J. Cryptol. (2009) 22: 259–281
DOI: 10.1007/s00145-008-9023-0

Journal of
CRYPTOLOGY

Constructive and Destructive Use of Compilers in Elliptic Curve Cryptography

M. Barbosa, A. Moss, and D. Page
Departamento de Informática, Universidade do Minho, Campus de Gualtar, 4710-057 Braga, Portugal
mbb@di.uminho.pt
and
Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road,
Bristol, BS8 1UB, UK
moss@cs.bris.ac.uk; page@cs.bris.ac.uk

Received 27 February 2007 and revised 21 April 2008
Online publication 15 May 2008

Abstract. Although cryptographic software implementation is often performed by expert programmers, the range of performance and security driven options, as well as more mundane software engineering issues, still make it a challenge. The use of domain specific language and compiler techniques to assist in description and optimisation of cryptographic software is an interesting research challenge. In this paper we investigate two aspects of such techniques, focusing on Elliptic Curve Cryptography (ECC) in particular. Our constructive results show that a suitable language allows description of ECC based software in a manner close to the original mathematics; the corresponding compiler allows automatic production of an executable whose performance is competitive with that of a hand-optimised implementation. In contrast, we study the worrying potential for naïve compiler driven optimisation to render cryptographic software insecure. Both aspects of our work are set within the context of CACE, an ongoing EU funded project on this general topic.

Key words. Elliptic curve cryptography (ECC), Implementation, Compilers, Optimisation, Specialisation.

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 security which represents a restrictive overhead within many applications. Not only must a given device execute algorithms that satisfy the application context, for example, the use of digital signatures on smart-cards, but increasingly it 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 by passive monitoring of features such as timing variation [29], power consumption [30] or electromagnetic emission [1].

Elliptic Curve Cryptography (ECC) offers a popular solution to the problem of implementing public key cryptography on mobile computing devices. 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 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 [10, Chaps. 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 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 naturally support the types and operations required and thus cannot perform optimisation and analysis phases typically offered when writing more conventional software. For example, the compiler cannot apply basic optimisations such as register allocation; it cannot detect or resolve security related errors as it might do with errors relating to functional correctness. As a result, cryptographic software is often 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.

An interesting research challenge is presented by the potential to use domain specific languages and compilation techniques in the presented context. The hope is that programmers using the results of such research 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 using automated analysis, optimisation and transformation, a programmer will improve their productivity, reduce their rate of error and generally produce software of a higher quality. Systems such as Cryptol [32], Sokrates [14], LaCodA [34] and SIMAP [39] have started to address this issue at various levels. Focusing on ECC based primitives, so that our domain is slightly orthogonal to previous work, we investigate three overarching topics: description of ECC based primitives in a natural manner using the CAO [40] language; automatic optimisation of those primitives using novel extensions to the CAO compiler; and the security implications of using specific forms of automation. Our results are intentionally exploratory, and we do not present or analyse a complete system. Instead, we set our work within the context of CACE,[1] an ongoing EU funded project on this general topic; the CACE project has the broad remit of maturing research such as that presented here, and producing robust tools from the result.

[1] http://www.cace-project.eu/.

The paper is organised as follows. We use Sect. 2 to present background material including a brief overview of the fundamentals behind ECC and a description of our experimental platform. In Sect. 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 Sect. 4: we focus on automatic specialisation of field arithmetic in Sect. 4.1, placement of modular reduction operations in Sect. 4.2, and cache conscious ordering of field operations in Sect. 4.3. In Sect. 5 we turn our attention to security aspects of the resulting implementation. We show in Sect. 5.1 how to automatically construct a class of side-channel secure programs, and in Sect. 5.2 how the concept of dynamic compilation can effect the security of such programs. We present some conclusions in Sect. 6.

2. Background

An Introduction to ECC Elliptic Curve Cryptography (ECC) was invented during the mid 1980s in independent work by Miller [36] and Koblitz [27], then generalised to include Hyperelliptic Curve Cryptography (HECC) by Koblitz [28] in 1989. We concentrate here only on ECC, for further reading on all issues covered in this basic introduction, see Menezes et al. [23] or Blake et al. [9,10]. Briefly then, an elliptic curve E over the finite field K is defined by the general Weierstrass equation, for $a_i \in K$,

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

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:

$$x_3 = \lambda^2 + a_1\lambda - a_2 - x_1 - x_2,$$

$$y_3 = (x_1 - x_3)\lambda - y_1 - a_1x_3 - a_3,$$

where

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

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.

Most ECC based schemes use the additive group structure presented by the points on E as a means to present a discrete logarithm problem as the basis for security. The Elliptic Curve Discrete Logarithm Problem (ECDLP) is constructed by considering scalar multiplication of a point $P \in E$ by the integer value d expressed as $Q = d \cdot P$ or, expanding the right-hand side to give a more natural description

$$Q = \underbrace{P + P + \cdots + P + P}_{\text{total of } d \text{ summands}}.$$

Given the values of d and P, it is easy to calculate Q using an additive version of common exponentiation algorithms. However, given only the values of P and Q, the value of d is computationally hard to recover.

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 to 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 commonly used is 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

$$E : Y^2 + a_1 XYZ + a_3 YZ^3 = X^3 + a_2 X^2 Z^2 + a_4 XZ^4 + a_6 Z^6.$$

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

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

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

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

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 [4] 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 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.

A caveat to the consistent use of this platform comes in Sect. 5.2, where we present an experiment into how dynamic compilation in Java impacts on issues of side-channel security. In this particular case, we implemented our experimental code in Java and executed it on Jikes Research Virtual Machine (RVM) [2]. Since we were interested in a proof of concept, porting the RVM to our ARM platform was too costly in terms of time; since the experiments are targeted at behaviour of the virtual machine rather than the host, we suggest the choice of host does not vastly alter our findings. As such, we opted to host the RVM on a 2.8 GHz Pentium 4 processor with an 8-kB L1 data cache and a 512-kB unified L2 cache. The RVM was compiled from source using GCC 4.0.1.

