THE CRAMER-SHOUP ENCRYPTION SCHEME

NATHANIEL T. WILLIAMS

CSC290 Final Project
Prof. Chris Brown

1. INTRODUCTION

The Cramer-Shoup cryptosystem was first described in 1998 by Ronald Cramer and Victor Shoup[CS98]. It is essentially an extension of the ElGamal system, with the addition of a hashing function and extra computations, rendering it provably secure versus adaptive chosen ciphertext attacks.

I have written the basic elements of an implementation of Cramer-Shoup in Perl, and will address the specifics of that later. What prompted this project in the first place was the fact that despite Cramer-Shoup having addressed supposed flaws in ElGamal, it has not seen any sort of significant adoptation in the 8 years since its proposal.

2. GOALS AND CONCERNS

First and foremost, I set out to write what I guess you might call a "toy" implementation of Cramer-Shoup. To put it mildly, I did not expect to write anything even remotely usable. Rather, I just wished to understand the practical issues of the scheme, and how we go about handling them.

I also had the idea that I would possibly be able to expound on some of the theory behind Cramer-Shoup's improvements over ElGamal. However, to do this in any adequate sense proved to be somewhat more than I was capable of. Proofs of security based on statistical simulations and abstract probabilistic spaces shall continue to vex me for at least a little while longer.

However, even without having to get too deep into any of the proofs, there was one thing that seemed apparent. I suppose it should be, as it is summed up in this theorem from [CS98], page 9:

> **Theorem 1** *The above cryptosystem is secure against adaptive chosen ciphertext attack assuming that (1) the hash function H is chosen from a universal one-way family and (2) the Diffie-Hellman decision problem is hard in the group G*

Date: December 19, 2006.

This may be a slightly too cynical interpretation, but it seems to me that Cramer-Shoup is just ElGamal with hashing tacked on to it. Granted, its done in a nice way and all, but I can understand why almost every description of it uses the phrase "extension of ElGamal".

But essentially, all of this provable niceness and security depends entirely on the strength of your hashing function. Given the necessity of hashing onto an arbitrary group, I can see how this might be something worth being concerned over. They do, however, suggest in [RS03] that SHA-1 might be sufficient to satisfy their theorem, "in practice, one might simply use SHA-1 directly, without a key — it is not unreasonable to assume that this already satisfies our definition of target collision resistance".

Another issue that much of the literature glosses over is your choice of the group G. For instance, in the above theorem, it is all too easy to take the second part for granted and not worry about Diffie-Hellman. But just as part one depends on the hash function, part 2 depends on your choice of group, something I found to be much more confusing than expected.

Section 4.2 in [RS03] discusses schemes for finding suitable groups over which elements can be easily computed, and elements of $\mathbb{Z}_p^*$ can be encoded. In particular, they mention a scheme involving Sophie Germain primes, that is, subgroups of $\mathbb{Z}_p^*$ of order q, such that $p = 2q + 1$. However, when I actually sat down to write the code for finding these primes, things didn't work so well.

You may notice that as it is now, the primes that I am generating in my implementation are only 128 bits. The reason for this, I think, is that one of the math or arbitrary precision packages that I am using is doing something for which the time required goes up very steeply. I initially tried to first find Sophie Germain primes and construct the group that way, but I found that what I thought was a very simple algorithm was taking non-trivial lengths of time for numbers with only 4 or 5 digits (around 12-18 bits). Past around 128 bits, any operation past simple arithmatic or modding became seemingly impossible. For finding the generators g_1 and g_2, I did occasionally have slightly better luck when I merely incremented p instead of generating a random new one, but it was still unusable. I honestly don't know if it was a problem with the packages, or the way I was going about using them. But, I do not think that only using a 128 bit prime with affect any of this project's deep educational value.

3. IMPLEMENTATION

The program itself consists of three subroutines: *keygen*, *encrypt*, and *decrypt*. The first generates suitable random numbers and computes the corresponding

group elements as described. All the elements for the public and private key are returned and the public key, along with a message is passed to the *encrypt*.

It is worth noting that the ciphertext is about four times as large as the corresponding plaintext. Large ciphertext was the complaint I saw the most often about ElGamal, and this is twice again as big as that. However, I don't think this is really a huge factor, as any "serious" implementation would be a hybrid one, where the asymmetric system is just used to encrypt the key for a symmetric one. DES keys are not that large, so the increased size would probably only be an issue in pathological cases like embedded systems on smart cards or the like.

The step that I thought was the strangest was at the end, when it says to test whether $u_1^{x_1+y_1\alpha} u_2^{x_2+y_2\alpha} = v$. If they are not equal, then the decryption wouldn't work, as in it would spit out gibberish. The fact that they are not equal does not tell us anything that we wouldn't be able to infer from a nonsense output.

