Computer Aided Cryptographic Engineering

Dan Page

SPEED, June 2007

(joint work with Andrew Moss, Manuel Barbosa, Nigel Smart ...)

(an apology: some of you will have heard me give a very similar talk before, I won't be offended if you walk out or go to sleep)

- In hardware implementation, so-called Electronic Design Automation (EDA) tools offer:
  - Improved quality (performance, area, power consumption).
  - Improved correctness (verification, analysis).
  - Improved productivity (synthesis, floorplanning, place and route).
- In software implementation, tool-chains (assemblers, compilers, debuggers) offer similar benefits.

- But in both cases, being domain-unspecific (i.e. general purpose) is a problem:
  - If a compiler doesn't know about a type or operation one uses, it can't perform analysis, optimisation or transformation.
  - If a compiler doesn't know about the semantics of an error, it can't detect or patch the cause.
- Roughly speaking, one doesn't get the benefits normally associated with "high-level development".

- In cryptographic software, this is particularly problematic:
  - Often ported to diverse platforms (smart card versus an e-commerce server).
  - Often resource sensitive (computation, storage).
  - Often use highly specialist techniques (ZK-proofs !?).
  - Always security sensitive (side-channel behaviour).
- So it makes sense to deploy more domain-specific tools.
- You've probably all got examples already that perform somewhat niche tasks.
  - A tool to generate GMP internals ?
  - A tool to optimise bit-sliced implementations ?
  - A tool to generate specific modular reduction functions ?
  - A tool to generate normal basis multipliers ?
- However these are are often too specific, not very robust (in my case at least !) or not very well integrated with each other.

- A potential answer is a toolbox rather than a tool:

  "Computer Aided Cryptographic Engineering"
  equals
  "EDA for cryptography"

- Important note: sometimes it's assumed we want tools which will "beat human programmers" ...
- ... but often this isn't the most important goal; for example
  - reduction in programmer effort (improvement in prototyping speed),
  - improvement in maintainability/portability,
  - and transfer of knowledge

  can be equally desirable in a given context.
- That is, we can probably tolerate a small "worseness ratio" if it produces improvements elsewhere.

- Some example “low-level” tools:
- `qhasm`
  - `http://cr.yp.to/qhasm.html`
  - Acts as a means of “high level” assembly language programming.
  - Includes excellent support for floating-point and vector operations.
  - Supports optimisation of register allocation, scheduling etc.
  - Has allowed many novel, high-performance cryptographic implementation tasks (Salsa20, Poly1305 etc.).
- `C--`
  - `http://www.cminusminus.org/`
  - Acts as (portable) interface between compiler back-end and optimising code generators.
  - Type-system is based on “machine level” units (i.e. machine words) rather than their content.
  - Supports optimisation of register allocation, function calls etc.

- Some example “mid-level” tools:
- CAO
  - `http://www.cs.bris.ac.uk/~page/research/cao.html`
  - Focused on description of ECC-based library components.
  - Able to perform domain-specific optimisations on curve arithmetic.
- LaCodA
  - `http://schmoigl-online.de/lacoda/lacoda/`
  - Java-like syntax, aims to be general purpose tool.
  - Compiler creates executable from program automatically.
  - Aims at allowing protocol analysis and reasonable performance in resulting executable.

- Some example “high-level” tools:
- SIMAP
  - `http://www.daimi.au.dk/~fagidiot/simap/`
  - Focused on secure multi-party computation protocols.
  - Supports notions of secure communication, secret sharing etc.
  - Compiled executable calls appropriate cryptographic sub-protocols and communication.
- Sokrates
  - `http://www.prosec.rub.de/sokrates.html`
  - Collaboration between Universities of Bern and Bochum, and IBM.
  - Basic idea is to describe a high level ZK protocol in a natural way.
  - Compiler produces executable prover and verifier programs.