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

```
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 [9, page 60], the right-hand side is the associated translation into CAO.

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, removes the associated tedium and error. 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, interpreted languages which support the types and operations required, such as Magma [17], are unlikely to yield efficient results on a mobile computing device. Conversely, using a language which supports efficient compilation, such as C, seldom results in easy translation since there is typically no natural support for required types and operations.

As a means of allowing common compilation techniques to be applied to natural descriptions of ECC, we present the CAO language and associated compiler system [40]. 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 structured so that back-end code generation can be replaced to target any platform; in this specific case it produces an NTL [43] based implementation detailed in Fig. 2, which closely matches that one would construct by hand, using a range of standard optimisation techniques:

Register Allocation One of the most tedious tasks in implementing sequences of operations on types not supported by a language is finding an allocation of temporary variables which is efficient, i.e. has an acceptably small memory footprint. Fortunately, this problem is well studied in the context of conventional compilers [38, Chap. 16].

Strength Reduction A common implementation task is the conversion of multiplication or exponentiation by small constants into an efficient addition chain that is less

```
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 Fig. 1, into an NTL based function (with slight hand modification used to improve readability).

costly than the naive description. An example is the computation of $8y_1^4$ in Fig. 1. Performing this optimisation manually obfuscates the program; early optimisation like this is a poor choice since it limits both understanding of the algorithm and portability of the program which implements it. For example, we might make an assumption about the cost ratio between addition and multiplication and hard code it into our implementation. If this assumption is false for a given target platform, we must rethink the implications of our optimisation and potentially replace it with something more appropriate. However, the compiler can perform this optimisation automatically given knowledge of the types and operations involved; this is a type of strength reduction [38, Chap. 12]. Further, since it has a knowledge of the target architecture (and hence the relative costs of operations) it can selectively apply the optimisation to give the best result. We typically only need to compute chains for small values and can achieve reasonable results with a basic algorithm, for example, the power-tree method of Knuth [26, Sect. 4.6.3].

Common Sub-expression Elimination (CSE) Intermediate results can potentially be shared between different parts of the computation without re-computation, should there be enough registers to accommodate them. An example is the reuse of the value y_1^2 to compute y_1^4 via one squaring in Fig. 1. Again, performing the optimisation manually obfuscates the algorithm and ties us to assumptions that may not hold on a given target platform. Again, the compiler can perform this optimisation automatically given it knows about the types and operations involved; this is a type of common sub-expression elimination [38, Chap. 13].

4. Optimisation of Curve Arithmetic

4.1. *Specialisation of Field Arithmetic*

The description of ECC in Sect. 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 [6] 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. 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 automatic generation of special purpose run-time support libraries from a corpus of general purpose library code. That is, if the CAO program uses a given finite field within the high-level program, the compiler instructs an auxiliary system to construct a run-time library specifically to support operations in that field (i.e. replace the calls to NTL produced by the initial compiler back-end in Sect. 3). This idea has recently been addressed by the $\mathtt{mp}\mathbb{F}_q$ system of Gaudry and Thomé [21], who use special purpose code generation; in contrast, we use the Tempo [18] specialiser which accepts arbitrary C source code as input.

Table 3 compares the performance of three different run-time libraries for arithmetic in the fields $\mathbb{F}_{2^n}$ and $\mathbb{F}_p$ for values of n and p that match those used in the SECG recommended domain parameters [42]. 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 libraries were constructed as follows:

Implementation A Entirely generic, hand written 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 [23, pages 52–53], multiplication was performed using the right-to-left comb method [23, pages 48–51], the reduction used a generic word-wise approach [23, pages 53–56]. For arithmetic in $\mathbb{F}_p$, standard integer addition [23, page 20], multiplication [23, page 31] and squaring [23, page 35] were used; modular reduction was performed using the method of Barrett [23, page 36].

Implementation B Same as Implementation A except that several core functions were specialised by hand to remove the most 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 [23, pages 44–46 and 53–56].

	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 of operation processing time (in microseconds) between three different implementations of arithmetic in $\mathbb{F}_{2^n}$ and $\mathbb{F}_p$ for standard values of n and p.

Implementation C Has the same forms of specialisation as Implementation B but applied automatically using Tempo. 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 [44]. This partly covers the fact that Tempo cannot make algorithmic selections on behalf of the programmer, e.g. Barrett reduction versus Solinas reduction for a given p.

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 absolute timings from Fig. 3 are somewhat unimportant and are highly related to higher-level algorithmic choices. The key thing to note from this experiment is that with the caveat that any specialisation needs to be performed in context with the application, one can automatically produce a specialised run-time library which is competitive with a hand written alternative. This positive result in terms of performance was achieved in a fraction of the time in terms of programmer effort.

4.2. *Lazy Reduction*

Avanzi [6, Sect. 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 [7] 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 $b = 2^w$. 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 [42] 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 [37] 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 [46] 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 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 input 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 [42] 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 [42]. 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.

We used the algorithm described above to produce exactly the reduction points shown in Table 1 which describes an operation sequence for ECC point doubling. In conjunction with the standard SECP521R1 curve and using our ARM based experimental platform (with field arithmetic produced by the specialiser described in Sect. 4.1), we benchmarked the operation sequence and found that the optimised version improved the overall execution time by roughly 2%. Note that this figure is closely tied to the operation sequence in question and choice of underlying field. Although in this case the improvement is admittedly marginal, it is crucial to note that it comes entirely for free: there no extra effort by the programmer. Furthermore, although the solution is not guaranteed to be optimal, the automated approach ensures easy maintainability should the high-level implementation be changed and hence require re-optimisation.