4. SOURCE CODE

```
#!/usr/bin/perl

#  an investigation of the Cramer-Shoup encryption scheme
#  by Nat Williams
#  CSC290A

use Crypt::Random qw( makerandom_itv );
use Math::BigInt lib => 'GMP';
use Crypt::Primes qw(maurer);
use Bit::Vector;
use Digest::SHA1 qw(sha1_hex);  #160 bit hash
use Math::Prime::XS qw( is_prime );
use Crypt::Random qw( makerandom_itv );
use Math::Pari qw( :DEFAULT :number :conversions );


#               0  1 2 3 4 5
#  PublicKey = (g1,g2,c,d,h,p)  (I'm leaving the hash out, assume sha1)
#
#  PrivateKey = (x1,x2,y1,y2,z,p)
#

# annoyingly, perl flattens out arrays in subroutine input and output
($pub[0], $pub[1], $pub[2], $pub[3], $pub[4], $pub[5], $priv[0], $priv[1], $priv[2],
    $priv[3], $priv[4],$priv[5]) = keygen();

#chomp($pt=<STDIN>);
$pt = "444444444444444444444444444444";

print "\nPublic Key:\n";
print "g1 = ",$pub[0],"\n";
print "g2 = ",$pub[1],"\n";
```

```
print "c = ",$pub[2],"\n";
print "d = ",$pub[3],"\n";
print "h = ",$pub[4],"\n";
print "p = ",$pub[5],"\n";
print "\nPrivate Key:\n";
print "x1 = ",$priv[0],"\n";
print "x2 = ",$priv[1],"\n";
print "y1 = ",$priv[2],"\n";
print "y2 = ",$priv[3],"\n";
print "z = ",$priv[4],"\n";
print "p = ",$priv[5],"\n";
# and p is priv[5]

@ct = &encrypt($pt,@pub);
print "\nCiphertext:\n";
print "u1 = $ct[0]\n";
print "u2 = $ct[1]\n";
print "e = $ct[2]\n";
print "v = $ct[3]\n";


$m = &decrypt(@ct,@priv);

print "\n";
print "original plaintext: $pt\n";
print "D(E(pt)):           $m\n";

sub keygen { #generate a key, oddly enough

 #  any bigger than this seems to freeze the program
 #  I don't know if it is a problem with one of the
 #  packages I'm using, like PARI, or one of my "algorithms",
 #  but go much higher, and the program just sits there using
 #  about 90% CPU
 my $p = Math::BigInt->new(maurer(Size => 128));

 my $g1 = Math::BigInt->new(makerandom_itv(
     Lower => 2,
     Upper => $p-1,
     Uniform => 0));

 #  but who will generate the generators?
 until(znorder(Mod($g1,$p))==($p-1)) {
  $g1 = Math::BigInt->new(makerandom_itv(
      Lower => 2,
      Upper => $p-1,
      Uniform => 0));
 }

 $g2 = Math::BigInt->new(makerandom_itv(
     Lower => 2,
     Upper => $p-1,
     Uniform => 0));
```

```
    until(znorder(Mod($g2,$p))==($p-1)) {
86   $g2 = Math::BigInt->new(makerandom_itv(
         Lower => 2,
         Upper => $p-1,
         Uniform => 0));
    }
91  #  g1 and g2 with order p-1 ^^^



    #  random elements in Z_p
96  my $x1 = Math::BigInt->new(makerandom_itv(
         Lower => 1,
         Upper => $p-1,
         Uniform => 1,
         ));
101
    my $x2 = Math::BigInt->new(makerandom_itv(
         Lower => 1,
         Upper => $p-1,
         Uniform => 1,
106      ));

    my $y1 = Math::BigInt->new(makerandom_itv(
         Lower => 1,
         Upper => $p-1,
111      Uniform => 1,
         ));

    my $y2 = Math::BigInt->new(makerandom_itv(
         Lower => 1,
116      Upper => $p-1,
         Uniform => 1,
         ));

    my $z = Math::BigInt->new(makerandom_itv(
121      Lower => 1,
         Upper => $p-1,
         Uniform => 1,
         ));

126 my $c = $g1->copy()->bmodpow($x1,$p)->bmul($g2->copy()->bmodpow($x2,$p))->bmod($p);
    my $d = $g1->copy()->bmodpow($y1,$p)->bmul($g2->copy()->bmodpow($y2,$p))->bmod($p);
    my $h = $g1->copy()->bmodpow($z,$p);


131 return ($g1,$g2,$c,$d,$h,$p,$x1,$x2,$y1,$y2,$z,$p);

    }

    sub encrypt{
136
    my $m = Math::BigInt->new($_[0]);
```

```
 my $g1 = Math::BigInt->new($_[1]);
 my $g2 = Math::BigInt->new($_[2]);
 my $c = Math::BigInt->new($_[3]);
 my $d = Math::BigInt->new($_[4]);
 my $h = Math::BigInt->new($_[5]);
 my $p = Math::BigInt->new($_[6]);

 my $r = Math::BigInt->new(makerandom_itv(
     Upper => $p-1,
     Lower => 1,
     Uniform => 1,
      ));

 my $u1 = $g1->copy()->bmodpow($r,$p);
 my $u2 = $g2->copy()->bmodpow($r,$p);
 my $e = $h->copy()->bmodpow($r,$p)->bmul($m)->bmod($p);
 #  this isn't my proudest coding moment here...
 #  the sha1 function outputs hex, which then goes into a
 #  Bit::Vector, which outputs decimal, into a BigInt.
 #  kind of gross.
 my $foo = Bit::Vector->new_Hex(160,sha1_hex("$u1$u2$e"));
 my $alpha = Math::BigInt->new($foo->to_Dec())->bmod($p);
 my $v = $c->copy()->bmodpow($r,$p);
  #print "v first stage = $v\n";
  my $foo = $d->copy()->bmodpow($r->copy()->bmul($alpha),$p);
  #print "foo = $foo\n";
  $v->bmul($foo);
  #print "v after mult = $v\n";
  $v->bmod($p);
  #print "v after mod = $v\n";

 return ($u1,$u2,$e,$v);

}

sub decrypt() {

 my $u1 = $_[0];
 my $u2 = $_[1];
 my $e  = $_[2];
 my $v  = $_[3];
 my $x1 = $_[4];
 my $x2 = $_[5];
 my $y1 = $_[6];
 my $y2 = $_[7];
 my $z  = $_[8];
 my $p  = $_[9];

 my $alpha = Math::BigInt->new(Bit::Vector->new_Hex(160,sha1_hex("$u1$u2$e"))->to_Dec
     ())->bmod($p);

 my $ex1 = $alpha->copy()->bmul($y1)+$x1;
 my $ex2 = $alpha->copy()->bmul($y2)+$x2;
```

```
 my $foo = $u1->copy()->bmodpow($ex1,$p)->bmul($u2->copy()->bmodpow($ex2,$p))->bmod(
     $p);
191  if ($foo->bcmp($v) != 0) {
  print "foo doesn't match v, something is wrong.\n";
 }
 else {
  my $m = $e->bmul($u1->copy()->bmodpow($z,$p)->bmodinv($p))->bmod($p);
196  return $m;
 }
}
```

