Fair and Sound Secret Sharing from Homomorphic Time-Lock Puzzles by Jodie Knapp and Elizabeth A. Quaglia IACR Cryptol. ePrint Arch. 2020 p.1078

Gregory Morse
gregory.morse@live.com

Eötvös Loránd Tudományegyetem/University (ELTE), Budapest, Hungary

Cryptography Seminar Fall Semester 2020
December 14, 2020

Summary

This will be a study of how prior research on Homomorphic Time-Lock Puzzles can be incorporated into Secret Sharing schemes. This allows for the guarantee of certain amount of elapsed time to occur over some number of unknown number of sharing rounds determined by the dealer based on well-established security assumptions. A concrete implementation was given using the multiplicatively homomorphic variant, which was further implemented as a protocol simulation suite in Python and tested to get an idea of real-world performance based on the security bit-size and the time hardness parameters.

Introduction

- Secret Sharing (SS), specifically Rational SS (RSS)
  - Multi-party Computation
  - Fairness - change of strategy is not beneficial
  - Soundness - no corruption
- Time-Delay Encryption (TDE)
  - Cryptographic Memory-Bound Function (CMBF) used previously
  - CMBF has high cost on players who have to verify proof-of-work (PoW) of other players' messages
- Homomorphic time-lock puzzles (HTLPs)
- Process of verification of this research in Python

Secret Sharing (SS) scheme

- (t, n) SS scheme has a (presumed honest) dealer D, secret s and a set of $P = \{P_1, P_2, ..., P_n\}$ of n players
- Security parameter λ e.g. 128 bits
- t is the threshold
- No subset of players $t\prime < t$ in P can learn secret s
- Every subset $t\prime \geq t$ is capable of reconstruction s

SS phases

- Two phases: share and reconstruction
- Communication can be done over a broadcast channel
- Sharing involves the dealer generating n shares and distributing one to each player in P
- If this is done by broadcast, the dealer must digitally sign using secure message authentication codes (MACs) and encrypt the shares
- Reconstruction involves two sub-phases: communication and processing
- Communication has each player in P broadcast their secret to all other players
- The processing phase occurs when the threshold is reached and each player reconstructs s from the shares received
- The SS scheme is thereby a tuple of three probabilistic polynomial time (PPT) algorithms: $(Setup, Share, Recon)$

Rational Secret Sharing (RSS)

- Game theory
- Player is rational if they have a preference in the reconstruction phase
- e.g. maximizing their payoff from the output
- Fairness - probability of learning the secret when deviating from the protocol strategy is negligably more than prescribed strategy
- Requires a computationally strict Nash equilibrium
- No benefit from deviating as well as a penalty for deviating
- Fair reconstruction mechanism needed
- Protocol induced side information (checking share) must still yield a computationally strict Nash equilibrium
- Soundness - probability of corrupted or incorrectly recovered secret e.g. not learning the secret or not knowing the secret is unrecoverable is negligable
- Achieved via a checking share

Time-lock Puzzles (TLPs)

- A TLP scheme is one which embeds a secret into a puzzle s.t. it cannot be decrypted until an amount of "time" τ has elapsed
  - Specifically it is an ordered pair of algorithms (PGen, PSolve)
  - $Z \leftarrow \text{PGen}\ (\tau, s)$ is a probabilistic algorithm with hardness or time parameter τ and solution $s \in \{0,1\}$ and outputs a puzzle Z
  - $s \leftarrow \text{PSolve}\ (Z)$ is a deterministic algorithm that takes a puzzle Z and outputs solution s
- A Homomorphic TLP (HTLP) is a tuple of four PPT algorithms (HP.Setup, HP.Gen, HP.Solve, HP.Eval)
- May be a Linearly HTLP (LHTLP), Multiplicative HTLP (MHTLP), or Fully HTLP (FHTLP)