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 $S$
```

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 $E(\mathbb{F}_p)$).

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}

4.3. *Cache Consciousness*

Cache memories [24], 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. Typical caches are most effective when two principles of locality hold in the address stream: (1) temporal locality, that recently accessed memory addresses are likely to be accessed again in the near future (2) spatial locality, that two addresses close to each other in memory will be accessed close together in time. It can therefore be attractive to restructure programs to better take advantage of the underlying cache memories; see [31] for an overview of common optimisation techniques.

In the remainder of this section we describe how we build on this observation in the CAO compiler system. Our goal is to enable the compiler to automatically restructure a high-level program so it is more cache conscious, and hence more efficient as highlighted above. Our approach is related to that of Sermulins et al. [41] where high-level cache-aware optimisations are applied in the compilation of a domain specific language for streaming applications. Note that Gupta et al. [22] show that a variant of the cache-aware instruction scheduling problem, expressed using a graph-based formalism very much in line with the one adopted in this work, is NP-complete.

An Optimisation Approach The proposed cache aware scheduling technique is applied within the high-level optimisation phase of the CAO compiler and is therefore subject to some restrictions. High-level operations, such as finite field calculations, are seen as atomic instructions. Although the compiler will have some knowledge of the run-time library, this is pre-compiled code which is not accessible for optimisation. Furthermore, since the CAO compiler is designed to target multiple platforms by replacing the back-end, any cache-oriented optimisation introduced at this level must be, to some extent, independent of target-specific details. Despite these restrictions, we show that it is possible to obtain performance benefits by introducing high-level cache-aware compiler optimisations early in the compilation process. For this, we present an algorithm that performs the analogy of heuristically guided instruction scheduling within a conventional compiler. The algorithm processes straight line instruction sequences and seeks to improve the temporal locality properties, which we formulate as an optimisation problem. We begin by defining a problem instance.

Definition 1. Let F be a function constituted of a list of instructions, each of them executing an operation I from a finite operation set L. Each instruction reads the values of operands $O[1]$ and $O[2]$[2] and places the result in an operand D, all from a finite set of operands O. More precisely, let $F = F[1], \ldots, F[|F|]$, where $|F|$ denotes the length of function F, and

$$F[i] = \big(I_i, D_i, O_i[1], O_i[2]\big) \in L \times O \times O \times \big(O \cup \{\perp\}\big)$$

denotes instruction i of function F, with $1 \leq i \leq |F|$.

We aim to manipulate the original function into a new version F' such that instructions which access the same data memory position and/or use the same pre-compiled run-time library operation are closer together. To allow for instruction reordering, we extend our problem definition to include information about the data dependencies between instructions within each function. We represent these dependencies as directed graphs.

[2] We allow $O[2]$ to take the special value $\perp$ to capture the possibility that some operations take only one operand, e.g. a squaring or a doubling.

Definition 2. Given a set F as in Definition 1, let P be a pair $P = (F, G)$, where $G = (V, E)$ is a directed graph in which V and E are the associated sets of nodes and edges, respectively. Let $|V| = |F|$ and, to each instruction $F[i]$, associate node $v_i \in V$. Let E contain an edge from node v_i to node v_j if and only if executing instruction $F[i]$ before instruction $F[j]$ disrupts the normal data flow inside the function. We say that instruction $F[i]$ depends on instruction $F[j]$.

We use the dependency graphs in Definition 2 to guarantee that the transformations we perform on the functions F are sound. That is, as long as we respect the dependencies, the program is functionally correct, even though the instructions are reordered. Definition 3 captures this notion.

Definition 3. A function F' is a valid transformation of a function F (written $F' \Lleftarrow F$) if F' can be generated reordering the instructions in F respecting the dependency graph G, i.e. if there is an edge $(v_i, v_j) \in E$, then instruction $F[i]$ must occur after instruction $F[j]$ in F'.

The goal is to find F' whose temporal locality properties imply a reduction (or ideally a minimisation) of the overhead due to cache accesses during execution. An instruction reading (resp. using code) from a memory location which is not currently in the data (resp. instruction) cache will cause an access to main memory; this is termed a cache-miss. Roughly speaking, our aim is to minimise cache misses by maximising temporal locality in the data and instruction streams.

Since we want our problem formulation to be at a high level of abstraction, our approach to approximating said overheads is straightforward. We assign an integer weight to each basic instruction in set L and, similarly, to each operand in the set O. This value provides a relative measure for the cost of loading the instruction code or the operand data into the cache if it is not already there when a particular instruction is executed. In practise these values should increase with the sizes of the memory representations of operations and operands. They can also be used to bias the optimality criteria into favouring operation locality over operator locality and vice-versa. It is through these values that the compiler can tune the solution search to match the characteristics of a particular run-time library or a particular target platform.

Definition 4. Let $\omega : L \to \mathbb{N}$ be a weight function that, for each basic instruction $l \in L$, provides a relative value $\omega(l)$ for the cache miss overhead associated with loading instruction l. Similarly, let $\phi : O \to \mathbb{N}$ be a weight function that, for each operand $o \in O$, provides a relative value $\phi(o)$ for the cache miss overhead associated with loading operand o.

Given these cost definitions, we are now in a position to provide a formulation of the problem of optimising the temporal locality of a function.

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

