold:

$$\begin{aligned} \{a_i \,|\, i \in [1,k]\} &= \{a'_i \,|\, i \in [1,k]\}, \\ \{(b_i, o_i) \,|\, i \in [1,k]\} &= \{(b'_i, o'_i) \,|\, i \in [1,k]\}, \\ |U_k^\tau(A)| &= |U_k^\tau(A')|. \end{aligned}$$

We use A* graph search over the quotient set[2] of equivalence classes of S (each represented by the triplet $\langle \{a_i\}, \{(b_i, o_i)\}, |U_k^\tau| \rangle$) to find an optimal path;[3] note that we still unambiguously specify *paths* through the quotient set as full addition chains $\langle a_1, b_1, o_1 \rangle, \ldots, \langle a_k, b_k, o_k \rangle$. The initial state $\langle \varnothing, \varnothing, 0 \rangle$ corresponds to an empty addition chain.

[2]It is straightforward to verify that this equivalence relation respects all search operations.

[3]Concretely, we do not expand a state during the search if we have already expanded an equivalent state.

Each search state is implicitly associated with a multi-squaring table block size B which is fixed after the initial state. The search priority queue is seeded with a separate initial state for each possible block size B.

For any given addition chain $A = \langle a_i, b_i, o_i \rangle_{i \in [1,k]} \in S$, A* search requires a heuristic lower bound $h(A)$ on the cost to reach the goal. We specify this based on a *relaxation* of the problem:

Definition 2. *A sequence $\langle a_i, b_i, o_i \rangle_{i \in [1,k]}$ is said to be a* relaxed addition chain for p *if*

1. $a_k \geq p$, *in place of the requirement that* $a_k = p$;
2. $\max\{b_i, a_i - b_i\} \leq \max(\{a_j \mid j < i\} \cup \{1\})$ *for all* $i \in [1, k]$, *in place of the requirement that* $\{b_i, a_i - b_i\} \subseteq \{a_j \mid j < i\} \cup \{1\}$.

Definition 3. *Let $\mathcal{R}_p(A)$ to be the set of* relaxed addition chains $\langle a_1, b_1, o_1 \rangle, \ldots, \langle a_k, b_k, o_k \rangle, \ldots, \langle a_{k'}, b_{k'}, o_{k'} \rangle$ *for p that extend $A = \langle a_i, b_i, o_i \rangle_{i \in [1,k]}$. We define the heuristic*

$$h(A) \coloneqq \min_{A' \in \mathcal{R}_{m-1}(A)} \text{cost}(A') - \text{cost}(A)$$

to be the minimum additional cost of a relaxed addition chain for $m - 1$ that extends A.

Since $h(A)$ is defined as the additional cost of the optimal solution to a relaxation of the problem, it is necessarily a consistent heuristic. This justifies our use of A* graph search rather than tree search.[17, p. 95]

6.3 Derivation of heuristic cost $h(A)$

The following lemma shows that we need only consider a restricted subset of $\mathcal{R}_{m-1}(A)$ to compute the heuristic cost $h(A)$.

Definition 4. *Define $\mathcal{R}'_p(A)$ to be the set of relaxed addition chains $A' = \langle a_i, b_i, o_i \rangle_{i \in [1,k']} \in \mathcal{R}_p(A)$ that satisfy the following conditions:*

$$\forall i \in [k+1, k'] : a_i = b_i + a_{i-1}; \tag{3}$$

$$\forall i \in [k+1, k'] : o_i = \tau \implies b_i = \begin{cases} \max T_{i-1} & \text{if } T_i = T_{i-1}, \\ a_{i-1} & \text{if } T_i \neq T_{i-1}; \end{cases} \tag{4}$$

$$\forall i \in [k+2, k'] : o_{i-1} = \tau \wedge T_{i-1} \neq T_k \implies o_i = \tau. \tag{5}$$

Lemma 1. $\min_{A' \in \mathcal{R}_p(A)} \text{cost}(A') = \min_{A' \in \mathcal{R}'_p(A)} \text{cost}(A')$.

Proof. Given any relaxed addition chain $A' \in \mathcal{R}_p(A)$, for each successive value of i for which either eq. (3) or (4) is not satisfied, we can simply set b_i and then a_i as required by the equations without changing the cost or invalidating the relaxed addition chain requirements. We can, therefore, uniquely represent any $A' \in \mathcal{R}_p(A)$ satisfying eqs. (3) and (4) by a sequence $c_{k+1}, \ldots, c_{k'}$ of values from the set $[1, m] \cup \{\tau, \tau'\}$ where

$$c_i = \begin{cases} b_i & \text{if } o_i = \delta; \\ \tau & \text{if } o_i = \tau \text{ and } b_i = \max T_{i-1}; \\ \tau' & \text{otherwise.} \end{cases}$$