Part 1: Implementation of Curve Arithmetic

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

```
typedef gfp := gf[ 2**192 - 2**64 - 1 ];

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;
}
```

- 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.
  - `y1**4` is computed as `(y1**2)**2`.
- Common sub-expression elimination shares results and avoids re-computation.
  - `y1**4` is computed using previously computed `y1**2`.
- Register allocation means we reduce the footprint in memory.
  - `y1**4` and `l3` can share allocated space.

- 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( 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 );
}
```

- What we've bought ourselves is the ability for a programmer to focus on the high-level algorithm rather than the low-level implementation.

- This is useful but maybe not scientifically very interesting !
- However, we've been investigating more domain-specific transformations at the curve arithmetic level ...
  - Improving performance through cache-conscious scheduling.
  - Improving performance through automatic delayed reduction.
  - Improving security through automatic function indistinguishability.
- ... some of the improvements are marginal, but they are free in terms of programmer effort !
- The more interesting blue-sky research goal is "automatic security checking".
  - An example is side-channel leakage from the implementation.
  - We've taken some (very) basic steps in this direction (e.g. use of the program counter model by David Molnar et al.) ...
  - ... but is seems quite a difficult area, partly because there isn't really a good security model !

Part 2: Implementation of Field Arithmetic

- But what about the field arithmetic level ?
- In a paper about performance issues in HECC, Roberto Avanzi makes an interesting statement about general purpose 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, write a specialised reduction 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 library.
  - Automatically transform it into high-performance code for given parameters.
- Fortunately this technique is well known, it is called specialisation or partial evaluation.
- There are (roughly) two ways one might approach this problem:
  1. Write a special purpose specialiser (or code generator).
     - We already saw this approach in the talk on $\texttt{mp}\mathbb{F}_q$.
     - Basically an exercise in meta-programming, i.e. write a program that writes other programs.
     - Potentially based on expandable templates or patterns.
  2. Use a general purpose specialiser ...

- Tempo is an example of a general purpose C specialiser.
  - Input is a C function and an execution context.
    - Some variables are static (don't change between calls).
    - Some variables are dynamic (do change between calls).
  - Static variables are replaced by constant values.
  - Aggressive loop unrolling, control flow pruning etc. is performed.
  - Output is a specialised C function valid only when matching execution context exists.
- Some "in progress" work we've been looking at represents cryptographic algorithms somewhat symbolically:
  - Say you are doing some arithmetic with multi-digit integers.
  - Now imagine some or all of the operands are fixed, one can "partially evaluate" the algorithm ... which is sort of what Tempo already does.
  - However, by asking the right domain-specific questions, e.g. "will this operation ever produce a carry", more interesting specialisations are possible than in Tempo.

- Consider a very basic example for addition in $\mathbb{F}_{2^n}$:

```
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 ];
}
```

- For a given field the value n is static so setting $n = 8$ prompts Tempo to give us:

```
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];
  r[6] = x[6] ^ y[6];
  r[7] = x[7] ^ y[7];
}
```

- ... hopefully the main point is clear: doing this by hand probably isn't a great plan any more !
- Plus, with the right infrastructure, specialisation can be done at compile-time or run-time.
  - Issues like instruction caches can make this a bit tricky !
  - Although these issues disappear in environments like Java.
- A blue-sky research goal might be "dynamic GMP" ...
  - GMP can already make limited (optimising) choices during execution.
  - But the code is still static and general purpose.
  - Dynamic specialisation bridges the gap somewhat.
- ... essentially if I use $\mathbb{F}_{2^{174}-3}$ enough, I get the advantages of specialised code for $\mathbb{F}_{2^{174}-3}$ even though GMP need not support it directly.
- This seems like a really cool project if you are an able student.

Part 3: Security Issues or “When Compilers Attack”

- This all sounds great ... but how can you trust the compiler or specialiser ?
  - On one hand, automation can provide obviously advantages.
  - On the other, it might do something unexpected and/or unpleasant.
- ... we already heard that GCC can't be relied upon to compile GMP !
- Ken Thompson described a "trojan-horse C compiler".
  - Basic idea is that a C compiler injects back doors into executable.
  - His moral is "you can't trust code that you did not totally create yourself".
- David Naccache warned of "overly aggressive optimisation".
  - Story described an encrypted on-chip interconnect or bus.
  - Turned out the synthesis engine (a compiler) had removed the encryption.

- 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 now the program which is actually being executed isn't the same program which you wrote !
- Also there are new and fiendish problems on the horizon ...
  - What happens if the compiler violates a patent by optimising in a certain way !
  - The compiler wrote the program being executed ... is the human or the computer going to be sued ?!
- ... I've no idea if there is any legal history to this concept (?).

- ... anyway, back to the dynamic compilation issue.
- To guard against side-channel attack, one can construct so-called indistinguishable point arithmetic; on $E(\mathbb{F}_{2^n})$ this might look like:

Doubling			Addition Step 1			Addition Step 2		
λ_1	$\leftarrow$	z_1^2	λ_1	$\leftarrow$	z_1^2	λ_{11}	$\leftarrow$	z_3^2
λ_3	$\leftarrow$	$y_1 \cdot z_1$	λ_2	$\leftarrow$	$\lambda_1 \cdot x_2$	λ_{12}	$\leftarrow$	$z_3 \cdot y_2$
z_3	$\leftarrow$	$x_1 \cdot \lambda_1$	λ_3	$\leftarrow$	$\lambda_1 \cdot z_1$	λ_{13}	$\leftarrow$	$\lambda_8 \cdot \lambda_{10}$
λ_4	$\leftarrow$	x_1^2	$\lambda_\perp$	$\leftarrow$	$\lambda_\perp^2$	λ_{14}	$\leftarrow$	λ_7^2