```
Input  : $(P, \omega, \phi)$
Output: $F'$, a quasi-optimal solution to the problem in Definition 5
$result \leftarrow F$, $best \leftarrow cost(\mathbf{x})$
for $s = 1$ upto $S$ do
    $\mathbf{x} \leftarrow F$, $cost \leftarrow cost(\mathbf{x})$
    for $t = 1$ upto $T$ do
        $thresh \leftarrow threshold(t, T)$
        $\mathbf{x}' \leftarrow neighbour(\mathbf{x})$, $cost' \leftarrow cost(\mathbf{x}')$
        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 $result$
```

Definition 5. Given a tuple (P, ω, ϕ) as in Definitions 1, 2 and 4, find a function F' such that $F' \Lleftarrow F$ and that

$$\sum_{i=1}^{|F'|} \delta_{I_i}\omega(I_i) + \delta_{O_i[1]}\phi\big(O_i[1]\big) + \delta_{O_i[2]}\phi\big(O_i[2]\big)$$

is minimal. 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 $|F'|$. For the special case of $O_i[2] = \bot$, it is 0.

The intuition behind the cost function in Definition 5 is that first occurrences of operations and operands invariably cause cache misses; for repeated occurrences, misses are more likely as the distance between repetitions increases.

An Optimisation Algorithm Our approach to solving the problem as described above is detailed in Algorithm 2. The algorithm represents an adaptation of Threshold Accepting [20], a generic optimisation algorithm and a close relative of simulated annealing. Note that we are not aiming to find the optimal solution but to find a good enough approximation of it that can be used in practical applications. A neighbour solution is derived from the current solution by randomly selecting a random mutation from a small set of heuristic transformations. These generally consist of choosing a random instruction and moving it gradually to another position where it is closer to another instruction which uses the same operands or operations. Mutations are accepted if they increase the solution cost by less than a threshold which 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.

We used the algorithm described above to improve the temporal locality of an operation sequence for ECC point doubling; the results are described by Table 2. The

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 $E(\mathbb{F}_p)$.

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

left-hand sequence is the natural ordering in the sense that it is converted directly from the formula [9, page 60]. The right-hand sequence, produced by the algorithm, exhibits better temporal locality in the instruction stream, since access to instructions that implement similar operations are grouped close together. Again in conjunction with the standard SECP521R1 curve and using our ARM based experimental platform (with field arithmetic produced by the specialiser described in Sect. 4.1), we benchmarked the two operation sequences and found that the optimised version provoked around 1% less instruction cache misses which improved the overall execution time by roughly the same amount, i.e. 1%. In a similar way to the lazy reduction example, this figure is closely tied to the operation sequence in question and choice of underlying field. For this specific example, the improvement is again marginal, but again it is provided entirely for free in terms of programmer and maintenance effort.

5. Security of Curve Arithmetic

Even when security proofs for cryptographic schemes hold, the security of underlying primitives is increasingly being attacked from a physical rather than mathematical point of view. Typically attacks consist of a collection phase, which provides the attacker with profiles of execution, and an analysis phase, which recovers the secret information from the profiles. Considering power consumption as the collection medium, attack methods can be split into two main classes. Simple Power Analysis (SPA) is where the attacker is given only one profile and is required to recover secret information by focusing mainly on the operation being executed. In contrast, Differential Power Analysis (DPA) uses statistical methods to form a correlation between a number of profiles and the secret information by focusing mainly on the data items being processed. These passive attacks can be extended by considering active, fault injection methods whereby the attacker can physically manipulate the host platform to make it perform erroneous operations.

For a more concrete example, consider the multiplication of a point by an integer d. This core operation in most ECC based systems can be realised using an additive the left-to-right double-and-add method [23, page 97], an additive version of binary exponentiation. In this algorithm, for a random d, the point doubling operation will be used about twice as often as the point addition operation. The algorithm goes through the d exponent bits and performs a doubling operation for each of them. In each iteration, if $d_i = 1$, the algorithm performs also a point addition operation, where d_i denotes the ith bit of integer d.

The actual point addition and doubling operations are often distinguishable from each other in a profile of execution because one is composed from a different sequence of finite field operations than the other. Using A and D to denote point addition and doubling respectively, the collection phase of an SPA attack presents the attacker with a profile detailing the operations performed during execution of the algorithm. For example, by monitoring execution of using the multiplier $d = 1001_2 = 9_{10}$, one obtains the profile $DDDA$. Given this single profile, the analysis phase can recover the secret value of d simply by spotting where the point additions occur. If the sequence DA occurs during iteration i, we have that $d_i = 1$, whereas if the sequence D occurs, then $d_i = 0$.

One way to avoid this problem is to employ the double-and-add-always method due to Coron [16]. In this case, a dummy addition is executed if the real one is not. Although the cases where $d_i = 0$ and $d_i = 1$ are now indistinguishable, this method imposes a significantly performance penalty since many more additions are performed than is necessary. A more considered approach involves using the flexibility of the curve group law in terms of how the point addition and doubling operations can be implemented through different curve parameterisations, point representations and so on. For example, one can manipulate the formula for point addition and doubling so that they are no longer different [13,25,33].