REFERENCES

[BJN00] Dan Boneh, Antoine Joux, and Phong Q. Nguen, *Why textbook elgamal and rsa encryption are insecure*, Advances in Cryptology—ASIACRYPT 2000 (Tatsuaki Okamoto, ed.), LNCS, no. 1976, Springer, 2000, pp. 30–43.

[Bon98] Dan Boneh, *The decision Diffie-Hellman problem*, Algorithmic number theory, LNCS, vol. 1423, Springer, 1998, pp. 48–63.

[BS99] Mihir Bellare and Amit Sahai, *Non-malleable encryption: equivalence between two notions, and an indistinguishability-based characterization*, Advances in cryptology—CRYPTO '99, LNCS, vol. 1666, Springer, 1999, pp. 519–536.

[Buc01] Johannes A. Buchmann, *Introduction to cryptography*, Springer, 2001.

[CP05] Richard Crandall and Carl Pomerance, *Prime numbers: a computational perspective*, second ed., Springer-Verlag, 2005.

[CS98] Ronald Cramer and Victor Shoup, *A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack*, Advaces in Cryptology—Crypto '98, LNCS, vol. 1462, Springer, 1998, pp. 13–25.

[CS02] ———, *Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption*, Advances in cryptology—EUROCRYPT 2002, LNCS, vol. 2332, Springer, 2002, pp. 45–64.

[Gro05] Jens Groth, *Cryptography in subgroups of $\mathbb{Z}_n^*$*, Second Theory of Cryptography Conference, TCC 2005 (Joe Kilian, ed.), LNCS, no. 3378, Springer, 2005, pp. 50–65.

[MSVV05] Consuelo Marfinez, Rainer Steinwandt, María Isabel González Vasco, and Jorge L. Villar, *A new cramer-shoup like methodology for group based provably secure encryption schemes*, Second Theory of Cryptology Conference, TCC 2005 (Joe Kilian, ed.), vol. 3378, Springer, 2005, pp. 495–509.

[RS03] Ronald and Victor Shoup, *Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack*, SIAM Journal on Computing **33** (2003), no. 1, 167–226.

[Sch96] Bruce Schneier, *Applied cryptogaphy*, second ed., John Wiley & Sons, 1996.

[Sho92] Victor Shoup, *Searching for primitive roots in finite fields*, Mathematics of Computation **58** (1992), no. 197, 369–380.

[Sho01] ———, *Using hash functions as a hedge against chosen ciphertext attack*, Advances in Cryptology—Crypto 2001, LNCS, vol. 1807, Springer, 2001, pp. 239–259.

[SJ00] Claus Peter Schnorr and Markus Jakobsson, *Security of signed elgamal encryption*, Advances in Cryptology—ASIACRYPT 2000 (Tatsuaki Okamoto, ed.), LNCS, vol. 1976, Springer, 2000, pp. 73–89.