Suppose that eqs. (3) and (4) hold for A' but eq. (5) is violated. Let i^* be the first value for which eq. (5) does not hold, i.e. $o_{i^*-1} = \tau$ and $T_{i^*-1} \neq T_k$ but $o_{i^*} = \delta$. Let $j = \max\{i \in [k+1, i^*-1] \mid T_k = T_i\}$. It follows from eq. (4) and from the assumption of eq. (5) holding for

$i \in [k+2, i^*-1]$ that $c_{j+1} = \tau'$ and $\{c_{j'} \mid j' \in [j+2, i^*-1]\} \subseteq \{\tau, \tau'\}$. Define the modified relaxed addition chain A'' by the sequence

$$c_{k+1}, \ldots, c_j, \min(b_{i^*}, a_j), c_{j+1}, \ldots, c_{i^*-1}, c_{i^*+1}, \ldots, c_{k'}.$$

Note that A'' is a valid relaxed addition chain that satisfies eqs. (3) and (4) and has one fewer violation of eq. (5) than A'. Since $b_{i^*} \leq a_{i^*-1}$, the modified chain A'' reaches a final value at least as high as A'. Furthermore, $\text{cost}(A'') \leq \mathit{cost}(A')$. We can therefore repeatedly apply this procedure in order to satisfy eq. (5). □

Given $A' = \langle a_i, b_i, o_i \rangle_{i \in [1,k']} \in \mathcal{R}'_{m-1}(A)$, let $w = \max\{i \in [k, k'] \mid T_k = T_{k'}\}$ be the maximum value reached in A' without the use of any additional multi-squaring tables. For fixed values w and $n = |T_{k'}(A')|$, lemma 1 implies we can compute the minimum cost of a relaxed addition chain extension as the sum of two independent terms:

1. $h^1_{w,n}(A)$, the minimum additional cost_n value of a relaxed addition chain extension $A'' \in \mathcal{R}'_w(A)$ for w that uses no additional multi-square tables;
2. $h^2_{w,n}(A)$, the cost of a minimum-length relaxed addition sequence that reaches $m-1$ starting from w using *only* additional multi-squaring tables. Note that since we have fixed n, the cost depends only on the length of the sequence.

To derive $h^1_{w,n}(A)$, note that to reach a value $\geq w$ with no additional multi-squaring tables, we can either use direct multi-squaring exclusively, with a cost of

$$g_\delta(w, a_k) \coloneqq \lceil \log_2 w/a_k \rceil \cdot c_{\text{mul}} + (w - a_k) \cdot c_{\text{sq}},$$

or for non-empty $T_k(A)$, repeatedly use the largest table $\max T_k(A)$ except for a possible last step of direct multi-squaring, with a cost of

$$g_\tau(w, n, a_k, t) \coloneqq u \cdot (\tau_{B,n} + c_{\text{mul}}) + 1_{r>0} \cdot (c_{\text{mul}} + \min(\tau_{B,n}, r \cdot c_{\text{sq}})),$$

where $u = \lfloor (w - a_k)/t \rfloor$, $r = w \bmod t$, and $t = \max T_k(A)$. Since we restrict the search space such that $t \cdot c_{\text{sq}} > \tau_{B,n}$ for all $t \in T_k$, the latter cost is necessarily lower whenever $T_k(A)$ is non-empty, and hence:

$$h^1_{w,n}(A) \coloneqq \text{cost}_n(A) - \text{cost}(A) + \begin{cases} g_\tau(w, n, a_k, \max T_k(A)) & \text{if } T_k \neq \varnothing; \\ g_\delta(w, a_k) & \text{otherwise.} \end{cases}$$

Note that $\text{cost}_n(A) - \text{cost}(A) = c_{\text{mem}}(n \cdot s_{m,B}) - c_{\text{mem}}(|T_k| \cdot s_{m,B}) + |U^\tau_k| \cdot (\tau_{B,n} - \tau_{B,|T_k|})$.

To derive $h^2_{w,n}(A)$, note that by lemma 1, we need only consider sequences of the form

$$\langle 2w, w, \tau \rangle, \ldots, \langle q_1 w, w, \tau \rangle, \langle 2q_1 w, q_1 w, \tau \rangle, \ldots, \langle q_1 q_2 w, q_1 w, \tau \rangle, \langle 2q_1 q_2 w, q_1 q_2 a_2, \tau \rangle, \ldots, \left\langle w \textstyle\prod_{i=1}^{\ell} q_i, w \prod_{i=1}^{\ell} q_i, \tau \right\rangle,$$