Finally, one can achieve an approximation of the previous approach by manipulating the implementation of these operations so that, when their execution is observed, they appear to be the same. In this particular case this can be achieved, for example, by splitting the more expensive point addition into two parts, each of which is identical in terms of the operations it performs to a point doubling. Put more simply, instead of recovering the profile above from the SPA collection phase, an attacker gets $XXXXX$, where X represents an atomic, indistinguishable operation which could be a double or one step in an addition; from this the attacker gets no useful information other than about the Hamming weight and size of d. Trichina and Bellezza [45] analyse the overhead and effectiveness of this approach using Jacobian projective coordinates on standard curves over $K = \mathbb{F}_{2^{163}}$. Table 3 details their manipulated formula for point doubling and addition; note that C is a pre-computed constant. Also note that the each of the two addition steps has been made to use the same sequence of operations as the doubling formula by the inserting a total of eight dummy operations, these are denoted by use of register $\lambda_\perp$. Chevallier-Mames et al. [15] propose a generalised formulation for constructing indistinguishable functions. SPA attacks typically exploit conditional instructions that depend on secret information: the solution is to make the sequences of instructions (processes) associated with both branches indistinguishable. The authors introduce the concept of side channel atomicity. All processes are transformed simply by padding them with dummy operations, so that they execute as a repetition of a small instruction sequence (a pattern) called a side-channel atomic block.

Table 3. Indistinguishable formula for point doubling $P_3 = (x_3, y_3, z_3) = 2 \cdot P_1$, given $P_1 = (x_1, y_1, z_1)$, and point addition $P_3 = (x_3, y_3, z_3) = P_1 + P_2$, given $P_1 = (x_1, y_1, z_1)$ and $P_2 = (x_2, y_2, z_2)$, using Jacobian projective coordinates on $E(\mathbb{F}_{2^n})$.

Index	Doubling	Addition Step 1	Addition Step 2
0	$\lambda_1 \leftarrow z_1^2$	$\lambda_1 \leftarrow z_1^2$	$\lambda_{11} \leftarrow z_3^2$
1	$\lambda_3 \leftarrow y_1 \cdot z_1$	$\lambda_2 \leftarrow \lambda_1 \cdot x_2$	$\lambda_{12} \leftarrow z_3 \cdot y_2$
2	$z_3 \leftarrow x_1 \cdot \lambda_1$	$\lambda_3 \leftarrow \lambda_1 \cdot z_1$	$\lambda_{13} \leftarrow \lambda_8 \cdot \lambda_{10}$
3	$\lambda_4 \leftarrow x_1^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$	$\lambda_{14} \leftarrow \lambda_7^2$
4	$\lambda_5 \leftarrow z_3 + \lambda_4$	$\lambda_\perp \leftarrow \lambda_\perp + \lambda_\perp$	$\lambda_{15} \leftarrow \lambda_{11} + \lambda_{13}$
5	$\lambda_6 \leftarrow C \cdot \lambda_1$	$\lambda_6 \leftarrow y_2 \cdot \lambda_3$	$\lambda_{16} \leftarrow \lambda_7 \cdot \lambda_{14}$
6	$\lambda_7 \leftarrow x_1 + \lambda_6$	$\lambda_7 \leftarrow x_1 + \lambda_2$	$x_3 \leftarrow \lambda_{15} + \lambda_{16}$
7	$\lambda_8 \leftarrow \lambda_4^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$
8	$\lambda_9 \leftarrow \lambda_8 \cdot z_3$	$z_3 \leftarrow z_1 \cdot \lambda_7$	$\lambda_{17} \leftarrow x_3 \cdot \lambda_{10}$
9	$\lambda_{10} \leftarrow \lambda_5 + \lambda_3$	$\lambda_8 \leftarrow y_1 + \lambda_6$	$\lambda_{18} \leftarrow \lambda_9 + \lambda_{12}$
10	$\lambda_{11} \leftarrow \lambda_7^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$
11	$x_3 \leftarrow \lambda_{11}^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$	$\lambda_\perp \leftarrow \lambda_\perp^2$
12	$\lambda_{12} \leftarrow \lambda_{10} \cdot x_3$	$\lambda_9 \leftarrow \lambda_8 \cdot x_2$	$\lambda_{19} \leftarrow \lambda_{18} \cdot \lambda_{11}$
13	$y_3 \leftarrow \lambda_9 + \lambda_{12}$	$\lambda_{10} \leftarrow z_3 + \lambda_8$	$y_3 \leftarrow \lambda_{17} + \lambda_{19}$

5.1. *Automatic Indistinguishability*

Although the general technique of creating indistinguishable functions to foil SPA style attack is well understood, a general barrier to implementation is how labour intensive and error prone the process is. This is especially true when operation sequences in the functions are more complex than in the stock example of ECC, for example, systems like XTR or HECC. However, the task is ideally suited to automation; to this end our focus in this section is the realisation of such automation to assist the development of secure software.

An Optimisation Approach We present the problem of building indistinguishable functions as an optimisation problem, following a similar approach to that in Sect. 4.2.

Definition 6. Let F be a list of N functions $F = F_1, F_2, \ldots, F_N$, where each function F_i is itself a list of instructions from a finite instruction set L:

$$F_i = F_i[1], F_i[2], \ldots, F_i\big[|F_i|\big],$$

where $|F_i|$ denotes the length of function F_i, and $F_i[j] \in L$ denotes instruction j of function F_i, with $1 \le j \le |F_i|$. Also, let $F_i[k..j]$ denote instructions k to j in function F_i, with $1 \le k \le j \le |F_i|$.

For concreteness one should think of the simple case of two functions F_1 and F_2 as performing ECC point addition and doubling. Further, the instruction set L is formed from three-address style operations [38] on elements in the base field, for example, addition and multiplication, and the functions are straight-line in that they contain no internal control flow structure.

We aim to manipulate the original functions into new versions F_i' such that the execution trace of all of them is some multiple of the execution trace of a shorter sequence.

We term this shorted sequence Π, the fixed pattern of operations which is repeated to compose the larger functions. Clearly we might need to add some dummy instructions to the original functions as well as reordering their instructions so that the pattern is followed. Again, we use the dependency graphs from Definition 2 to guarantee that the transformations we perform on the functions F_i are sound. We also extend the notion of valid transformation to include the insertion of dummy operations.

