Compiler Assisted Elliptic Curve Cryptography

M. Barbosa[1], A. Moss[2] and D. Page[2]

[1] Departamento de Informática, Universidade do Minho,
Campus de Gualtar, 4710-057 Braga, Portugal.
`mbb@di.uminho.pt`

[2] Department of Computer Science, University of Bristol,
Merchant Venturers Building, Woodland Road,
Bristol, BS8 1UB, United Kingdom.
`{moss,page}@cs.bris.ac.uk`

Abstract. Although cryptographic implementation tasks are often undertaken by expert programmers, a plethora of performance and security driven options, as well as more mundane software engineering issues, still make this a challenge. In an attempt to transfer expert knowledge into automated tools, we investigate the use of domain specific language and compilation techniques for cryptographic software, focusing on ECC in particular. Specifically, we describe experiments for specialisation of finite field arithmetic from general purpose code, and the description and optimisation of ECC point arithmetic using a cryptography-aware language and compiler. Our main results show that it is possible to allow description of ECC based software in a manner close to the original mathematics, while allowing the automatic production of an executable whose performance is close to that of a hand-optimised implementation.

1 Introduction

The increasing ubiquity of mobile computing devices has presented programmers with a problem. On one hand, such devices are required to be as compact and low-power as possible; on the other hand they are increasingly required to perform significant computational tasks. This dichotomy is further complicated by the issue of security which represents a restrictive overhead within many applications. Not only must the device execute algorithms that satisfy the application context, for example the use of digital signatures on smart-cards, but increasingly they must implement countermeasures against physical attack. An example is the concept of side-channel attack. By targeting the algorithm implementation rather than the mathematical underpinnings, such attacks are often able to recover secret information from a device simply by passive monitoring of features such as timing variation [15], power consumption [16] or electromagnetic emission [1].

Elliptic Curve Cryptography (ECC) offers a popular solution to the problem of implementing security using public key cryptography in constrained environments. The security of RSA, the most popular algorithm in other domains such as e-commerce, is based on the hardness of integer factorisation; ECC is based on the the Elliptic Curve Discrete Logarithm Problem (ECDLP). Since there is no known sub-exponential time algorithm to solve the ECDLP, ECC keys can be shorter than their RSA analogues while achieving the same security level: a 160-bit ECC key is roughly equivalent to a 1024-bit RSA key. This means an ECC based system is typically more efficient and utilises less resources than one based on RSA. Furthermore, flexibility in the mathematics that underpins ECC means that countermeasures against side-channel attack are both well studied and readily available; see for example [7][Chapters 4 and 5].

At face value, ECC based cryptographic schemes seem an ideal partner for mobile computing. However, the programmer is still faced with the problem of actually implementing said schemes. This presents two further hurdles. Firstly, the programmer is expected to be expert in an an extremely broad and fast moving field. The assumption that such a rich body of research can be absorbed and applied without error is tenuous for even the most expert programmer. Secondly, the programming tools presented to the developer to assist the construction of software within this specific context are relatively rudimentary. In particular, conventional programming languages and compilers are less than ideal: they do not support the types and operations required and thus cannot perform the optimisation and analysis typically offered to the programmer. Often, cryptographic software is described in a pseudo-high-level language: there are structured control flow statements but operations are otherwise at the level one would expect in a low-level language. The compiler cannot apply even basic optimisations such as register allocation when non-native types are used; it certainly cannot detect or resolve security related errors as it might do with errors relating to functional correctness. Standard software engineering issues such as maintainability add even further to this increasingly difficult task.

The natural solution is to investigate the use of domain specific languages and compilation techniques for cryptography. We attack the problem simultaneously in two directions. Firstly, we allow cryptographic software to be written in a domain specific language. The central principle is that concepts of cryptographic interest, particularly those relating to implementation, should be conveyed to the cryptography-aware compiler by the programmer via first class language features. The hope is that programmers will derive similar benefits to those experienced by switching from low-level assembly languages to higher-level languages. That is, by expressing their programs in a more natural manner and offering automated analysis, optimisation and transformation, a programmer will improve their productivity, reduce their rate of error and generally produce software of a higher quality. Secondly, we use specialisation techniques to automatically construct efficient run-time support systems from generic library code. By having the high-level program communicate system parameters to the compiler, it can generate a run-time that specifically matches the needs of that program and thus can be more efficient than the general library code.

The paper is organised as follows. We use Section 2 to present background material including brief overview of the fundamentals behind ECC and a description of our experimental platform. In Section 3 we present an implementation of curve arithmetic that utilises domain specific programming language and compilation techniques. Methods for optimising this implementation are then demonstrated in Section 4: we focus on automatic specialisation of field arithmetic in Section 4.1, placement of modular reduction operations in Section 4.2, and cache conscious ordering of field operations in Section 4.3. Finally we present some conclusions and areas for further work in Section 5.

