

Compilers in (Elliptic Curve) Cryptography

Dan Page

ECC 2006

- ▶ Most people probably use something like C for their implementation tasks.
- ▶ There are already some “security-aware” compilers one can use to construct better software:
 - ▶ C compilers can instrument buffer-overflow detection and prevention systems.
- ▶ But “cryptography-aware” compilers are becoming an increasingly attractive option.
 - ▶ C makes it difficult to write cryptographic programs in a natural, maintainable form:
 - ▶ Core types and operations are not first-class.
 - ▶ Often need heterogeneous languages for writing primitives and protocols.
 - ▶ C compilers can't help with traditional optimisations since it doesn't know anything about the types or operations involved.
 - ▶ C compilers can't help detect cryptographic security problems because the semantics have disappeared during translation into code.

▶ SIMAP

- ▶ <http://www.daimi.au.dk/~fagidiot/simap/>
- ▶ C-like syntax.
- ▶ Focused on secure multi-party computation protocols.
- ▶ Compiler creates executable from program automatically.
- ▶ Executable includes calls to appropriate cryptographic sub-protocols and communication.

▶ LaCodA

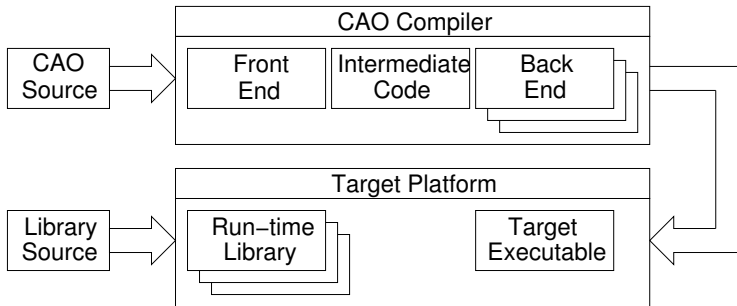
- ▶ <http://schmoigl-online.de/lacoda/lacoda/>
- ▶ Java-like syntax.
- ▶ No specific focus, aims to be general purpose.
- ▶ Compiler creates executable from program automatically.
- ▶ Aims at allowing protocol analysis and reasonable performance in resulting executable.

▶ Variety of older systems CASPER, CVS ...

- ▶ Mainly based on writing protocols in some process calculus.

- ▶ CAO stands for C-and-occam, the main aims are to:
 - ▶ Promote cryptographically interesting concepts to first-class language features.
 - ▶ Make sure the compiler and run-time system can support these features efficiently.
- ▶ Introduce a rich set of types.
 - ▶ For example big-integers, finite fields, rings ...
- ▶ Introduce a rich set of operations.
 - ▶ For example maps, randomness, exponentiation, rotation ...
- ▶ Allow symbols to be tagged with attributes.
 - ▶ This permits mark-up of program with appropriate semantic information.
- ▶ Promote single-source descriptions:
 - ▶ Allow description of parallelism ...
 - ▶ Allow multiple target architectures and libraries ...
- ▶ CAO also aims to address performance and analysis of physical-security vulnerabilities in more detail.

- ▶ The overall architecture uses compilation techniques in two directions:
 - ▶ From CAO source code towards executable.
 - ▶ From library source code towards run-time.



- ▶ But this is meant to relate to ECC somehow ...
- ▶ So for the sake of a consistent example, I'll focus on curves over $K = \mathbb{F}_{2^n}$ such that

$$E(K) : y^2 + xy = x^3 + ax^2 + b$$

for curve parameters $a, b \in K$.

- ▶ The three main parts of the talk will be:
 1. Experiments on “compiler generated” field arithmetic.
 2. Usage of “compiler assisted” curve arithmetic implementation.
 3. Experiments on “compiler blamed” security problems !

Part 1: Specialisation of Field Arithmetic

- ▶ In a paper about performance issues in HECC, Roberto Avanzi makes a nice statement about 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 some ways this is quite an obvious statement:
 - ▶ Expert programmers routinely optimise and specialise their programs to avoid such overheads.
 - ▶ For example, inline or macro-ise addition which is just XORs.
 - ▶ For example, write a specialised reduction method for a given field.
- ▶ But in terms of software engineering, it is vastly important:
 - ▶ Not all programmers are experts !
 - ▶ Hand optimisation is labour intensive and error prone.
 - ▶ Early optimisation hinders portability.