The goal is to find Π and matching F_i' whose processing overhead compared to the original programs is minimised. Hence, our problem definition must also include the concept of computational cost. For the sake of generality, we assign to each basic instruction in set L an integer weight value that provides a relative measure of its computational weight. We represent this weight as $\omega(l)$ for a particular instruction l. We define the optimisation problem of building indistinguishable functions as follows.

Definition 7. Given a pair (P, ω) as in Definition 6 and the natural extension of Definition 2, find a pattern Π and a list of functions $F' = F_1', F_2', \dots, F_N'$ such that

$$\begin{cases} \Pi = \Pi[1], \Pi[2], \dots, \Pi[|\Pi|], & \Pi[k] \in L, 1 \le k \le |\Pi|, \\ F_i' \models F_i, & 1 \le i \le N, \\ |F_i'| = 0 \pmod{|\Pi|}, & 1 \le i \le N, \\ F_i'[j] = \Pi[(j \bmod |\Pi|) + 1], & 1 \le i \le N, 1 \le j \le |F_i'| \end{cases}$$

and that the summation $\sum_{i=1}^{N} \sum_{j=1}^{|F_i'|} \omega(F_i'[j])$ is minimal.

Intuition on the hardness of satisfying these constraints comes from noticing similarities with well-known NP-complete optimisation problems such as the Minimum Bin Packing, Longest Common Subsequence and Nearest Codeword problems [19].

An Optimisation Algorithm Our approach to solving this problem uses an algorithm which is almost identical to Algorithm 2. The optimisation algorithm makes S attempts to find an optimal pattern size, which is selected randomly in each s-iteration. In each of these attempts, the original functions are taken as the starting solution, with the minor change that they are padded with dummy instructions, so as to make their size multiple of the pattern size. The quality of a solution is measured using a cost function that sees the complete set of instructions in a solution $\mathbf{x}$ as a matrix with $|\Pi|$ columns and $(\sum_{i=1}^{N} |F_i'|)/|\Pi|$ rows, in which each function occupies $|F_i'|/|\Pi|$ consecutive rows.

During optimisation, the pattern Π is adjusted to each solution by taking $\Pi[k]$ as the most common basic instruction in column k of the matrix. A dummy instruction is always taken to be of the same type as the pattern instruction for its particular column, so dummy instructions are ignored when adjusting Π to a particular solution. The overall cost of a solution has two components: (1) the cost associated with deviations from the pattern evaluated as the sum, taken over all non-matching instructions in the matrix, of the weight difference relative to the corresponding pattern instruction; and (2) the cost associated with dummy operations evaluated as the accumulated weight of all the dummy instructions in the matrix. The relative importance of these components can be tuned to put more or less emphasis on indistinguishability. This affects the trade-off between indistinguishability and processing overhead.

Again, a neighbour solution is derived from the current solution by randomly selecting one mutation from a small set of heuristic transformations. In this case, the transformations consist of moving instructions within a particular function, filling spaces previously occupied by dummy instructions or just freed by moving other instructions, until they match the pattern instruction in the target column. After application of the selection heuristic, the neighbour solution is optimised by removing rows or columns containing only dummy instructions.

If the final solution produced by the optimisation algorithm includes deviations from the chosen pattern, these can be eliminated in an optional post-processing phase. In this phase, we increase the pattern size to cover the mismatches and introduce extra dummy operations to retain indistinguishability. If the number of mismatches is large, this produces a degenerate solution which is discarded due to the high related cost.

Using this optimisation algorithm, we have been able to produce results equivalent to various hand-made solutions published in the literature for small sized problems, and to construct indistinguishable versions of the much larger functions for point addition and doubling in genus 2 hyper-elliptic curves over finite fields. To save space, we refer to the full version of this work [8] for a complete set of results.

5.2. *A Compilation Related Vulnerability*

An Overview of Dynamic Compilation Dynamic compilation techniques have been a major factor in allowing interpreted languages, such as Java, to satisfy the demands of high-performance application areas. The basic idea is that the interpreter, from here on termed the virtual machine, monitors execution of a program at run-time. By profiling aspects of execution such as frequency of calls to a given method, the virtual machine can identify so called hotspots: regions of the program which are critical to performance. Once these regions are identified, the virtual machine invokes an aggressive re-compilation phase on them that results in either more efficient interpretable code or native code that can be executed directly by the host platform. By using the re-compiled code, execution can proceed more efficiently and without sacrificing portability of the original program. For all the positive aspects of this highly active research area, the basic premise that code written by the programmer might not be the same code executed by the host platform has some serious implications in terms of security. For example, one typically demands that the implementation of a cryptographic primitive exactly matches the specification so that security proofs can be valid.