2 Background

An Introduction to ECC Elliptic Curve Cryptography (ECC) was invented during the mid 1980s in independent work by Miller [18] and Koblitz [13], then generalised to include Hyperelliptic Curve Cryptography (HECC) by Koblitz [14] in 1989. We concentrate here only on ECC, for further reading on all issues covered in this basic introduction, see Menezes et al. [11], Blake et. al [6, 7] or Cohen et al. [8].

An elliptic curve E over the finite field K is defined by the general Weierstrass equation

$$E(K) : Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$

for $a_i \in K$. The K-rational points on a curve E, i.e. those $(x, y) \in K^2$ which satisfy the curve equation, plus the point at infinite $\mathcal{O}$, form an additive group under a group law defined by the chord-tangent process. Using basic coordinate geometry and given two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, one constructs arithmetic to compute the point $P_3 = (x_3, y_3) = P_1 + P_2$ as follows:

$$\begin{aligned} x_3 &= \lambda^2 + a_1\lambda - a_2 - x_1 - x_2 \\ y_3 &= (x_1 - x_3)\lambda - y_1 - a_1x_3 - a_3 \end{aligned}$$

where

$$\lambda = \begin{cases} \frac{3x_1^2+2a_2x_1+a_4-a1y1}{2y_1+a_1x_1+a_3} & if\ P_1 = P_2 \\ \\ \frac{y_1-y_2}{x_1-x_2} & if\ P_1 \neq P_2 \end{cases}$$

We term the case where $P_1 \neq P_2$ (resp. $P_1 = P_2$) point addition (resp. point doubling). Calculating the negation of a point, i.e. finding $-P_1$ given P_1, is computationally easy and so subtraction is usually performed using a negation following by an addition.

The point arithmetic described above includes an inversion in K, which is an expensive operation, to compute the value λ. To eliminate it, one can consider the use of projective coordinates and represent points on E using a triple $(x, y, z) \in K^3$ rather than simply $(x, y) \in K^2$. Of many systems, one of the most common is the use of Jacobian projective coordinates, a map between projective and affine spaces given by

$$(X, Y, Z) \mapsto (X/Z^2, Y/Z^3)$$

where the curve equation is now given by the homogenised Weierstrass equation

$$E : Y^2 + a_1XYZ + a_3YZ^3 = X^3 + a_2X^2Z^2 + a_4XZ^4 + a_6Z^6.$$

One can show that the resulting point arithmetic can be constructed without inversions in K. Furthermore, for specific K we simplify the general Weierstrass equation via a change of variables; the most common cases of $K = \mathbb{F}_p$, for some large prime $p > 3$, and $K = \mathbb{F}_{2^n}$, for some integer n, yield

$$E(\mathbb{F}_p) : Y^2 = X^3 + aXZ^4 + bZ^6$$

for $a, b \in \mathbb{F}_p$ and

$$E(\mathbb{F}_{2^n}) : Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6$$

for $a, b \in \mathbb{F}_{2^n}$. For $E(\mathbb{F}_p)$ it is common to fix $a = -3$ since this simplifies arithmetic on points.

With all of this in mind, the efficiency of ECC based schemes depends on three main features: the efficiency of arithmetic in K, the efficiency of arithmetic involving points on E and the efficiency of the exponentiation algorithm to compute point multiplications.

An Introduction to the Experimental Platform To provide a consistent experimental platform for the rest of the paper we selected a typical embedded processor solution from ARM. More specifically, we selected the ARM946E-S macro-cell [2] which incorporates a 32-bit ARM9 processor core. Although the core can be clocked much faster, we opted to use a modest 16 MHz. The processor core supports the ARM Thumb instruction set and a range of DSP extensions, although we do not use either. The macro-cell allows the processor core to be coupled internally to a configurable amount of Harvard style cache memory. For each of the data and instruction caches, we opted for the smallest 4-way set associative format with a 4-kB capacity arranged in 32-byte lines. Configured as such, the macro-cell is ideal for deployment in applications where high performance, low cost, small size and low power are key. ARM cites the embedded, media, communication and networking markets as targets; the macro-cell plays a central role in the Nintendo DS and Nokia N-Gage products. Development for, and simulation of, the ARM946E-S was performed using the ARM Developer Suite (ADS) 1.2. Although it is not perfectly accurate, we used the ARMulator instruction set simulator to obtain run-time statistics since this afforded a good compromise between ease and accuracy of experimentation.

3 Implementation of Curve Arithmetic

The basic purpose of a compiler for a high-level language is to translate a program into a lower-level (or executable) form. Essentially this mechanises the processes that an expert programmer might perform by hand and, as a result,

$$
\begin{aligned}
\lambda_1 &\leftarrow 3(x_1 - z_1^2)(x_1 + z_1^2) \\
z_3 &\leftarrow 2y_1z_1 \\
\lambda_2 &\leftarrow 4x_1y_1^2 \\
x_3 &\leftarrow \lambda_1^2 - 2\lambda_2 \\
\lambda_3 &\leftarrow 8y_1^4 \\
y_3 &\leftarrow \lambda_1(\lambda_2 - x_3) - \lambda_3
\end{aligned}
$$

```
dbl( x1 : gfp, y1 : gfp, z1 : gfp )
   : gfp, gfp, gfp
{
  l1 : gfp := 3 * ( x1 - z1**2 )
              * ( x1 + z1**2 );
  z3 : gfp := 2 * y1 * z1;
  l2 : gfp := 4 *x1 * y1**2;
  x3 : gfp := l1**2 - 2 * l2;
  l3 : gfp := 8 * y1**4;
  y3 : gfp := l1 * ( l2 - x3 ) - l3;

  return x3, y3, z3;
}
```