λ_5	$\leftarrow$	$z_3 + \lambda_4$	$\lambda_\perp$	$\leftarrow$	$\lambda_\perp + \lambda_\perp$	λ_{15}	$\leftarrow$	$\lambda_{11} + \lambda_{13}$
λ_6	$\leftarrow$	$C \cdot \lambda_1$	λ_6	$\leftarrow$	$y_2 \cdot \lambda_3$	λ_{16}	$\leftarrow$	$\lambda_7 \cdot \lambda_{14}$
λ_7	$\leftarrow$	$x_1 + \lambda_6$	λ_7	$\leftarrow$	$x_1 + \lambda_2$	x_3	$\leftarrow$	$\lambda_{15} + \lambda_{16}$
λ_8	$\leftarrow$	λ_4^2	$\lambda_\perp$	$\leftarrow$	$\lambda_\perp^2$	$\lambda_\perp$	$\leftarrow$	$\lambda_\perp^2$
λ_9	$\leftarrow$	$\lambda_8 \cdot z_3$	z_3	$\leftarrow$	$z_1 \cdot \lambda_7$	λ_{17}	$\leftarrow$	$x_3 \cdot \lambda_{10}$
λ_{10}	$\leftarrow$	$\lambda_5 + \lambda_3$	λ_8	$\leftarrow$	$y_1 + \lambda_6$	λ_{18}	$\leftarrow$	$\lambda_9 + \lambda_{12}$
λ_{11}	$\leftarrow$	λ_7^2	$\lambda_\perp$	$\leftarrow$	$\lambda_\perp^2$	$\lambda_\perp$	$\leftarrow$	$\lambda_\perp^2$
x_3	$\leftarrow$	λ_{11}^2	$\lambda_\perp$	$\leftarrow$	$\lambda_\perp^2$	$\lambda_\perp$	$\leftarrow$	$\lambda_\perp^2$
λ_{12}	$\leftarrow$	$\lambda_{10} \cdot x_3$	λ_9	$\leftarrow$	$\lambda_8 \cdot x_2$	λ_{19}	$\leftarrow$	$\lambda_{18} \cdot \lambda_{11}$
y_3	$\leftarrow$	$\lambda_9 + \lambda_{12}$	λ_{10}	$\leftarrow$	$z_3 + \lambda_8$	y_3	$\leftarrow$	$\lambda_{17} + \lambda_{19}$

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

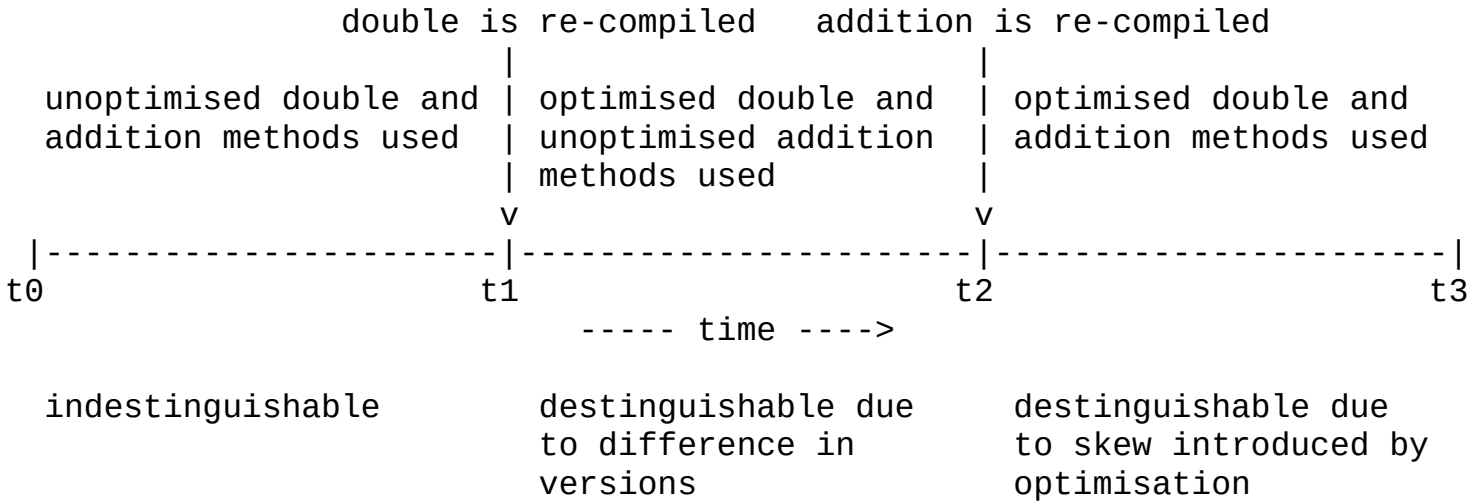
- The idea is that from an SPA attack on a double-and-add style exponentiation, instead of getting a trace such as

  $$DDDA$$

  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.
- But the dynamic compilation system changes our side-channel secure code; we get an execution time-line that looks like:



Part 4: Conclusions

- There are plenty of technical challenges:

1. The optimisation “ordering problem” is tough.
   - Lots of optimisations might work against each other, deciding which to apply when is hard.
   - Other assumptions such as “spilling registers isn’t *so* bad” are no longer valid.
2. Interaction between optimisation and security is difficult.
   - Optimisations for performance and security are often mutually exclusive.
   - Even when this is not exactly true, knowing which choices to make is hard ...
   - ... and even harder when we don’t have a good model for physical security.
3. 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 much more tricky ...

- Also there are some political challenges:

1. Development of some form of unification:
   - There are several good projects getting off the ground, all with different goals.
   - Would be ideal to have some one language/compiler/system so outcomes can be shared.
2. Raising interest and profile in community:
   - Lots of people believe this isn't interesting or worthwhile; publishing papers is hard !
   - Already some interest within ECRYPT, need to widen interest.

Questions ?