where $\ell = n - |T_k(A)|$, with $q_1 - 1$ steps using a table for w, $q_2 - 1$ steps using a table for $q_1 w$, up to $q_\ell - 1$ steps using a table for $w \prod_{i=1}^{\ell-1} q_i$. The final value reached is $w \prod_{i=1}^{\ell} q_i$. The total number of steps is $\sum_{i=1}^{\ell}(q_i - 1)$, with each step costing $c_{\text{mul}} + \tau_{B,n}$. For a given number of steps (and cost), the final value is maximized if $\min_{i=1}^{\ell} q_i \geq \max_{i=1}^{\ell} q_i - 1$. Hence,

$$h^2_{w,n}(A) = (c_{\text{mul}} + \tau_{B,n}) \cdot g_2(w, m-1, n - |T_k(A)|),$$

where $g_2(w, p, \ell) = \min\left\{ \sum_{i=1}^{\ell}(q_i - 1) \;\middle|\; q \in \mathbb{Z}^\ell_{\geq 2} \wedge w \prod_{i=1}^{\ell} q_i \geq p \right\}$.

The final heuristic value $h(A)$ is equal to $\min_{w,n} h^1_{w,n}(A) + h^2_{w,n}(A)$. Note that $h^1_{w,n}(A)$ is monotonically non-decreasing with w and n, and $h^2_{w,n}(A)$ is monotonically non-increasing with w. The heuristic cost $h(A)$ depends only a_k, $|T_k|$, $|U^\tau_k|$, and $\max T_k$, which respects the equivalence relation on addition chains that we defined.

6.4 Efficient computation of $h(A)$

We can compute $h(A)$ more efficiently using some precomputation. For the fixed value $p = m-1$ and each value of $\ell \in [0, \lceil \log_2 p \rceil]$, we precompute a minimal set of pairs

$$G^p_0 \coloneqq \{\langle p, 0\rangle\};$$
$$G^p_\ell \coloneqq \left\{ \langle w, g_2(w,p,\ell)\rangle \;\middle|\; w = \operatorname*{arg\,min}_{w' \in [1,p]} g_2(w',p,\ell) \right\}.$$

Then to compute $h(A)$, for $n \in [|T_k|, |T_k| + \lceil \log_2 p/a_k \rceil]$, we compute

$$\min_{\langle w,x\rangle \in G^p_{n-|T_k|}} (\tau_{B,n} + c_{\mathrm{mul}}) \cdot x + h^1_{w,n}(A).$$

We maintain the lowest total cost α found for the current and previous values of n. We iterate over pairs $\langle w, x\rangle$ in order of increasing w and $h^1_{w,n}(A)$ (and decreasing x), pruning the initial sequence of pairs containing all but the highest value of $w \le a_k$, and all values $x \ge \alpha - \mathrm{cost}_n(A) + \mathrm{cost}(A)$. We stop iterating over $\langle w, x\rangle$ pairs once $h^1_{w,n}(A) \ge \alpha$.

Because $h(A)$ depends on only part of the state information, namely a_k, $|T_k|$, $|U^\tau_k|$, and $\max T_k$, the computational cost of computing the heuristic can be reduced significantly through memoization.

7 Inversion performance evaluation

We implemented the A* search procedure in Python and computed optimal Itoh–Tsujii inversion addition chains (for each binary field) for the Intel Westmere and Intel Haswell test platforms, estimating c_{sq}, c_{mul}, and $\tau_{B,n}$ separately for each platform as described in section 5. We varied c_{mem} in order to obtain all Pareto-optimal solutions (with respect to memory use and the model-predicted computational cost). We used a simple program to generate the C++ inversion code and appropriate multi-squaring tables for each addition chain computed by the search procedure, in order to evaluate the performance. The results are shown in fig. 3.

In addition to using the full cost model (referred to as the *variable table cost* (VT) model in the result figures and tables), we also computed optimal addition chains (using the same search algorithm) under several restricted models in order to evaluate the utility of each component of our model:

1. A *table-free* (TF) model in which $\tau_{B,n} = \infty$, such that only direct multi-squaring is used, and the search is reduced to merely finding a minimum-length addition chain. A table-free implementation, as used by Bluhm and Gueron [4], has the advantage of being constant time with respect to the input, which is necessary for the security of some applications though is often unnecessary in the hashing setting.
2. A *minimum-length* (ML) model in which table-based multi-squaring is used but not accounted for by the addition chain search. As in the table-free case, the search is reduced to merely finding a minimum-length addition chain, but multi-squaring tables are used for any value b_i in the chain for which $c_{\mathrm{sq}} \cdot b_i > \tau$, for some threshold τ. Choosing $\tau = \tau_{B,1}$ roughly corresponds to the approach taken by prior work [14, 19] using multi-squaring tables.