Fig. 1. Two descriptions of point doubling $P_3 = (x_3, y_3, z_3) = 2 \cdot P_1$ given $P_1 = (x_1, y_1, z_1)$ using Jacobian projective coordinates on $E(\mathbb{F}_p)$. The left-hand side is described in terms of the original formula from [6][Page 60], the right-hand side is the associated translation into CAO.

removes the associated tedium and error. Since the compiler is empowered with knowledge about the semantics of operations and types in the language, it can manipulate the program into a high-quality result while allowing the program specification to exist in a form which is natural to the programmer.

As such, an ideal route to implementation of ECC point arithmetic would be to simply write down formula, using a high-level programming language, as one finds them in a text book and then execute the compiled result. However, using a language which supports the types and operations required, such as Magma [9], seldom results in an efficient result. Conversely, using a language which supports efficient compilation, such as C, seldom results in easy translation since there is typically no native support for required types and operations.

As a means of allowing common compilation techniques to be applied to natural descriptions of ECC, we use the CAO language and associated compiler system [20]. Figure 1 demonstrates how one might translate text book formula for point doubling, using Jacobian projective coordinates on $E(\mathbb{F}_p)$, into a CAO function. Notice that the CAO function is able to naturally express the original formula since the language is equipped with a type system that includes $\mathbb{F}_p$. The CAO compiler is able to produce the NTL [22] based implementation detailed in Figure 2, which closely matches that one would construct by hand, using a range of standard optimisation techniques.

```
void dbl( ZZ_p& x3, ZZ_p& y3, ZZ_p& z3,
          ZZ_p& x1, ZZ_p& y1, ZZ_p& z1 )
{
  ZZ_p t0, t1, t2, t3, t4;

  sqr( t2, z1     ); sub( t1, x1, t2 ); add( t0, t1, t1 );
  add( t1, t0, t1 ); add( t0, x1, t2 ); mul( t4, t1, t0 );
  add( t0, x1, x1 ); add( t1, t0, t0 ); sqr( t0, y1     );
  mul( t3, t1, t0 ); sqr( t0, t0     ); add( t0, t0, t0 );
  add( t0, t0, t0 ); add( t2, t0, t0 ); add( t0, y1, y1 );
  mul( z3, t0, z1 ); sqr( t1, t4     ); add( t0, t3, t3 );
  sub( x3, t1, t0 ); sub( t0, t3, x3 ); mul( t0, t4, t0 );
  sub( y3, t0, t2 );
}
```

Fig. 2. The result of automatically compiling a CAO implementation of point doubling, shown in Figure 1, into an NTL based function (with slight hand modification used to improve readability).

4 Optimisation of Curve Arithmetic

4.1 Specialisation of Field Arithmetic

The description of ECC in Section 2 highlights the pivotal role of field arithmetic in overall performance. However, general purpose software libraries are often less than ideal in this context. Perhaps the most succinct written description of the problem is given by Avanzi [3] while discussing issues of performance in HECC. He states that general purpose software libraries:

> ... all introduce fixed overheads for every procedure call and loop, which are usually negligible for very large operands, but become the dominant part of the computations for small operands such as those occurring in curve cryptography.

In part, this is an obvious statement. Expert programmers routinely optimise and specialise their programs to avoid such overheads, potentially with some assistance from their compiler. This is especially true given that there are various ECC standards which specify a limited range of parameterisations. One can easily specialise for these particular cases. However, it is a vastly important statement from a software engineering perspective. Most programmers are not expert, especially in the context of cryptography where they may not even fully understand the underlying mathematics; they are bound by deadlines as well as performance targets; they might need to port their code to many different platforms and environments rather than for one-off use in a research paper.

To combat this problem, we investigated the feasibility of automatically generating special purpose field arithmetic code from a corpus of general purpose

	SECT163R1			SECT233R1			SECT283R1			SECT409R1			SECT571R1		
	Add	Sqr	Mul	Add	Sqr	Mul	Add	Sqr	Mul	Add	Sqr	Mul	Add	Sqr	Mul
A	5	77	577	7	68	937	7	115	961	10	102	1454	15	212	3545
B	1	27	386	1	33	743	2	42	756	4	48	1166	6	66	3017
C	1	27	373	1	30	681	2	36	719	4	44	1454	6	77	3077

	SECP192R1			SECP224R1			SECP256R1			SECP384R1			SECP521R1		
	Add	Sqr	Mul	Add	Sqr	Mul	Add	Sqr	Mul	Add	Sqr	Mul	Add	Sqr	Mul
A	9	178	188	10	234	248	11	283	301	13	571	618	30	1099	1179
B	7	58	60	7	76	80	9	147	159	9	221	261	20	384	747
C	7	58	60	7	76	80	9	147	159	9	221	261	20	384	747