- ▶ What we'd really like is a best-of-both-worlds situation:
 - ▶ Take a generic field/curve code library.
 - ▶ Automatically transform it into high-performance field/curve code for given parameters.
- ▶ Fortunately this technique is well known, it is called specialisation or partial evaluation.
- ▶ We wrote some \mathbb{F}_{2^n} code which was generic for any field parameters:
 - ▶ Squaring used table based coefficient thinning.
 - ▶ Multiplication used right-to-left comb method.
 - ▶ Reduction used a generic word-wise approach.
- ▶ We then optimised this code by hand:
 - ▶ Macro-ised (that is, forced inlining and unrolling) many functions.
 - ▶ Unrolled loops and propagated constants for each field.
 - ▶ Wrote specialised reduction for each field.

- ▶ For $K = \mathbb{F}_{2^n}$ we write:

```
inline void gf2n_add( int n,          unsigned int r[],
                    const unsigned int x[],
                    const unsigned int y[] )
{
    for( i = 0; i < n; i++ )
        r[ i ] = x[ i ] ^ y[ i ];
}
```

- ▶ Tempo converts this into:

```
inline void gf2n_add( int n,          unsigned int r[],
                    const unsigned int x[],
                    const unsigned int y[] )
{
    r[0] = x[0] ^ y[0];
    r[1] = x[1] ^ y[1];
    r[2] = x[2] ^ y[2];
    r[3] = x[3] ^ y[3];
    r[4] = x[4] ^ y[4];
    r[5] = x[5] ^ y[5];
}
```

► For $K = \mathbb{F}_{2^n}$ we write:

```
inline void gf2n_shl( int n,          unsigned int r[],
                    const unsigned int x[], int d )
{
    unsigned int d0 = 0;
    unsigned int d1 = 0;

    int i;

    for( i = 0; i < n; i++ )
    {
        unsigned int d2 = x[ i ];

        d1 = d2 >> ( 8 * sizeof( unsigned int ) - d );
        d1 = d1 & ( ( 1 << d ) - 1 );

        r[ i ] = ( d2 << d ) | d0;

        d0 = d1;
    }
}
```

► Tempo converts this into:

```
inline void gf2n_shl( int n,          unsigned int *r,
                    const unsigned int *x, int d )
{
    unsigned int d0;
    unsigned int d1;
    unsigned int _Lvector_shl_4_d2;

    d0 = 0u;
    _Lvector_shl_4_d2 = x[0];
    d1 = _Lvector_shl_4_d2 >> 31 & 1;
    r[0] = _Lvector_shl_4_d2 << 1 | d0;
    d0 = d1;
    _Lvector_shl_4_d2 = x[1];
    d1 = _Lvector_shl_4_d2 >> 31 & 1;
    r[1] = _Lvector_shl_4_d2 << 1 | d0;
    d0 = d1;

    ...

    _Lvector_shl_4_d2 = x[5];
    d1 = _Lvector_shl_4_d2 >> 31 & 1;
    r[5] = _Lvector_shl_4_d2 << 1 | d0;
    d0 = d1;
    _Lvector_shl_4_d2 = x[6];
    d1 = _Lvector_shl_4_d2 >> 31 & 1;
    r[6] = _Lvector_shl_4_d2 << 1 | d0;
    d0 = d1;
}
```

- ▶ Comparison of these implementations with automatically optimised code on $K = \mathbb{F}_{2^{163}}$ was encouraging:
 - ▶ We used a 2.8 GHz Pentium 4 processor, GCC 4.0.0 and Tempo 1.202; results in clock cycles.

	Add	Sqr	Mul
Generic	126	944	11180
Hand Specialised	102	596	8211
Auto Specialised	102	548	8130

- ▶ Simply using the field parameters as input, Tempo does as well as we did !
 - ▶ Actually it does a bit better because it bothered to unroll the main loop in the reduction function.
 - ▶ But it can't deal with inline assembly language, we are working on a system to do this ...