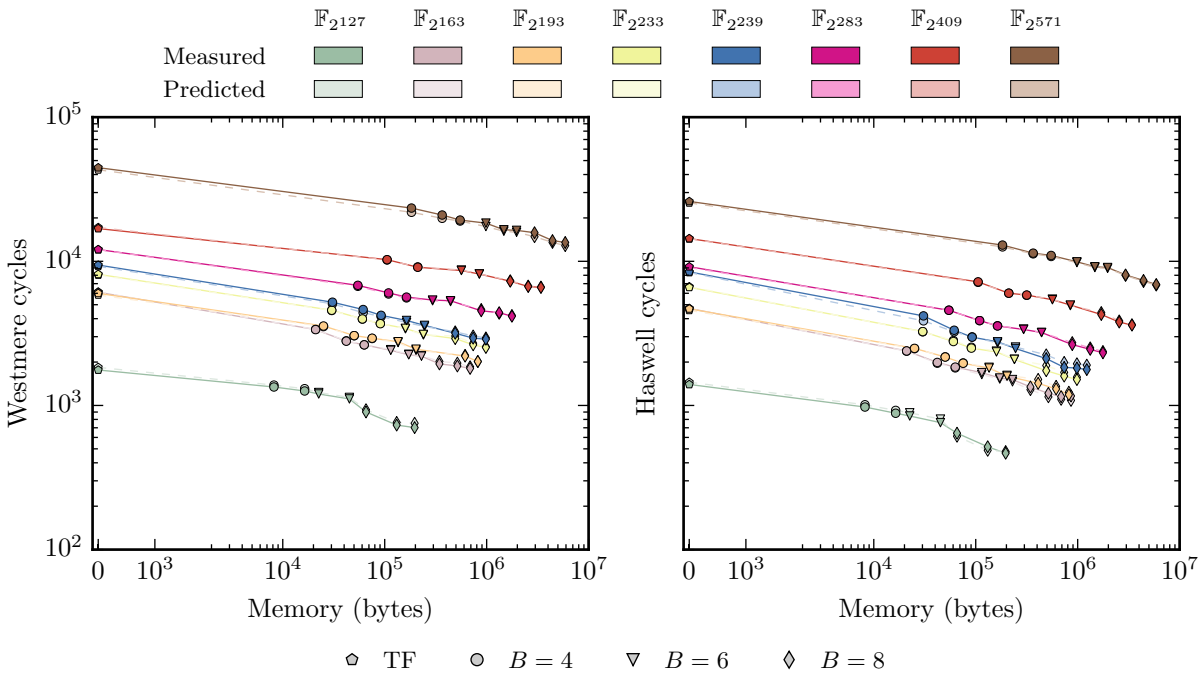

Figure 3: Inversion time/memory Pareto frontiers, showing the trade-off between memory use and CPU cycles per field inversion. All Pareto-optimal points obtained by varying c_{mem} and jointly optimizing over both the addition chain and the multi-square table block size B are shown; the corresponding block size for each point is indicated by the marker. The execution times predicted by the full variable table cost model are shown with dashed lines in lighter shades. The points on the left with zero memory use correspond to table-free (TF) addition chains. No Pareto-optimal points using a block size $B \in \{10, 12, 16\}$ were found.

3. A *single table cost* (ST) model in which we fix $\tau_{B,n} = \tau_{B,1}$, the cost of table-based multi-squaring estimated from a single-table benchmark.

Detailed results are given in appendix A.

8 Discussion

Reliable fine-grained performance measurement can be challenging due to compiler instruction reordering and optimization, processor branch prediction and out-of-order execution, and operating system jitter, among other factors, and these effects are magnified in our estimation of the cost of field inversion implementations as the sum of the costs of many individual operations. By carefully controlling for other sources of variance, including virtual page table size, the number of concurrent tables, and operation-specific benchmarking overhead, our cost model achieved a very low median error rate of 1.50% (fig. 4) in spite of these challenges.

Our A* search procedure proved to be highly effective in computing optimal addition chains for any desired performance/memory tradeoff. Even with an unoptimized Python search implementation, our relaxation-based heuristic made the computation of complete Pareto frontiers (requiring many independent searches) take only a few CPU minutes (appendix B), a negli-