Fig. 3. A comparison, with time in terms of microseconds, between three different implementations of arithmetic in $\mathbb{F}_{2^n}$ and $\mathbb{F}_p$ for standard values of n and p.

library code. The idea is that once the programmer has specified parameters in a high-level program, the compiler can specialise the generic library code into an efficient run-time and link it to the application. As such, Table 3 compares the performance of three different implementations of arithmetic in the field $\mathbb{F}_{2^n}$ and $\mathbb{F}_p$ for values of n and p that match those used in the SECG recommended domain parameters [21]. We focused on the core operations of field addition, squaring and multiplication. Although field inversion would also be required for a full ECC implementation, the use of projective coordinates means this operation is not significantly relevant to performance. The three implementations were constructed as follows:

Implementation A Entirely generic implementation in the sense that the same, correctly parameterised code would work for any n or p. For arithmetic in $\mathbb{F}_{2^n}$ the standard table based coefficient thinning method was used for squaring [11][Pages 52-53], multiplication was performed using the right-to-left comb method [11][Pages 48-51], the reduction used a generic word-wise approach [11][Pages 53-56]. For arithmetic in $\mathbb{F}_p$ standard integer addition [11][Page 20], multiplication [11][Page 31] and squaring [11][Page 35] were used; modular reduction was performed using the method of Barrett [11][Page 36].

Implementation B Same as Implementation A except that several core functions were specialised by hand to remove some of the obvious bottlenecks in performance. For example, in the arithmetic for $\mathbb{F}_{2^n}$ both the addition and vector shift functions were turned into macros with their inner loops fully unrolled so as to remove function call and loop overheads. The standard specialised reduction functions were implemented and utilised for each field [11][Pages 44-46 and 53-56].

Implementation C Has the same forms of specialisation as Implementation B but applied automatically using the Tempo 1.202 [10] specialisation system.

In order to specialise a given C function, the user specifies an execution context which details variables that will have static, constant values or dynamic, changing values. Tempo uses the value of static variables to perform aggressive transformations such as constant propagation, loop unrolling and dead code elimination; the end result is a function that is semantically the same as the original when the execution context is the same as that specified. The input to Tempo was taken directly from Implementation A with the only other inputs being constants relating to the field parameterisation that defined the execution context. The one caveat to this is the reduction function for arithmetic in $\mathbb{F}_p$ which was automatically generated using an external program that implemented the method of Solinas [23].

We used the ARM C compiler in all cases, with assembly language inserts to accelerate specific code segments and all compiler options tuned for speed. Comparison between Implementations A and B reveal what one would expect: the specialised version is quicker because the main overheads have been eliminated by hand. The more interesting result is that Implementation C which was generated automatically from Implementation A matches the performance of the hand specialised code in Implementation B: it actually often performs better, due to a more aggressive loop unrolling strategy than that undertaken by hand, until the point where it became too aggressive and misses in the instruction cache hampered the result. In hindsight it should not be surprising that Tempo was able to perform well with the given library code since the specialisation is mainly related to loop unrolling, constant propagation and some static control flow: essentially the specialisation requires no specific domain knowledge.

The key thing to note is that this positive result in terms of performance was achieved in a fraction of the time in terms of programming effort. With the caveat that any specialisation needs to be performed in context with the application using the field arithmetic in order to achieve good results, we can easily realise our goal of compiler assisted implementation by specialising a general purpose library into a special purpose run-time for said application. There is no need for the error prone and tedious specialisation by hand for each set of parameters; this acts as an aid to expressiveness in the library code, programmer productivity and portability of the entire system.

4.2 Lazy Reduction

Avanzi [3][Section 2.2] utilises what he terms lazy modular reduction techniques to improve the performance of his results. Lazy reduction removes specific modular reduction operations, combining them in others so that their cost is amortised. When working with the finite field $\mathbb{F}_p$ for example, this relaxes the constraint that intermediate results are strict members of $\mathbb{F}_p$ but improves performance by potentially eliminating computation.

An easy example of the potential for lazy reduction is presented by use of Barrett reduction [4] to implement arithmetic in $\mathbb{F}_p$. Working on a processor with word-size w one represents p using a vector of k base-b digits where $b = 2^w$.