Both side-channel and fault injection attacks are commonly set in the context of devices such as smart-cards and embedded processors since they generally require that the attacker has physical access to the processing running the code under attack. The security risks here are exacerbated by the use of both types of device as conduits for sensitive financial or identity related information. However, Boneh and Brumley [11] and Bernstein [12] recently showed that timing attacks against remove servers are even possible without physical access to the target. The use of Java in both these areas hints at the problems one might expect. In particular, we motivate our work by noting the link between the fields of dynamic compilation and physical security that occurs as a result of the potential use of Just In Time (JIT) compilers on JavaCard based multi-application smart-cards. Examples include Jazelle enabled ARM processors which accelerate execution of Java in embedded contexts and include support for JIT compilation [3]. These

Table 4. Results, with time in terms of microseconds, for elliptic curve point doubling and addition with and without dynamic compilation.

Method	With AOS	Without AOS
Doubling	0.03159	0.80985
Addition Step 1	0.02527	0.81016
Addition Step 2	0.03776	0.80993

types of JavaCard are widely used as a platform for rapid development of smart-card based applications; dynamic compilation allows the physical devices to remain simple and inexpensive while delivering acceptable levels of performance.

Dynamic Compilation Versus Side-Channel Security Using the indistinguishable formula in Table 3, we implemented an SPA resistant double-and-add based point multiplication program in Java and ran it on the Jikes Research Virtual Machine (RVM) [2]. Using default options for the adaptive optimisation system (AOS) [5], RVM made an attempt to dynamically re-compile both the point doubling and addition methods as well as constituent methods for finite field arithmetic. As a product of aggressive inlining and subsequent optimisation, the timings in Table 4 show significant improvement, and slight skewing, of the unoptimised code.

The main point of note is that an execution profile of the optimised doubling method will differ radically from non-optimised version of the addition method. Even though we have gone to some lengths to make them indistinguishable from each other, the optimised versus non-optimised code will be distinguishable by virtue of their significantly different composition. Further, the time that re-compilation of these methods takes place is of interest. From the RVM log, we found that the doubling method was re-compiled at least 6 milliseconds and as much as 20 milliseconds before the addition methods depending on the value of d. Intuitively this makes sense: if the doubling method is called more often, it is more likely to be quickly identified as a hotspot by either counter or sampling based monitors. Given that the RVM carries out re-compilation in the background, one could expect a window whereby the optimised double method is in use at the same time as the non-optimised addition method.

Figure 4 details the general timeline of execution more simply. The point multiplication begins at time t_0 with both the double and addition methods in an non-optimised form. Then, at time t_1 the double method is identified as a hotspot and re-compiled; this leads to the double and addition methods now being distinguishable from each other due to their differing composition. At time t_2 the addition methods are re-compiled but are still distinguishable from the double due to differing results from optimisation; this is highlighted by their differing timings after optimisation in Table 4. Execution concludes at time t_3.

In short, the RVM adaptive optimisation system has introduced a side-channel vulnerability where there was none in the original code. Although the window of vulnerability may be small and may require some effort to mount a realistic attack against, information about bits of the secret value d can clearly be leaked during this period: one simply mounts a SPA type power analysis attack as described above and reads the bits of d directly from the operation trace during the vulnerable period.

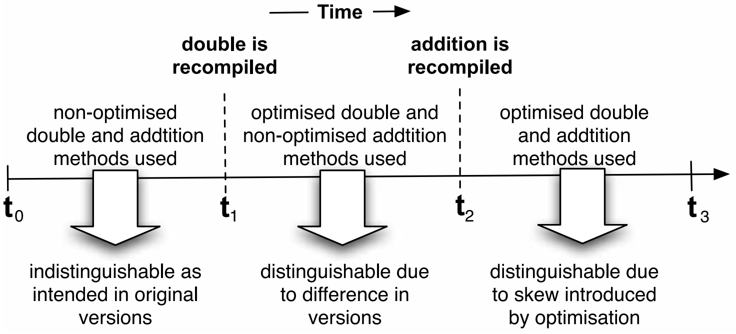
Time
double is recompiled
addition is recompiled
non-optimised double and addtition methods used
optimised double and non-optimised addtition methods used
optimised double and addtition methods used
t_0
t_1
t_2
t_3
indistinguishable as intended in original versions
distinguishable due to difference in versions
distinguishable due to skew introduced by optimisation

Fig. 4. A timeline of execution within three specific periods.

6. Conclusions

Thanks to a wealth of research and associated literature, implementation of ECC has been demystified to the extent that it 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 based system are far from trivial.

We investigated the use of language and compilation techniques to assist the programmer to solve this problem. We introduced the CAO language and associated compiler as a means of naturally describing cryptographically interesting programs. Such programs can be analysed by the compiler and undergo domain specific, cryptography-aware analysis, transformation and optimisation phases. Counter arguments to the use of such techniques are common. For example, one might posit that expert programmer will always produce more optimal programs, or reason that legal issues surrounding patent violation make use of automatic tools difficult. However, this remains an interesting research area; our ongoing goal is that the knowledge and experience of expert practitioners be partially transfered into mechanised tools to improve both productivity, maintainability, portability and overall software quality.

A significant problem remains in that by using such tools to assist the act of engineering software, one needs to trust them from a security perspective. We presented a result that shows naïve dynamic compilation allows execution of different code than specified by the programmer, and how can counteract side-channel resistant implementations. Without a clearer picture of security models for side-channel attack [35], it is not clear how the dual goals of performance and security can be balanced in this context; this remains an open research question.

Acknowledgements

The authors would like to thank various anonymous referees for their helpful comments, and 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, P. Rohatgi, The EM Side-Channel(s). In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2523, 29–45, 2002.