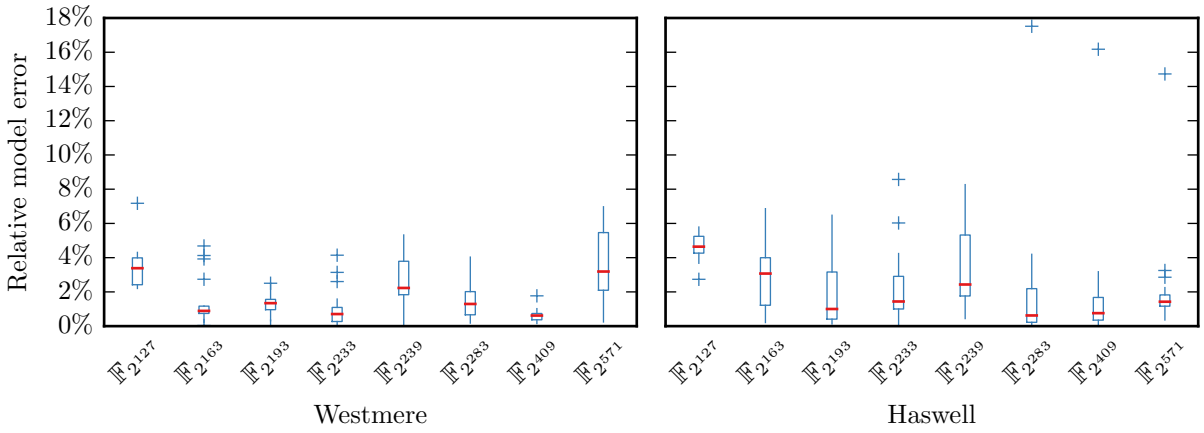

Figure 4: Field inversion cost model relative errors (box plot). The error was calculated as the absolute value of the difference of the predicted cost and the measured cost, divided by the measured cost. The results were aggregated over all distinct optimal addition chains for all multi-square block sizes B and all values of c_{mem}. The overall median relative error was 1.50%.

gible amount of precomputation time for most elliptic curve cryptographic applications. In comparison, without a heuristic the search is completely intractable even for small field degrees.

The accuracy of our cost model allowed our search procedure to find addition chains that significantly reduced computational and memory costs compared to addition chains found by the ad-hoc criteria used by prior work (appendix A). Comparisons to simplified models confirm the utility of our full model: performance is improved by accounting for the cost of table-based multi-squaring in the addition chain optimization, and the performance is further improved by modeling the dependence of this cost on the number of concurrent multi-squaring tables in use. Compared to choosing the *best possible* set of multi-square tables for a fixed minimum-length addition chain, the joint model-based optimization over both addition chain and multi-square tables performed by our search procedure results in significantly better performance.

We imposed several limitations on our model, in the interest of keeping the complexity of the cost model and search procedure manageable: *(a)* inversion implementations are restricted to using a single block size B for all multi-squaring tables; and *(b)* other methods of multi-squaring are not considered, such as using a combination of one or more lookup tables and direct squaring. We expect, however, that any improvement in performance or memory use from lifting these restrictions would be small.

The use of multi-squaring tables introduces susceptibility to side channel attacks. However, blinding can be used to effectively prevent any information leak from memory access patterns, at the cost of just two additional field multiplications (and m random bits).

In the case that multiple independent field inversions must be computed, a speedup orthogonal to our investigation of optimal addition chains is also possible. Using Montgomery's trick, n independent elements can be inverted simultaneously at the cost of just 1 field inversion and $3(n-1)$ field multiplications. [18] Unless the batch size n is very large, however, it is still advantageous to reduce the cost of the single inversion required.

Alternative inversion methods exist, but are less efficient for software implementations on modern CPUs. Itoh–Tsujii inversion was originally proposed for use with a normal basis rather than polynomial basis representation of field elements. Under a normal basis $\{\beta^{2^i} \mid i \in [0, m-1]\}$, multi-squaring is just a cyclic shift, with negligible computational cost. However, software-

implemented multiplication is typically much slower with a normal basis than with a polynomial basis,[10, p. 72] and in practice this outweighs the speedup of multi-squaring. Inversion can also be performed using the Extended Euclidean Algorithm, but due to the large number of branching operations and shifts by arbitrary numbers of bits required, the performance is not competitive with polynomial basis Itoh–Tsujii inversion on modern CPU architectures. [19]

References

[1] Omran Ahmadi, Darrel Hankerson, and Francisco Rodrıguez-Henrıquez. Parallel formulations of scalar multiplication on Koblitz curves. *Journal of Universal Computer Science*, 14(3):481–504, 2008.

[2] Diego F. Aranha, Pierre-Alain Fouque, Chen Qian, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. Binary elligator squared. Cryptology ePrint Archive, Report 2014/486, 2014. `http://eprint.iacr.org/`.

[3] Hatem M. Bahig. Improved generation of minimal addition chains. *Computing*, 78(2):161–172, 2006.

[4] Manuel Bluhm and Shay Gueron. Fast software implementation of binary elliptic curve cryptography. *IACR Cryptology ePrint Archive*, 2013:741, 2013.

[5] Joppe W. Bos, Thorsten Kleinjung, Ruben Niederhagen, and Peter Schwabe. Ecc2k-130 on cell cpus. In *Progress in Cryptology–AFRICACRYPT 2010*, pages 225–242. Springer, 2010.