Index	Operation	Reduction	Index	Operation	Reduction
0	$\lambda_1 \leftarrow z_1^2$	red_{mul}	11	$\lambda_{11} \leftarrow \lambda_{10} + \lambda_{10}$	red_{mul}
1	$\lambda_2 \leftarrow x_1 - \lambda_1$	red_{sub}	12	$\lambda_{12} \leftarrow \lambda_6^2$	red_{mul}
2	$\lambda_3 \leftarrow x_1 + \lambda_1$		13	$\lambda_{13} \leftarrow \lambda_{11} + \lambda_{11}$	red_{add}
3	$\lambda_4 \leftarrow \lambda_2 \cdot \lambda_3$	red_{mul}	14	$x_3 \leftarrow \lambda_{12} - \lambda_{13}$	red_{sub}
4	$\lambda_5 \leftarrow \lambda_4 + \lambda_4$		15	$\lambda_{14} \leftarrow \lambda_8^2$	
5	$\lambda_6 \leftarrow \lambda_5 + \lambda_4$		16	$\lambda_{15} \leftarrow \lambda_{14} + \lambda_{14}$	
6	$\lambda_7 \leftarrow y_1 \cdot z_1$		17	$\lambda_{16} \leftarrow \lambda_{15} + \lambda_{15}$	
7	$z_3 \leftarrow \lambda_7 + \lambda_7$	red_{mul}	18	$\lambda_{17} \leftarrow \lambda_{16} + \lambda_{16}$	red_{mul}
8	$\lambda_8 \leftarrow y_1^2$	red_{mul}	19	$\lambda_{18} \leftarrow \lambda_{11} - x_3$	red_{sub}
9	$\lambda_9 \leftarrow x_1 \cdot \lambda_8$		20	$\lambda_{19} \leftarrow \lambda_6 \cdot \lambda_{18}$	red_{mul}
10	$\lambda_{10} \leftarrow \lambda_9 + \lambda_9$		21	$y_3 \leftarrow \lambda_{19} - \lambda_{17}$	red_{sub}

Table 1. Sequence of operations with delayed reduction for point doubling $P_3 = (x_3, y_3, z_3) = 2 \cdot P_1$, given $P_1 = (x_1, y_1, z_1)$ (Jacobian projective coordinates on $E(\mathbb{F}_p)$).

Barrett presents a method for taking an integer $0 \leq x < b^{2k}$ and reducing it modulo p without the need for an expensive division operation. If p does not occupy a full k words, this leaves some unused storage. Consider for example the specification of the SECP521R1 curve [21] where $p = 2^{521} - 1$, a value that requires seventeen 32-bit words of storage but does not occupy 23 bits in the top word. One would normally input values to the reduction function in the range $[0..p^2)$, represented in $2k$ words, as the result of a multiplication. However, given this specific value of p the function can comfortably accept values in, for example, the range $[0..16p^2)$ due to the fact that $16p^2 < b^{2k}$. The key issue is that for this sort of suitable p, the cost of reduction with the relaxed input range is no more than with the strict range: this is ideal for combination with the idea of lazy reduction.

Montgomery representation [19] offers another efficient way to perform arithmetic in $\mathbb{F}_p$. To define the Montgomery representation of x, denoted x_M, one selects an $R = b^t > p$ for some integer t; the representation then specifies that $x_M \equiv xR \pmod{p}$. To compute the product of x_M and y_M held in Montgomery representation, one interleaves a standard integer multiplication with an efficient reduction technique tied to the choice of R. We term the conglomerate operation Montgomery multiplication and denote it by $z_M = x_M \star y_M$. Ordinarily, one has that $x_M, y_M, z_M \in [0 \ldots p)$ but it is possible to construct a redundant, or non-reduced Montgomery representation so that the input ranges are relaxed to $x_M, y_M \in [0 \ldots \epsilon p)$ for some suitable value of ϵ; roughly, this means selecting $R = b^t > \epsilon^2 p$.

For example, Walter [24] selects $\epsilon = 2$ in order to remove the need for the conditional, final subtraction in the implementation of $\star$. For suitable p and ϵ this again gives potential for combination with the idea of lazy reduction. However, there is one extra caveat in realising this combination. Consider the integer

Algorithm 1: An algorithm to automatically find lazy reduction points.

```
Input  : A straight-line function $F$, a bound on computation $I$ and
         initial weight $T_{init}$.
Output: A set of lazy reduction points $S$, or $\perp$ on failure.

$S \leftarrow \perp$
for $T = T_{init}$ downto $0$ do
    for $i = 0$ upto $I$ do
        Pick a set $R \subset F$ of reduction sites so as to satisfy:
         1. if $d$ defines symbol $r$, which is later input to an operation
            requiring a fully reduced operand, place a reduction after $d$.
         2. otherwise place reductions randomly so there are $T$ in total.

        Check that the ranges of symbols in $F$ satisfy:
         1. for each symbol $s$, the symbol is within the maximum range.
         2. for each definition $d$, the source operands are within the
            range specified by the operation.
         3. for each definition $d$, the target operands are within the
            range of some reduction operation.

        if $R$ passes all constraints then
            Evaluate $c = cost(R)$, the cost of placed reductions.
            if $S = \perp$ or $c < cost(S)$ then
                $S \leftarrow R$
return $S$
```

multiplication of two values held in Montgomery form $z = x_M \cdot y_M = xyR^2$, and a standard value held in Montgomery form $w_M = wR$. Unlike with the use of Barrett reduction, where values are simply integers and the reduction is simply accelerated, Montgomery form imposes a further constraint in that one cannot add together z and w_M or, more generally, unreduced and reduced representations.