Part 2: Compilation of Curve Arithmetic

- ▶ Consider an ideal method for implementing a point doubling function on $E(\mathbb{F}_{2^{163}})$:
 1. Take *Guide to Elliptic Curve Cryptography* or similar from bookshelf.
 2. Look up formula for point doubling on $E(\mathbb{F}_{2^n})$.
 3. Type in formula directly then compile, execute and debug.
- ▶ This is (sort of) possible by writing a CAO program:

```

typedef gf2n := gf[ 2**163 + 2**7 + 2**6 + 2**3 + 1 ];

a : gf2n;
b : gf2n;

dbl( X1 : gf2n, Y1 : gf2n, Z1 : gf2n ) : gf2n, gf2n, gf2n
{
  X3 : gf2n;
  Y3 : gf2n;
  Z3 : gf2n;

  Z3 := X1**2 * Z1**2;
  X3 := X1**4 + b + Z1**4;
  Y3 := b * Z1**4 * Z3 + X3 * ( a * Z3 + Y1**2 + b * Z1**4 );

  return X3, Y3, Z3;
}

```

- ▶ Since the field type is first-class in the language, the compiler can apply standard optimisations.
- ▶ Strength reduction changes multiplication/powering by small constants into an addition chain.
 - ▶ $x1^{**4}$ is computed as $(x1^{**2})^{**2}$.
- ▶ Common sub-expression elimination shares results and avoids re-computation.
 - ▶ $x1^{**4}$ is computed using previously computed $x1^{**2}$.
- ▶ Register allocation means we reduce the footprint in memory.
 - ▶ $z1^{**2}$ and $z1^{**4}$ can share allocated space.
- ▶ A simple form of range analysis can spot where to place delayed reductions.
 - ▶ This isn't really useful here but could improve an implementation over \mathbb{F}_p .

- ▶ Late breaking (i.e. half baked) idea; take two contrived (i.e. semantically meaningless) functions with calls to ZZ_p:

```
void f()
{
    add( t0, t1, t2 );
    add( t0, t1, t2 );
    add( t0, t1, t2 );
    add( t0, t1, t2 );
    mul( t3, t4, t5 );
    mul( t3, t4, t5 );
    mul( t3, t4, t5 );
    mul( t3, t4, t5 );
    mul( t3, t4, t5 );
    sub( t6, t7, t8 );
    sub( t6, t7, t8 );
    sub( t6, t7, t8 );
    sub( t6, t7, t8 );
    sqr( t9, t10 );
    sqr( t9, t10 );
    sqr( t9, t10 );
    sqr( t9, t10 );
}

void g()
{
    add( t0, t1, t2 );
    mul( t3, t4, t5 );
    sub( t6, t7, t8 );
    sqr( t9, t10 );
    add( t0, t1, t2 );
    mul( t3, t4, t5 );
    sub( t6, t7, t8 );
    sqr( t9, t10 );
    add( t0, t1, t2 );
    mul( t3, t4, t5 );
    sub( t6, t7, t8 );
    sqr( t9, t10 );
    add( t0, t1, t2 );
    mul( t3, t4, t5 );
    sub( t6, t7, t8 );
    sqr( t9, t10 );
}
```

- ▶ Function f is **consistently** faster than function g .
 - ▶ About 1000-3000 cycles, or about 1-3% ... not a lot !
 - ▶ Difference probably down to cache behaviour.
- ▶ ECC code often has similar(ish) sequences to this.
- ▶ Optimisation is just heuristically guided instruction scheduling.

- ▶ Anyway ... at the moment CAO generated code isn't pretty !
- ▶ But in terms of performance and functionality it isn't too far off what one would write by hand:

```

void dbl( gf2n& __ident_anon0,
          gf2n& __ident_anon1,
          gf2n& __ident_anon2,
          gf2n& X1,
          gf2n& Y1,
          gf2n& Z1 )
{
    gf2n __t50;
    gf2n __t52;
    gf2n __t51;
    gf2n __t53;
    sqr( __t50, X1 );
    sqr( __t52, Z1 );
    mul( __ident_anon2, __t50, __t52 );
    sqr( __t50, __t50 );
    add( __t51, __t50, b );
    sqr( __t50, __t52 );
    add( __ident_anon0, __t51, __t50 );
    mul( __t53, b, __t50 );
    mul( __t52, __t53, __ident_anon2 );
    mul( __t51, a, __ident_anon2 );
    sqr( __t50, Y1 );
    add( __t50, __t51, __t50 );
    add( __t50, __t50, __t53 );
    mul( __t50, __ident_anon0, __t50 );
    add( __ident_anon1, __t52, __t50 );
}