[6] Daniel R. L. Brown. The encrypted elliptic curve hash. *IACR Cryptology ePrint Archive*, 2008:12, 2008.

[7] Daniel R. L. Brown, Adrian Antipa, Matt Campagna, and Rene Struik. ECOH: the Elliptic Curve Only Hash. *Submission to NIST*, 2008.

[8] Kenny Fong, Darrel Hankerson, Julio López, and Alfred Menezes. Field inversion and point halving revisited. *Computers, IEEE Transactions on*, 53(8):1047–1059, 2004.

[9] Jorge Guajardo and Christof Paar. Itoh-Tsujii inversion in standard basis and its application in cryptography and codes. *Designs, Codes and Cryptography*, 25(2):207–216, 2002.

[10] Darrel Hankerson, Scott Vanstone, and Alfred J Menezes. *Guide to elliptic curve cryptography*. Springer, 2004.

[11] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Information and computation*, 78(3):171–177, 1988.

[12] Jeremy Maitin-Shepard. C++ Elliptic Curve Multiset Hash library. `http://jeremyms.com/ecmh`.

[13] National Institute of Standards and Technology. FIPS 186-4: Digital Signature Standard (DSS), Federal Information Processing Standard (FIPS), publication 186-4. Technical report, Department of Commerce, Gaithersburg, MD, USA, July 2013. URL `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`.

[14] Thomaz Oliveira, Julio López, Diego F Aranha, and Francisco Rodríguez-Henríquez. Two is the fastest prime: lambda coordinates for binary elliptic curves. *Journal of Cryptographic Engineering*, 4(1):3–17, 2014.

[15] Gabriele Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. September 2010.

[16] Certicom Research. *SEC 2: Recommended Elliptic Curve Domain Parameters*. Standards for Efficient Cryptography, September 2000. URL `http://www.secg.org/download/aid-386/sec2_final.pdf`. Version 1.0.

[17] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 2010. ISBN 9780136042594.

[18] Hovav Shacham and Dan Boneh. Improving ssl handshake performance via batching. In *Topics in Cryptology—CT-RSA 2001*, pages 28–43. Springer, 2001.

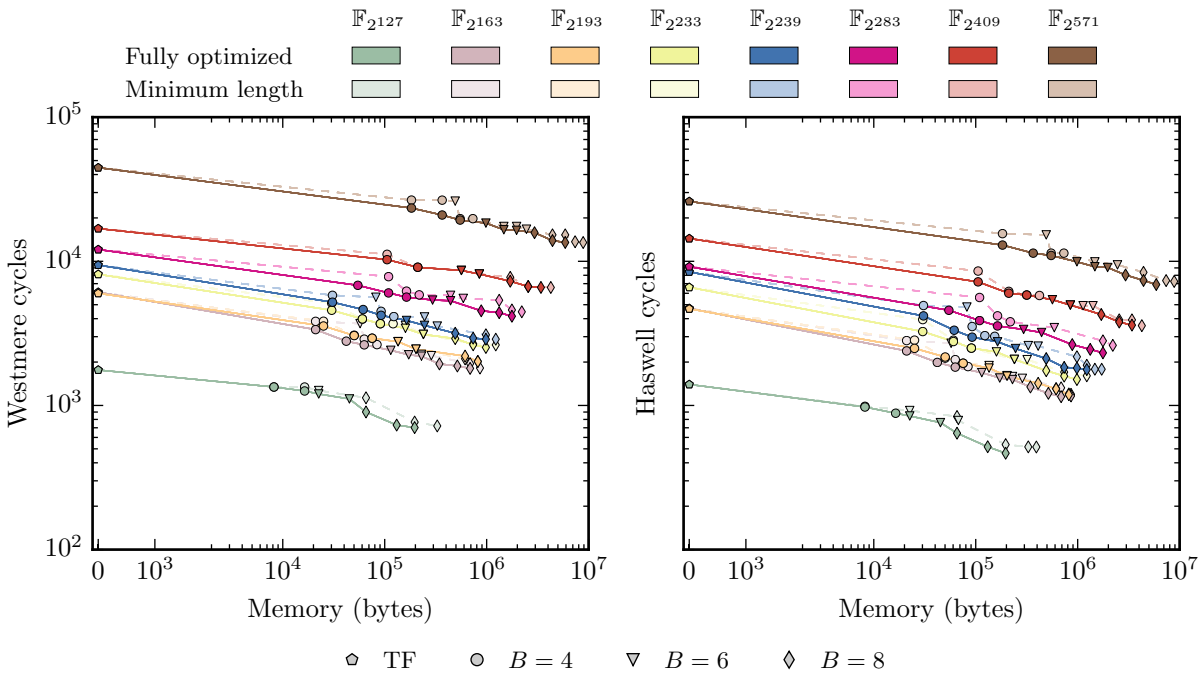

Figure 5: Effect of cost model on inversion time/memory Pareto frontiers. For each field, a minimum-length addition chain was computed, and all inversion implementations obtained by varying the number of multi-square tables and the block size B were evaluated. The Pareto-optimal points (according to the actual timing results) for these minimum-length addition chain implementations are shown with dashed lines and in lighter colors, overlaid by the Pareto-optimal points obtained from our complete cost model by varying c_{mem}.