Defining Reasonable Constraints Our task is to take a program F and automatically select a set $R \subset F$ of points after which reduction operations will be placed. We assume that F is straight-line and fairly short (which holds or can be made to hold for most ECC related functions); that arguments to F are fully reduced and that both return values and global variables need to be fully reduced at the end of the program.

Because of the large degree of freedom involved, we use a Monte Carlo approach to form a solution, guided by a number of constraints on features such as input and output ranges for given operations. For example, for a sequence of additions, subtractions and multiplications in $\mathbb{F}_p$ we might impose the following constraints:

1. The values of intermediate results cannot exceed the c_{max}.

2. We demand that
   - $x, y \in [0..c_{add})$, and $z \in [0..2 \cdot c_{add})$ for $z = x + y$ type operations.
   - $x, y \in [0..p)$, and $z \in (-p..p)$ for $z = x - y$ type operations.
   - $x, y \in [0..c_{mul})$, and $z \in [0..c_{mul}^2)$ for $z = x \cdot y$ type operations.
3. We distinguish three reduction operations
   - $y = red_{add}(x) = x \bmod p$ where $x \in [0..c_{red_{add}})$ and $y \in [0..p)$.
   - $y = red_{sub}(x) = x \bmod p$ where $x \in (-c_{red_{sub}}..+c_{red_{sub}})$ and $y \in [0..p)$.
   - $y = red_{mul}(x) = x \bmod p$ where $x \in [0..c_{red_{mul}})$ and $y \in [0..p)$

For example, we might parameterise our constraint set as

$$\begin{array}{ll} c_{max} = 16p^2 & c_{red_{add}} = 2p \\ c_{add} = 8p^2 & c_{red_{sub}} = p \\ c_{mul} = 4p & c_{red_{mul}} = 16p^2 \end{array}$$

to roughly match the SECP521R1 curve [21] implemented using either Barrett or Montgomery based arithmetic.

An Optimisation Algorithm Algorithm 1 gives a sketch of the (somewhat naive) automated approach. Using the parameterisation above and run on the code sequence for point doubling on $E(\mathbb{F}_p)$, our approach automatically produces the weight 13 solution shown in Table 1 after just a second or so of processing. This solution would be suitable, for example, in the case of the SECP521R1 curve [21]. Notice that the fact that our redundant representation has relaxed the ranges of input operands to the reduction operation red_{mul} means that we can accumulate several additive operations as unreduced intermediates, and include their reduction in a subsequent call to red_{mul} with no extra cost. The solution is not guaranteed to be optimal, but good quality solutions are found quickly; it is vital to see that this automation frees the programmer from performing the task manually, and ensures easy maintainability should F be changed and hence require re-optimisation.

4.3 Cache Consciousness

Cache memories [12], which the ARM946E-S is enabled with, are small areas of very fast memory placed between the processor and main memory. They hold a subset of main memory, the aim being to hold the working set of a program and hence accelerate memory access. However, the effectiveness of a given cache is largely determined by the composition of the access stream; typical caches work best when two principles of locality hold within the access stream. Temporal locality means that recently accessed memory addresses are likely to be accessed again in the near future. Spatial locality means that two addresses close to each other in memory will be accessed close together in time. As such, it can be attractive to restructure programs to better take advantage of the underlying cache memories; see [17] for an overview of common optimisation techniques.

Index	Original	Reordered
0	$\lambda_1 \leftarrow z_1^2$	$\lambda_1 \leftarrow z_1^2$
1	$\lambda_2 \leftarrow x_1 - \lambda_1$	$\lambda_2 \leftarrow x_1 - \lambda_1$
2	$\lambda_3 \leftarrow x_1 + \lambda_1$	$\lambda_3 \leftarrow x_1 + \lambda_1$
3	$\lambda_4 \leftarrow \lambda_2 \cdot \lambda_3$	$\lambda_4 \leftarrow \lambda_2 \cdot \lambda_3$
4	$\lambda_5 \leftarrow \lambda_4 + \lambda_4$	$\lambda_7 \leftarrow y_1 \cdot z_1$
5	$\lambda_6 \leftarrow \lambda_5 + \lambda_4$	$\lambda_5 \leftarrow \lambda_4 + \lambda_4$
6	$\lambda_7 \leftarrow y_1 \cdot z_1$	$\lambda_6 \leftarrow \lambda_5 + \lambda_4$
7	$z_3 \leftarrow \lambda_7 + \lambda_7$	$z_3 \leftarrow \lambda_7 + \lambda_7$
8	$\lambda_8 \leftarrow y_1^2$	$\lambda_{12} \leftarrow \lambda_6^2$
9	$\lambda_9 \leftarrow x_1 \cdot \lambda_8$	$\lambda_8 \leftarrow y_1^2$
10	$\lambda_{10} \leftarrow \lambda_9 + \lambda_9$	$\lambda_{14} \leftarrow \lambda_8^2$
11	$\lambda_{11} \leftarrow \lambda_{10} + \lambda_{10}$	$\lambda_9 \leftarrow x_1 \cdot \lambda_8$
12	$\lambda_{12} \leftarrow \lambda_6^2$	$\lambda_{10} \leftarrow \lambda_9 + \lambda_9$
13	$\lambda_{13} \leftarrow \lambda_{11} + \lambda_{11}$	$\lambda_{11} \leftarrow \lambda_{10} + \lambda_{10}$
14	$x_3 \leftarrow \lambda_{12} - \lambda_{13}$	$\lambda_{13} \leftarrow \lambda_{11} + \lambda_{11}$
15	$\lambda_{14} \leftarrow \lambda_8^2$	$x_3 \leftarrow \lambda_{12} - \lambda_{13}$
16	$\lambda_{15} \leftarrow \lambda_{14} + \lambda_{14}$	$\lambda_{15} \leftarrow \lambda_{14} + \lambda_{14}$
17	$\lambda_{16} \leftarrow \lambda_{15} + \lambda_{15}$	$\lambda_{16} \leftarrow \lambda_{15} + \lambda_{15}$
18	$\lambda_{17} \leftarrow \lambda_{16} + \lambda_{16}$	$\lambda_{17} \leftarrow \lambda_{16} + \lambda_{16}$
19	$\lambda_{18} \leftarrow \lambda_{11} - x_3$	$\lambda_{18} \leftarrow \lambda_{11} - x_3$
20	$\lambda_{19} \leftarrow \lambda_6 \cdot \lambda_{18}$	$\lambda_{19} \leftarrow \lambda_6 \cdot \lambda_{18}$
21	$y_3 \leftarrow \lambda_{19} - \lambda_{17}$	$y_3 \leftarrow \lambda_{19} - \lambda_{17}$