Concrete Instantiation

- Multiplicative HTLP (MHTLP)
- Described in: G. Malavolta and S. A. K. Thyagarajan. Homomorphic time-lock puzzles and applications. In A. Boldyreva and D. Micciancio, editors, Lecture Notes in Computer Science, volume 11692, pages 620-649. Annual International Cryptology Conference, CRYPTO 2019, Springer, 2019.
- Consists of (MHP.Setup, MHP.Gen, MHP.Eval, MHP.Solve) over a ring $(\mathbb{J}_N, \cdot)$ of which Solve is deterministic and the others are probabilistic
- MHP.Solve has complexity $\Omega(2^\tau)$ although exponentiation by sequential squaring allows reduction by $\log_2$
- Hence practically speaking τ time means the time for *tau* λ-bit multiplications
- $\mathbb{J}_p$ for a prime p is the Jacobian subgroup $\mathbb{J}_p \subseteq \mathbb{Z}_p^*$ defined as $\mathbb{J}_p = \{x \in \mathbb{Z}_p^* \| \exists y \in \mathbb{Z}_p^* s.t. y^2 = x \pmod p\}$
- So $\mathbb{J}_p$ is the cyclic group of elements over $\mathbb{N}_p^*$ which have a Jacobian symbol of $+1$

Python verification

- Started with the Python example for SS from Wikipedia:
- https://en.wikipedia.org/wiki/Shamir
- However none of this code beyond Euclidean Greatest Common Divisor (GCD), its extension for field division and polynomial evaluation at a given x value was used
- `_extended_gcd(a, b)`
- `_eval_at(poly, x, prime)`
- `_divmod(num, den, p)` simply calls `_extended_gcd(den, p)`
- Why verify? The paper had several mistakes in the appendix. A careful read is not as certain to find these as implementation will.
  - Public parameters specified an unused prime p where $p > \{s, n\}$ presumably as the polynomial field
  - $y = f(x) \mod p \mod N$ would seems to cause information loss when $f(x) > p > N$
  - Protocol specified r shares to recover a polynomial of degree r which of course must be degree $r - 1$
  - Lagrange interpolation used product not sum symbol for the outer sum

Miller-Rabin primality testing

- The basis for the test, Fermat's little theorem can provide a faster initial check $\forall a \in \mathbb{Z} a^p \equiv a \pmod{p}$ or $a^{p-1} \equiv 1 \pmod{p}$ if a, p are coprime and p is not a Carmichael number
- If $\exists a \| a^d \neq 1 \pmod{n} \wedge a^{2^r d} \neq 1 \pmod{n}$ where $0 \leq r \leq s-1$, s, d are positive integers and d is odd then n is not prime
- In that case a is a witness for compositeness of n
- Otherwise a is a strong liar if n is composite
- $\frac{1}{4}$ of bases a are strong liars so k iterations gives probability $\frac{1}{4^k}$
- Just make k related to the bits/security parameter will be more than sufficient - reductions in rounds and probability bounds possible
- Running time is $\mathcal{O}(k \log^3 n)$ reducible to $\tilde{\mathcal{O}}(k \log^2 n)$ with FFT multiplication

Python implementation - Random prime generation and Miller Rabin primality testing

```
def is_safe_prime(n, k):
  if (n % 12) != 11: return False #n > 7
  return is_all_prime([(n - 1) >> 1, n], [k >> 2, k >> 1])
def is_all_prime(nl, kl): #Miller-Rabin test for k rounds
  if any(n < 3 or (n & 1) == 0 for n in nl): return False
  r, d, idxs = [1 for _ in nl], [n - 1 for n in nl], [i for i in range(len(nl))]
  for i in idxs:
    while d[i] & 1 == 0: #write n as 2^r*d where d odd
      r[i] += 1; d[i] >>= 1
  while len(idxs) != 0:
    if any(not miller_rabin(nl[i], d[i], r[i]) for i in idxs): return False
    for i in idxs: kl[i] -= 1
    idxs = [i for i in idxs if kl[i] != 0]
  return True #probably prime
def is_prime(n, k): return is_all_prime([n], [k])
def miller_rabin(n, d, r):
  a = random.randint(2, n - 2)
  x = pow(a, d, n)
  if x == 1 or x == n - 1: return True
  for _ in range(r - 1):
    x = pow(x, 2, n)
    if x == n - 1: return True
  else: return False #composite
def find_prime(slambda, tester=is_prime):
  while True:
    p = random.randint(1 << slambda, 1 << (slambda+1))
    if tester(p, slambda): return p
```