[19] Jonathan Taverne, Armando Faz-Hernández, Diego F Aranha, Francisco Rodríguez-Henríquez, Darrel Hankerson, and Julio López. Software implementation of binary elliptic curves: impact of the carry-less multiplier on scalar multiplication. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 108–123. Springer, 2011.

[20] Edward G Thurber. Efficient generation of minimal length addition chains. *SIAM Journal on Computing*, 28(4):1247–1263, 1999.

A Cost model comparison

For a fixed block size of $B = 8$ and a fixed cost $c_{\text{mem}} = 0$, the addition chains that resulted from each of the four cost model variants are compared in table 2. The optimal addition chains and multi-square table sets under each cost model for each CPU architecture are shown, along with the measured execution time, and the execution time predicted by our complete cost model.[4]

Relative to the table-free implementation, the use of multi-square tables with the same addition chain (the minimum-length model) provides a large improvement, reducing the computational cost by a factor of 2–4. Optimizing the addition chain as well, using the single table

[4]The predicted execution times shown are based on the complete cost model in all cases, even for entries corresponding to addition chosen computed based on one of the restricted models.

cost model, provided on average an additional moderate reduction in cost. The full variable table cost model further reduced the computational cost for some of the fields. While in some cases, such as for $\mathbb{F}_{2^{409}}$, the same addition chain was optimal under multiple cost models, the full model never resulted in worse performance.

To compare the cost models under broader conditions, we generated inversion implementations using the same minimum length (ML) addition chains shown in table 2 for *each possible* threshold τ and block size B. We measured the performance of each implementation, and computed an *empirical* time/memory Pareto frontier. We compared this to the actual performance along the Pareto frontier obtained based on the *estimated* costs of optimal points under our complete model. The results are shown in fig. 5. Averaging over all Pareto-optimal minimum length implementations (and both the Westmere and Haswell CPU architectures), the optimal addition chains under our cost model required 15.3% less memory at the same performance level, or 6.7% less computation time at the same level of memory use. Using the minimum length criteria alone to select the addition chain essentially leaves the performance up to chance: different minimum-length addition chains, when used with multi-squaring tables, can have very different computational cost.

B Search efficiency

Run times and other statistics for the search procedure itself, summed over all of the independent searches required to compute the time/memory Pareto frontiers for each field degree, are given in table 3. The run times measure the single-core CPU time of our Python implementation run on the Westmere test machine, which was used for all searches. As our focus was on algorithmic efficiency rather than low-level optimization of the search procedure, the reported run times should be seen only as a general guide. Even for a fixed field degree, the number of search states expanded, and therefore the run time, is highly dependent on the particular ratios of c_{sq}, $\tau_{B,n}$, and c_{mem} to c_{mul}.

Given that a shortest path search over addition chains without the use of a heuristic would be completely intractable, it is clear from the run times and numbers of states expanded that our relaxed addition chain heuristic is highly effective. The large ratio of total heuristic calls to unique tuples $\langle a_k, |T_k|, |U_k^\tau|, \max T_k \rangle$ demonstrates the utility of memoizing heuristic values.

Table 2: $\mathbb{F}_{2^m}$ inversion addition chain cost model comparison. The optimal addition chains computed for the Intel Westmere (W) and Intel Haswell (H) machines under the *table-free* (TF), *minimum length* (ML), *single table cost* (ST), and the full *variable table cost* (VT) models are shown, along with the associated actual execution time in CPU cycles and predicted execution time (predict.) under the full variable table cost model. A fixed block size of $B = 8$ and $c_{\text{mem}} = 0$ were used. The addition chains were optimized independently for each test machine and cost model, but in some cases the same optimal chain was found for multiple machines/cost models.