Table 2. Two orderings of operations for the point doubling $P_3 = (x_3, y_3, z_3) = 2 \cdot P_1$, given $P_1 = (x_1, y_1, z_1)$, using Jacobian projective coordinates on $E(\mathbb{F}_p)$.

With this in mind, consider the operation sequences in Table 2 which implement ECC point doubling on $E(\mathbb{F}_p)$. The left-hand sequence is what one might call the natural ordering in the sense that it is converted directly from the formula [6][Page 60]. The right-hand sequence has been reordered slightly and, while it preserves the same semantics (i.e. it computes the same result), one can think of it as having different locality properties. Specifically, the right-hand sequence exhibits better temporal locality in the instruction stream, since access to instructions that implement similar operations are grouped close together. To highlight the effect of this, we implemented the two sequences using the previously described, automatically specialised field arithmetic for the SECP521R1 curve [21]. Keeping in mind that each operation is implemented using a potentially long sequence of machine instructions, we found that over many executions the right-hand sequence caused about 100 less misses in the instruction cache, per-execution, than the left-hand sequence. Many factors would influence the previous result: the cache architecture, linking and relocation of the executable image, any form of multi-tasking, use of cache locking and so on. It should be clear however that even though the saving is small, the use of compiler techniques to automatically realise this saving is attractive.

An Optimisation Algorithm The technique of reordering the operations in Table 2 to match some goal (locality is the guiding heuristic) is a simpler version of that previously used to improve side-channel security (indistinguishability is the guiding heuristic) [5]. This suggests that the algorithm used to perform the latter optimisation could be successfully adapted to reorder sequences of instructions so that they are more cache-friendly. The result is applied early in the CAO compilation process, and is based on the generic optimisation algorithm described in Algorithm 2. The optimiser makes S attempts to find an optimal permutation of the input instruction sequence. In each of these attempts, the original function is taken as the starting solution. The inner loop uses a set of randomised heuristics which mutate the instruction sequence by introducing sound permutations (i.e. respecting inter-instruction dependencies) to obtain a neighbour solution. This solution is accepted if it does not represent a relative cost increase greater than the current threshold value. The threshold varies with t, starting at a larger value and gradually decreasing. The number of iterations S and T must be adjusted according to the size of the problem. The cost function tries to capture temporal locality quality

$$\sum_{i=1}^{n} \delta_{I_i}\omega(I_i) + \delta_{O_i[1]}\phi(O_i[1]) + \delta_{O_i[2]}\phi(O_i[2]).$$

It accumulates the potential cache-induced overhead associated with the operation (I_i) and operands ($O_i[1]$ and $O_i[2]$) at each of the n instructions in the candidate solution. It is parameterised with weight information (functions ω and ϕ) which provide a relative measure of the impact of a cache miss for each operation and operand. This weight information can also be used to bias the algorithm towards favouring instruction cache locality or data cache locality. The δ values represent the distance, i.e. the index difference, to the previous instruction where the same operation/operand has occurred (for first occurrences this is taken to be the full size of the function). The intuition behind this cost function is that cache misses are more likely as the distance between repetitions increases.

We used the algorithm described above to improve the temporal locality of two test case functions related to elliptic curve cryptography. The first function is the ECC point doubling example described Table 2. The other test case is significantly larger in size and corresponds to a point doubling for the general case of the explicit formulae for genus 2 hyper-elliptic curves over finite fields using affine coordinates. The results we obtained confirm the validity of this high-level approach, which produces a (admittedly marginal, but completely free) performance enhancement around 3%.