MHTLP setup - MHP.Setup$(1^\lambda, \tau)$

- $p, q, p\prime = \frac{p-1}{2}, q\prime = \frac{q-1}{2}$ are primes where p, q are λ-bit RSA safe primes, $p\prime, q\prime$ are Sophie Germaine primes
- $N := p * q$ is the RSA modulus making convenient Euler's totient calculation ($\phi(\cdot)$) and its order is $\frac{\phi(N)}{2}$
- Euler's Totient of n is the count of positive integers relatively prime to n up to n
- $\phi(N) = N \prod_{x|n} \left(1 - \frac{1}{x}\right) = (p-1) * (q-1)$ because $\phi(pq) = \phi(p)\phi(q)$ and $\phi(x) = x - 1$ if x is prime
- Sample a uniform $\tilde{g} \leftarrow_\$ \mathbb{Z}_N^*$ and set $g := -\tilde{g}^2 \pmod N$ s.t. $g \in \mathbb{J}_N$ and g is a generator of $\mathbb{J}_N$
- Why it works? Because $\tilde{g}$ is has order $\frac{\phi(N)}{2}$ or $\frac{\phi(N)}{4}$ with all but negligible probability
- Compute $h := g^{2^\tau}$, *tau* in MHTLP is number of modular sequential squarings
- Exponentiation to 2^τ by the dealer can be reduced by $\pmod{\frac{\phi(N)}{2}}$
- Output the four-tuple public parameters $pp := (\tau, N, g, h)$

Python implemention - MHTLP setup

```
def mhp_psetup(slambda, tau):
  #RSA - generate 2 safe/Sophie Germain primes
  p = find_prime(slambda, is_safe_prime)
  q = find_prime(slambda, is_safe_prime)
  n = p * q
  gtilda = random.randint(0, n)
  #https://en.wikipedia.org/wiki/Euler%27s_totient_function#Computing_Euler'
       s_totient_function
  phi = (p - 1) * (q - 1)
  twopowtau = (1 << tau) % (phi >> 1)
  g = -pow(gtilda, 2, n) % n
  if jacobi(g, n) != 1: raise ValueError
  h = pow(g, twopowtau, n)
  return tau, n, g, h
```

Python implementation - MHTLP puzzle generation - MHP.PGen(pp, s)

- Given $pp := (\tau, N, g, h)$
- Sample uniform $r \leftarrow 1, ..., N^2$
- Generate elements $u := g^r \pmod N$, $v := h^r \cdot s \pmod N$
- Output $Z := (u, v)$ as the puzzle

```
def mhp_pgen(pp, s):
  tau, n, g, h = pp
  r = random.randint(1, n * n)
  return pow(g, r, n), (pow(h, r, n) * s) % n
```

Python implementation - MHTLP puzzle solve - MHP.PSolve(pp, Z)

- Given $pp := (\tau, N, g, h)$ and $Z := (u, v)$
- Compute $w := u^{2^\tau}$
- Output $s := \frac{v}{w} \equiv v \cdot w^{-1} \pmod{N} \equiv \frac{(g^{2^\tau})^r \cdot s}{(g^r)^{2^\tau}}$

```
def mhp_psolve(pp, z):
  tau, n, g, h = pp
  u, v = z
  twopowtau = (1 << tau)
  w = pow(u, twopowtau, n)
  invw = _extended_gcd(w, n)
  return (v * invw[0]) % n
```

Python implementation - MHTLP puzzle evaluation - MHP.PEval($\bigotimes, pp, Z_1, \ldots, Z_n$)

- This is the homomorphic multiplication step
- Reduces computational work by a factor of n
- $\bigotimes$ can be regular multiplication ($*$) in concrete example
- $\prod$ here is implied to be $\prod_{\bigotimes}$
- Given $pp := (\tau, N, g, h)$ and $Z_i := (u_i, v_i) \in \mathbb{J}_N^2$
- Compute $\tilde{u} := \prod_{i=1}^{n} u_i \pmod N$ and $\tilde{v} := \prod_{i=1}^{n} v_i \pmod N$
- Output the puzzle $(\tilde{u}, \tilde{v})$

```
def mhp_eval(mult, pp, zs):
  tau, n, g, h = pp
  utilda, vtilda = 1, 1
  for u, v in zs:
    utilda, vtilda = mult(utilda, u) % n, mult(vtilda, v) % n
  return utilda, vtilda
```

