

Implementing Provably Correct High-Performance Ciphers with Sketching

Presenter: Armando Solar-Lezama

UC Berkeley

www.cs.berkeley.edu/~asolar

PI: Ras Bodik Collaborators: G. Arnold, V. Saraswat, S. Seshia, L. Tancau



The problem: coding ciphers is hard

Ciphers are designed in terms of well understood cryptographic primitives, such as bit permutations or big-integer arithmetic. But good implementations must make efficient use of the primitives provided by the architecture, from fixed precision arithmetic to SIMD operations. The gap between the two is too large to be bridged efficiently by traditional compilers, so efficient cipher implementations are painfully handwritten at a low level of abstraction, sometimes even with the aid of ad-hoc code generators.

Solution: **synthesize** the implementation from a **sketch**

With sketching, a cipher is first specified cleanly in terms of high level primitives. The programmer then suggests an implementation by sketching it. The sketch is a piece of code containing holes in place of any difficult expressions; thus, low level details such as index expressions or the contents of lookup tables can be left as holes to be filled in by the SKETCH compiler. The resulting code is guaranteed to be a faithful implementation of the original specification.

Sketching Private Key Ciphers

We have used the SKETCH compiler to synthesize an implementation of AES competitive with the one found in OpenSSL. The specification of a round of AES is provided in terms of bit permutations and polynomial multiplications in $GF(2^8)$, but the sketch shown below implements it as a series of table lookups. The SKETCH compiler synthesized some 1024 constants for the table initializers, saving the programmer a huge amount of effort.

Sketch Example: AES Round

```
word[4] AESroundSK
(byte[16] in, byte[16] rkey) implements round{
  word[??] T0 = ??, T1 = ??, T2 = ??, T3 = ??;
  word[4] out;
  out[0] = T0[ in[??] ]^T1[ in[??] ]^
           T2[ in[??] ]^T3[ in[??] ];
  out[1] = T0[ in[??] ]^T1[ in[??] ]^
           T2[ in[??] ]^T3[ in[??] ];
  out[2] = T0[ in[??] ]^T1[ in[??] ]^
           T2[ in[??] ]^T3[ in[??] ];
  out[3] = T0[ in[??] ]^T1[ in[??] ]^
           T2[ in[??] ]^T3[ in[??] ];
  return out;
}
```

Synthesized code replaces all **holes (??)**

Sketch Example: Karatsuba Multiplication

```
bigint<2*N> kar(bigint<N> x, bigint<N> y)
    implements x * y {
  if (N<=1) return x*y;
  bigint<N/2> x0 = x[0:N/2-1], x1=x[N/2:N-1];
  bigint<N/2> y0 = y[0:N/2-1], y1=y[N/2:N-1];

  bigint<N> t00 = kar(x1, y1);
  bigint<N> t01 =
    kar( poly(1,x0,x1,y0,y1), poly(1,x0,x1,y0,y1) );
  bigint<N> t11 = karatsuba(x1, y1);

  return polySparseMul(2, N/2, t00)
    + polySparseMul(2, N/2, t01)
    + polySparseMul(2, N/2, t11);
}
```

Impact

After we release the SKETCH compiler, cipher writers will be able to:

- Easily try different implementation strategies for a cipher
- Write specialized implementations for new architectures with no risk of bugs
- Publish a spec for their cipher knowing that all implementations sketched by third parties will be guaranteed to adhere to it.

Sketching Public Key Ciphers

SKETCH can aid in coding big integer operations used in public key cryptography. The Karatsuba example shows a sketch of the karatsuba multiplication algorithm which multiplies two big integers by recursively applying the following formulas:

$$x = x_1 * b^{N/2} + x_0 \quad y = y_1 * b^{N/2} + y_0$$

$$t_{00} = x_0 * y_0 \quad t_{01} = (x_1 - x_0) * (y_1 - y_0) \quad t_{11} = x_1 * y_1$$

$$x * y = (b^N + b^{N/2}) * t_{11} - b^{N/2} * t_{01} + (b^{N/2} + 1) * t_{00}$$

With sketching, the programmer doesn't need to derive the details of the algorithm; these are derived automatically from the sketch on the left.

The sketch uses polynomial hole **poly** to express that t_{01} is the product of two linear expressions, and the hole **polySparseMul** expresses the product of the t_{xx} by a polynomial of b .