	Model	Addition chain (b_i)	Tables	Westmere Actual/Predict.	Haswell Actual/Predict.
$\mathbb{F}_{2^{127}}$	TF	1 1 1 3 7 7 21 21 63	–	1761 / 1848	1397 / 1460
	ML	1 1 1 3 7 7 21 21 63	7 21 63	721 / 774	515 / 490
	ST	1 1 2 5 1 11 21 21 63	5 11 21 63	726 / 783	485 / 495
	VT	1 2 2 4 1 11 21 21 63	11 21 63	**701** / 764	**465** / 489
$\mathbb{F}_{2^{163}}$	TF	1 1 2 5 10 20 1 40 81	–	6102 / 6160	4742 / 4732
	ML/ST/VT(H)	1 1 2 5 10 20 1 40 81	5 10 20 40 81	1853 / 1920	**1206** / 1111
	VT(W)	1 2 2 4 10 20 1 41 81	10 20 41 81	**1808** / 1890	
$\mathbb{F}_{2^{193}}$	TF	1 1 3 6 12 24 48 96	–	5996 / 5875	4673 / 4595
	ML/ST	1 1 3 6 12 24 48 96	6 12 24 48 96	2091 / 2105	1231 / 1291
	VT	1 2 4 4 12 24 48 96	12 24 48 96	**2018** / 2044	**1190** / 1258
$\mathbb{F}_{2^{233}}$	TF	1 1 1 3 4 7 11 29 58 116	–	8127 / 8071	6598 / 6696
	ML	1 1 1 3 4 7 11 29 58 116	7 11 29 58 116	2682 / 2744	1611 / 1681
	ST	1 1 1 3 7 1 15 29 58 116	7 15 29 58 116	2578 / 2654	1515 / 1601
	VT	1 1 3 2 6 1 15 29 58 116	15 29 58 116	**2524** / 2584	**1506** / 1600
$\mathbb{F}_{2^{239}}$	TF	1 1 1 3 7 14 28 7 56 119	–	9382 / 9271	8741 / 8402
	ML/ST/VT(H)	1 1 1 3 7 14 28 7 56 119	7 14 28 56 119	2912 / 3006	**1783** / 1937
	VT(W)	1 1 3 2 6 14 28 56 14 126	14 28 56 126	**2885** / 2977	
$\mathbb{F}_{2^{283}}$	TF	1 1 1 3 3 7 17 34 68 10 136	–	12071 / 12090	9174 / 9101
	ML(W)	1 1 1 3 3 7 17 34 68 10 136	17 34 68 136	4452 / 4540	
	ML(H)	1 1 1 3 3 7 17 34 68 10 136	10 17 34 68 136		2648 / 2650
	ST/VT	1 2 4 1 8 1 18 35 1 71 141	18 35 71 141	**4145** / 4244	**2317** / 2398
$\mathbb{F}_{2^{409}}$	TF	1 1 3 6 12 3 24 51 102 204	–	16993 / 17210	14375 / 14373
	ML/ST/VT	1 1 3 6 12 3 24 51 102 204	24 51 102 204	**6676** / 6666	**3631** / 3664
$\mathbb{F}_{2^{571}}$	TF	1 1 1 3 4 11 22 7 44 95 95 285	–	44828 / 43235	26063 / 25492
	ML	1 1 1 3 4 11 22 7 44 95 95 285	22 44 95 285	14568 / 13469	7301 / 7212
	ST/VT	1 1 2 2 7 7 14 35 70 5 145 285	35 70 145 285	**13646** / 12922	**6962** / 6883

Table 3: A* search performance for Pareto frontier computation. For each field and block size B, a set of Pareto-optimal addition chains were computed using several independent searches with successively lower memory bounds. An optimal table-free addition chain was also computed for each field. The total number (#) of searches performed, the total time required by all searches, the number of search states expanded (successors computed), and the number of heuristic calls are shown. Due to the use of memoization, the heuristic was only computed once (per search) for each unique tuple $\langle a_k, |T_k|, |U_k^\tau|, \max T_k \rangle$, as described in section 6.4.

	Westmere model				Haswell model			
Field	#	Time (s)	States expanded	Heuristic calls Total / Uniq.	#	Time (s)	States expanded	Heuristic calls Total / Uniq.
$\mathbb{F}_{2^{127}}$	11	1.04	4 881	48 144/ 3 518	15	1.21	5 805	56 196/ 4 200
$\mathbb{F}_{2^{163}}$	20	0.56	2 640	25 285/ 4 872	21	0.51	2 317	22 456/ 5 015
$\mathbb{F}_{2^{193}}$	19	0.10	598	3 803/ 1 311	19	0.12	691	4 774/ 1 520
$\mathbb{F}_{2^{233}}$	19	2.09	8 877	98 210/ 9 067	18	8.48	25 895	408 592/10 143
$\mathbb{F}_{2^{239}}$	19	51.35	161 499	2 401 801/55 121	19	83.85	227 070	3 871 342/48 086
$\mathbb{F}_{2^{283}}$	18	13.10	51 094	588 551/24 589	17	9.39	36 464	425 197/13 807
$\mathbb{F}_{2^{409}}$	15	0.58	2 883	24 642/ 2 932	15	1.24	5 301	55 089/ 5 204
$\mathbb{F}_{2^{571}}$	16	115.97	352 675	4 861 047/57 307	16	157.53	471 921	6 478 492/51 543