Python implementation - Jacobian symbol and random numbers in $\mathbb{J}_n$

- Jacobi symbol is generalization of Legendre symbol from positive odd primes to all positive integers
- +1 is quadratic residue, -1 is non-quadratic residue, 0 if divisible
- 8 mathematical properties combine to yield elegant computation algorithm

```
def rand_j_n(n):
  while True:
    p = random.randint(0, n)
    if jacobi(p, n) == 1: return p
#https://en.wikipedia.org/wiki/Jacobi_symbol
def jacobi(n, k):
  if k <= 0 or k % 2 == 0: raise ValueError
  n %= k
  t = 1
  while n != 0:
    while n % 2 == 0:
      n //= 2
      r = k % 8
      if r == 3 or r == 5: t = -t
    n, k = k, n
    if n % 4 == 3 and k % 4 == 3: t = -t
    n %= k
  return t if k == 1 else 0
```

Python implementation - Polynomial addition and multiplication

```
#multiplication via Karatsuba or faster method...
def mulpoly(poly1, poly2, p): #Kronecker substitution...
  lp1, lp2 = len(poly1), len(poly2)
  result = [0 for _ in range(lp1+lp2-1)]
  for i in range(len(poly1)):
    for j in range(len(poly2)):
      result[i + j] = (result[i + j] + (poly1[i] * poly2[j]) % p) % p
  return result
def addpoly(poly1, poly2, p):
  lp1, lp2 = len(poly1), len(poly2)
  result = [0 for _ in range(max(lp1, lp2))]
  for i in range(len(result)):
    offs1, offs2, offsr = lp1 - 1 - i, lp2 - 1 - i, len(result) - 1 - i
    if i >= lp2: result[offsr] = poly1[offs1]; continue
    if i >= lp1: result[offsr] = poly2[offs2]; continue
    result[offsr] += (poly1[offs1] + poly2[offs2]) % p
  return result
```

Lagrange Interpolation

- Over a polynomial ring $K[x]$ in a prime field K, it is in fact a special case of the Chinese Remainder Theorem (CRT) e.g. the system $f(x) \equiv f(x_i) \pmod{x - x_i}$ since x_i are distinct and $x - x_i$ are pairwise coprime
- $f\prime(x) = \sum_{i \in [1..k-1]} \left[f(y_i) * \prod_{l \in [1..k-1], l \neq i} \frac{x - x_l}{x_i - x_l} \right] =$
  $a_0\prime + a_1\prime x + a_2\prime x^2 + \ldots + a_{k-1}\prime x^{k-1}$
- Assumption: No two x_j can be the same
- If only recovering secret can simplify by setting $x = 0$ s.t.
  $a_0\prime = \sum_{i \in [1..k-1]} \left[f(y_i) * \prod_{l \in [1..k-1], l \neq i} \frac{-x_l}{x_i - x_l} \right]$
- The secret $s = a_0 = a_0\prime$ is recovered if in fact $f\prime(x) = f(x)$
- The polynomial is recovered to check that $f\prime(y_0) = s_0$ and r the correct number of rounds has been determined

Python implementation - Lagrange Interpolation

```
def _lagrange_to_poly(x_s, y_s, p):
  k = len(x_s)
  assert k == len(set(x_s)), "points must be distinct"
  poly = []
  for i in range(k):
    num, den = [y_s[i]], 1
    for l in range(k):
      if i == l: continue
      num = mulpoly([1, -x_s[l]], num, p)
      den *= (x_s[i] - x_s[l]) % p
    inv, _ = _extended_gcd(den, p)
    poly = addpoly(mulpoly(num, [inv], p), poly, p)
  return list(reversed(poly))
```

Protocol (Setup and first part of Share)

- Setup phase: starts with the dealer (D) setting up public parameters
- $pp\prime := (pp_1, pp_2, r, d) \equiv$ $(\mathsf{MHP.Setup}(1^\lambda, \tau), \{y_0, ..., y_m\} \leftarrow_\$ \mathbb{J}_N, r \leftarrow_\$ \mathcal{G}, d \leftarrow_\$ \mathcal{G}\prime)$ where $\mathcal{G} = \{1, ..., N^2\}$, $\mathcal{G}\prime = \{1, ..., N^2\}$ and $m = r + d$
- $\mathcal{G}, \mathcal{G}\prime$ are efficiently samplable distributions suggested as geometric distributions over $\mathbb{N}$ depending on player preferences β, the probability of success in a repeated Bernoulli trial (head or tail coin toss)

Protocol (Setup and first part of Share) (cntd.)