5 Conclusions

Thanks to a wealth of research and associated literature, implementation of ECC has been demystified to the extent that its use is no longer exclusively restricted to expert programmers. A balance to this increase in understanding is the wide range of options as regards implementation and parameterisation: even when the right algorithms and parameters are selected, the engineering and programming tasks involved in construction of a working ECC cryptosystem are far from trivial.

To address the issue of high-performance ECC implementation, we investigated the use of compilation techniques to automatically assist a programmer. The overarching goal is that the knowledge and experience of aforementioned expert practitioners can be (partially) transfered into mechanised tools to improve both productivity software quality. We introduced the CAO language and

Algorithm 2: An optimisation algorithm to improve temporal data and instruction locality within a function.

```
Input  : Instruction sequence and weight values for operations/operands.
Output: Reordered sequence with quasi-optimal temporal locality.

$result \leftarrow F$
$best \leftarrow cost(\mathbf{x})$
for $s = 1$ upto $S$ do
    $\mathbf{x} \leftarrow F$
    $cost \leftarrow cost(\mathbf{x})$
    for $t = 1$ upto $T$ do
        $\mathbf{x}' \leftarrow neighbour(\mathbf{x})$
        $thresh \leftarrow threshold(t, T)$
        $cost' \leftarrow cost(\mathbf{x}')$
        if $(cost'/cost - 1) < thresh$ then
            $\mathbf{x} \leftarrow \mathbf{x}'$
            $cost \leftarrow cost'$
    if $cost < best$ then
        $result \leftarrow \mathbf{x}$
        $best \leftarrow cost$
return $result$
```

associated compiler as a means of naturally describing cryptographically interesting programs. These programs can be analysed by the compiler and undergo cryptography-aware analysis, transformation and optimisation phases.

Acknowledgements

The authors would like to thank Gregory Zaverucha for pointing out the automated method to generate modular reduction functions for arithmetic in $\mathbb{F}_p$.

References

1. D. Agrawal, B. Archambeault, J.R. Rao and P. Rohatgi. The EM Side-Channel(s). In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 2523, 29–45, 2002.
2. ARM Limited. *ARM946E-S Technical Reference Manual.* Available from: `http://www.arm.com/documentation/`
3. R.M. Avanzi. Aspects of Hyperelliptic Curves over Large Prime Fields in Software Implementations. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 3156, 148–162, 2004.
4. P.D. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology (CRYPTO)*, Springer-Verlag LNCS 263, 311–323, 1986.
5. M. Barbosa and D. Page. On the Automatic Construction of Indistinguishable Operations. In *Cryptology ePrint Archive*, Report 2005/174, 2005.

6. I.F. Blake, G. Seroussi and N.P. Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, 1999.
7. I.F. Blake, G. Seroussi and N.P. Smart. *Advances in Elliptic Curve Cryptography.* Cambridge University Press, 2004.
8. H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, CRC Press, 2005.
9. Computational Algebra Group, University of Sydney. *Magma Computational Algebra System.* Available from: `http://magma.maths.usyd.edu.au/magma/`
10. C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E-N. Volanschi, J. Lawall and J. Noyá. Tempo: Specializing Systems Applications and Beyond. In *ACM Computing Surveys*, **30** (3), 1998.
11. D. Hankerson, A. Menezes and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, 2004.
12. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2006.
13. N. Koblitz. Elliptic Curve Cryptosystems. In *Mathematics of Computation*, **48**, 203–209, 1987.
14. N. Koblitz. Hyperelliptic Cryptosystems. *Journal of Cryptology*, **1** (3), 139–150, 1989.
15. P.C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology (CRYPTO)*, Springer-Verlag LNCS 1109, 104–113, 1996.
16. P.C. Kocher, J. Jaffe and B. Jun. Differential Power Analysis. In *Advances in Cryptology (CRYPTO)*, Springer-Verlag LNCS 1666, 388–397, 1999.
17. M. Kowarschik and C. Wei. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies*, Springer-Verlag LNCS 2625, 213–232, 2003.
18. V. Miller. Uses of Elliptic Curves in Cryptography. In *Advances in Cryptology (CRYPTO)*, Springer-Verlag LNCS 218, 417–426, 1985.
19. P.L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, **44**, 519–521, 1985.
20. D. Page. *CAO : A Cryptography Aware Language and Compiler*. Available from: `http://www.cs.bris.ac.uk/home/page/research/cao.html`
21. Standards for Efficient Cryptography Group (SECG). *SEC 2: Recommended Elliptic Curve Domain Parameters*, 2000. Available from: `http://www.secg.org`
22. V. Shoup. *NTL: A Library for doing Number Theory.* Available from: `http://www.shoup.net/ntl/`
23. J.A. Solinas. Generalized Mersenne Numbers. *Technical Report CORR 99-39*, University of Waterloo, 1999.
24. C.D. Walter. Montgomery Exponentiation Needs No Final Subtractions. *Electronics Letters*, **35**, 1831–1832, 1999.