[2] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, S.F. Hummel, D. Lieber, V. Litvinov, M.F. Mergen, T. Ngo, J.R. Russell, V. Sarkar, M.J. Serrano, J.C. Shepherd, S.E. Smith, V.C. Sreedhar, H. Srinivasan, J. Whaley, The Jalapeño Virtual Machine. In *IBM System Journal*, **39**(1), 2000.

[3] ARM Limited. *Jazelle White Paper*. Available from: http://www.arm.com/documentation/.

[4] ARM Limited. *ARM946E-S Technical Reference Manual*. Available from: http://www.arm.com/documentation/.

[5] M. Arnold, S.J. Fink, D. Grove, M. Hind, P.F. Sweeney, Adaptive Optimization in the Jalapeño JVM. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2000.

[6] R.M. Avanzi, Aspects of Hyperelliptic Curves over Large Prime Fields in Software Implementations. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 3156, 148–162, 2004.

[7] P.D. Barrett, Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology (CRYPTO)*, LNCS 263, 311–323, 1986.

[8] M. Barbosa, D. Page, On the Automatic Construction of Indistinguishable Operations. In *Cryptology ePrint Archive*, Report 2005/174, 2005.

[9] I.F. Blake, G. Seroussi, N.P. Smart, *Elliptic Curves in Cryptography*. Cambridge University Press, Cambridge, 1999.

[10] I.F. Blake, G. Seroussi, N.P. Smart, *Advances in Elliptic Curve Cryptography*. Cambridge University Press, Cambridge, 2004.

[11] D. Boneh, D. Brumley, Remote Timing Attacks Are Practical. Available from: http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf.

[12] D.J. Bernstein, Cache-timing Attacks on AES. Available from: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[13] É. Brier, M. Joye, Weierstraß Elliptic Curves and Side-channel Attacks. In *Public Key Cryptography (PKC)*, LNCS 2274, 335–345, 2002.

[14] J. Camenisch, M. Rohe, A.-R. Sadeghi, Sokrates – A Compiler Framework for Zero-Knowledge Protocols. In *Western European Workshop on Research in Cryptology (WEWoRC)*, 2005.

[15] B. Chevallier-Mames, M. Ciet, M. Joye, Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. In *IEEE Transactions on Computers*, **53**(6), 760–768, 2004.

[16] J.-S. Coron, Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 1717, 292–302, 1999.

[17] Computational Algebra Group, University of Sydney. *Magma Computational Algebra System*. Available from: http://magma.maths.usyd.edu.au/magma/.

[18] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, J. Noyá, Tempo: Specializing Systems Applications and Beyond. In *ACM Computing Surveys*, **30** (3), 1998.

[19] P. Crescenzi, V. Kann, A Compendium of NP Optimization Problems. Available from: http://www.nada.kth.se/~viggo/problemlist/.

[20] G. Dueck, T. Scheuer, Threshold Accepting: A General Purpose Optimization Algorithm Appearing Superior to Simulated Annealing. In *Journal of Computational Physics*, **90**(1), 161–175, 1990.

[21] P. Gaudry, E. Thomé, The $\mathtt{mp}\mathbb{F}_q$ Library and Implementing Curve-based Key Exchanges. In *Software Performance Enhancement for Encryption and Decryption (SPEED)*, 49–64, 2007.

[22] D. Gupta, B. Malloy, A. McRae, The Complexity of Scheduling for Data Cache Optimization. In *Information Sciences*, **100** (1–4), 1997.

[23] D. Hankerson, A. Menezes, S. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer-Verlag, New York, 2004.

[24] J.L. Hennessy, D.A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Los Altos, 2006.

[25] M. Joye, J.-J. Quisquater, Hessian Elliptic Curves and Side-Channel Attacks. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2162, 402–410, 2001.

[26] D. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, 1999.

[27] N. Koblitz, Elliptic Curve Cryptosystems. In *Mathematics of Computation*, **48**, 203–209, 1987.

[28] N. Koblitz, Hyperelliptic Cryptosystems. *Journal of Cryptology*, **1**(3), 139–150, 1989.

[29] P.C. Kocher, Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology (CRYPTO)*, LNCS 1109, 104–113, 1996.

[30] P.C. Kocher, J. Jaffe, B. Jun, Differential Power Analysis. In *Advances in Cryptology (CRYPTO)*, LNCS 1666, 388–397, 1999.

[31] M. Kowarschik, C. Wei, An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies*, LNCS 2625, 213–232, 2003.

[32] J.R. Lewis, B. Martin, Cryptol: High Assurance, Retargetable Crypto Development and Validation. In *Military Communications Conference*, **2**, 820–825, 2003.

[33] P.-Y. Liardet, N.P. Smart, Preventing SPA/DPA in ECC Systems Using the Jacobi Form. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2162, 391–401, 2001.

[34] S. Lucks, N. Schmoigl, E.I. Tatli, The Idea and the Architecture of a Cryptographic Compiler. In *Western European Workshop on Research in Cryptology (WEWoRC)*, 2005.

[35] S. Micali, L. Reyzin, Physically Observable Cryptography (Extended Abstract). In *Theory of Cryptography*, LNCS 2951, 278–296, 2004.

[36] V. Miller, Uses of Elliptic Curves in Cryptography. In *Advances in Cryptology (CRYPTO)*, LNCS 218, 417–426, 1985.

[37] P.L. Montgomery, Modular Multiplication Without Trial Division. *Mathematics of Computation*, **44**, 519–521, 1985.

[38] S.S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, Los Altos, 1997.

[39] J.D. Nielsen, M.I. Schwartzbach, A Domain-Specific Programming Language for Secure Multiparty Computation. In *Programming Languages and Analysis for Security (PLAS)*, 2007.

[40] D. Page, *CAO : A Cryptography Aware Language and Compiler*. Available from: http://www.cs.bris.ac.uk/home/page/research/cao.html.

[41] J. Sermulins, W. Thies, R. Rabbah, S. Amarasinghe, Cache Aware Optimization of Stream Programs. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

[42] Standards for Efficient Cryptography Group (SECG). *SEC 2: Recommended Elliptic Curve Domain Parameters*, 2000. Available from: http://www.secg.org.

[43] V. Shoup, *NTL: A Library for doing Number Theory*. Available from: http://www.shoup.net/ntl/.

[44] J.A. Solinas, Generalized Mersenne Numbers. *Technical Report CORR 99-39*, University of Waterloo, 1999.

[45] E. Trichina, A. Bellezza, Implementation of Elliptic Curve Cryptography with Built-In Counter Measures against Side Channel Attacks. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2523, 98–113, 2002.

[46] C.D. Walter, Montgomery Exponentiation Needs No Final Subtractions. *Electronics Letters*, **35**, 1831–1832, 1999.