```

- ▶ One can start to implement other parts of the system in CAO as well.
- ▶ The attribute mechanism offers a nice way to detail semantics of the program:

```
mul( Px : gf2n,  
     Py : gf2n, d : int : { secret } ) : gf2n, gf2n  
{  
  Qx : gf2n := 0;  
  Qy : gf2n := 1;  
  Qz : gf2n := 0;  
  
  i : int;  
  
  for( i := sizeof( d ) - 1; i >= 0; i := i - 1 )  
  {  
    ( Qx,Qy,Qz ) := dbl( Qx,Qy,Qz );  
  
    if( d[i] == 1 )  
      ( Qx,Qy,Qz ) := add( Qx,Qy,Qz, Px,Py );  
  }  
  
  Qx := Qx / Qz**2;  
  Qy := Qy / Qz**3;  
  
  return Qx, Qy;  
}
```

- ▶ But this implementation of point multiplication is insecure against SPA attacks:
 - ▶ Profile execution and spot where add and double operations occur.
 - ▶ During iteration i , if an addition occurs we know that $d_i = 1$.
- ▶ The compiler knows d is tagged with the `secret` attribute so it can do some simple analysis:
 - ▶ Unbalanced control flow can be automatically located using the program-counter model of Molnar et al.
 - ▶ In this case, the unbalanced branch depends on d which is secret.
- ▶ Therefore, we can easily spot this as a vulnerability and issue a warning to the programmer:

```
warning [ecc.cao:42:28-34] : branch based on secret condition
  for( i := sizeof( d ) - 1; i >= 0; i := i - 1 )
    ^
warning [ecc.cao:46:7-16] : branch based on secret condition
  if( d[i] == 1 )
    ^
```

Part 3: Dynamic Compilation Versus Security

- ▶ This all sounds great ... but how can you trust your compiler ?
 - ▶ On one hand, compiler assistance can obviously help us out.
 - ▶ On the other, it might do something unexpected and/or unpleasant.
- ▶ Dennis Ritchie describes a “trojan-horse C compiler” for example.
- ▶ David Naccache gave a more modern illustration of this at Eurocrypt 2006:
 - ▶ Story described an encrypted on-chip interconnect or bus.
 - ▶ Turned out the synthesis engine (a compiler) had removed the encryption.
 - ▶ The optimiser realised from end-to-end, the logic was doing nothing !

- ▶ Dynamic compilation is vital for performance of interpreted languages like Java:
 - ▶ The virtual machine monitors your programs being run.
 - ▶ It locates regions that are executed often, i.e. the hot-spots.
 - ▶ The hot-spots are aggressively re-compiled.
 - ▶ Specialised fast versions of the hot-spots replace the slower versions.
 - ▶ The program runs faster because the most often used code is highly optimised.
- ▶ But how does this effect the security of the program ?
 - ▶ Effectively the program being executed is no longer the one written by the programmer.
 - ▶ This doesn't sound like a great idea ...

- ▶ To guard against side-channel attack, one can construct so-called indistinguishable point arithmetic on $E(\mathbb{F}_{2^n})$:

Doubling		Addition Step 1		Addition Step 2	
λ_1	$\leftarrow Z_P^2$	λ_1	$\leftarrow Z_P^2$	λ_{11}	$\leftarrow Z_R^2$
λ_3	$\leftarrow Y_P \cdot Z_P$	λ_2	$\leftarrow \lambda_1 \cdot X_Q$	λ_{12}	$\leftarrow Z_R \cdot Y_Q$
Z_R	$\leftarrow X_P \cdot \lambda_1$	λ_3	$\leftarrow \lambda_1 \cdot Z_P$	λ_{13}	$\leftarrow \lambda_8 \cdot \lambda_{10}$
λ_4	$\leftarrow X_P^2$	λ_{\perp}	$\leftarrow \lambda_{\perp}^2$	λ_{14}	$\leftarrow \lambda_7^2$
λ_5	$\leftarrow Z_R + \lambda_4$	λ_{\perp}	$\leftarrow \lambda_{\perp} + \lambda_{\perp}$	λ_{15}	$\leftarrow \lambda_{11} + \lambda_{13}$
λ_6	$\leftarrow C \cdot \lambda_1$	λ_6	$\leftarrow Y_Q \cdot \lambda_3$	λ_{16}	$\leftarrow \lambda_7 \cdot \lambda_{14}$
λ_7	$\leftarrow X_P + \lambda_6$	λ_7	$\leftarrow X_P + \lambda_2$	X_R	$\leftarrow \lambda_{15} + \lambda_{16}$
λ_8	$\leftarrow \lambda_4^2$	λ_{\perp}	$\leftarrow \lambda_{\perp}^2$	λ_{\perp}	$\leftarrow \lambda_{\perp}^2$
λ_9	$\leftarrow \lambda_8 \cdot Z_R$	Z_R	$\leftarrow Z_P \cdot \lambda_7$	λ_{17}	$\leftarrow X_R \cdot \lambda_{10}$
λ_{10}	$\leftarrow \lambda_5 + \lambda_3$	λ_8	$\leftarrow Y_P + \lambda_6$	λ_{18}	$\leftarrow \lambda_9 + \lambda_{12}$
λ_{11}	$\leftarrow \lambda_7^2$	λ_{\perp}	$\leftarrow \lambda_1^2$	λ_{\perp}	$\leftarrow \lambda_{\perp}^2$
X_R	$\leftarrow \lambda_{11}^2$	λ_{\perp}	$\leftarrow \lambda_{\perp}^2$	λ_{\perp}	$\leftarrow \lambda_{\perp}^2$
λ_{12}	$\leftarrow \lambda_{10} \cdot X_R$	λ_9	$\leftarrow \lambda_8 \cdot X_Q$	λ_{19}	$\leftarrow \lambda_{18} \cdot \lambda_{11}$
Y_R	$\leftarrow \lambda_9 + \lambda_{12}$	λ_{10}	$\leftarrow Z_R + \lambda_8$	Y_R	$\leftarrow \lambda_{17} + \lambda_{19}$

- ▶ Actually the compiler can do this automatically as well, but that is another story !

- ▶ The idea is that from an SPA attack on a double-and-add style exponentiation, instead of getting a trace such as

DDDA

where *D* a doubling indicates and *A* an addition, the attacker gets

XXXXX

for the atomic sequence *X* which could be a double or either of the addition steps.

- ▶ Look at the timings of the indistinguishable arithmetic with and without dynamic compilation:
 - ▶ We used a 2.8 GHz Pentium 4 processor with Jikes Research Virtual Machine (RVM); results in milli-seconds.

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

Conclusions

- ▶ This isn't about making a perfect compiler that can beat/replace an expert programmer.
- ▶ That is, we can probably tolerate a small %'age "worseness ratio" if it improves other things.
 - ▶ ... mainly mundane things like productivity, maintainability, portability and so on.
 - ▶ or maybe we are just interested in rapid prototyping before manually finishing the job.
- ▶ It is about transferring the knowledge of expert programmers into automated tools to help everyone else.
- ▶ As software and cryptography become more complex, it isn't realistic for engineers to keep track of and implement academic results.

▶ Some larger challenges ...

1. Development of a unified language definition:

- ▶ There are several good projects getting off the ground, all with different goals.
- ▶ Would be ideal to have one language/compiler core so outcomes can be shared.

2. The “compiler optimisation ordering problem” is harder.

- ▶ Some of the typical assumptions are wrong as well ...

3. Development of a (good) model for side-channel security:

- ▶ Automated security proof is a challenging goal but some progress has been made.
- ▶ Side-channel security is harder because we don't have a (good) model of security !

4. Proofs of compiler correctness from security perspective:

- ▶ In terms of functional correctness we can hope to prove compiler correctness.
- ▶ In terms of security things are a bit more tricky ...

Blatant Advert

<http://www.cs.bris.ac.uk/~page/research/cao.html>

it's only a crap prototype at the moment but we **really** want to make it better ...

Questions ?