- This is a $(r, r+1)$ threshold scheme s.t. given $r+1$ shares of s, a threshold of r shares is sufficient to reconstruct s
- D generates a secret $s \in \mathbb{J}_N$
- Share phase: D creates shares and sub-shares and puzzles
- D generates a random polynomial $f(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{r-1} x^{r-1}$ where $a_0 = s$ and $\{a_1, ...a_{r-1}\} \leftarrow_\$ \mathbb{J}_N$
- D computes real and fake shares

$$s_i \begin{cases} i \in \{0..r\} & := f(y_i) \pmod N \\ i \in \{r+1..m\} & \leftarrow_\$ \mathbb{J}_N \end{cases}, s_i \in \mathbb{J}_N$$

Python implementation - Protocol Simulation (Setup Phase and first part of Share Phase)

```
def mhp_test(pp):
    n, d = 4, 3
    r = 4
    m = r + d
    s = rand_j_n(pp[1])
    print("Secret", s)
    #while True:
    # p = find_prime(s.bit_length()+1)
    # if p > s and p > n: break
    #print(pp[1].bit_length(), p.bit_length(), s.bit_length())
    p = pp[1]
    y = [rand_j_n(pp[1]) for _ in range(m)]
    poly = [s] + [rand_j_n(pp[1]) for _ in range(r-1)]
    si = [_eval_at(poly, y[i], p) for i in range(r+1)]
    #si = [_eval_at(poly, y[i], p) % pp[1] for i in range(r+1+1)]
    assert poly == _lagrange_to_poly(y[1:r+1], si[1:], p)
    assert s == _lagrange_interpolate(0, y[:r+1], si, p)
    fakes = [rand_j_n(pp[1]) for _ in range(m - r)]
    si = si + fakes
    assert len(si) == m + 1
```

Protocol (Remainder of Share Phase)

- D creates sub-shares $s_{i,j} \leftarrow_{\$} \mathbb{QR}_N$ for $j \in [1..n-1]$ and finally $s_{i,n} = s_i \cdot (\prod_{j=1}^{n-1} s_{i,j})^{-1}$
- Thereby $s_i = \prod_{j=1}^{n} s_{i,j} \pmod N$
- D generates sub-puzzles $\mathcal{Z}_{i,j}$:=MHP.PGen$(pp_1, s_{i,j})$
- D makes lists $\text{list}_j = \{\mathcal{Z}_{1,j}, ..., \mathcal{Z}_{r,j}, \mathcal{Z}_{r+1,j}, ...\mathcal{Z}_{m,j}\}$ for each corresponding player P_j
- Finally D broadcasts public parameters $pp\prime$, checking share s_0 and distributes list_j to P_j for $j \in [1..n]$ to each player

Python implementation - Protocol Simulation (cntd.)

```
subshares = [[] for _ in si]
puzzles = [[] for _ in si]
for i in range(m):
  prod = 1
  for q in range(n-1):
    subshares[i].append(rand_j_n(pp[1]))
    puzzles[i].append(mhp_pgen(pp, subshares[i][q]))
    prod = (prod * subshares[i][q]) % pp[1]
  inv = _extended_gcd(prod, pp[1])
  subshares[i].append((si[i+1] * inv[0]) % pp[1])
  puzzles[i].append(mhp_pgen(pp, subshares[i][n-1]))
sk = []
for i in range(m):
  prod = 1
  for q in range(n):
    prod = (prod * subshares[i][q]) % pp[1]
  sk.append(prod)
assert sk == si[1:]
```

Protocol (Reconstruction Phase)

- k rounds occur where $1 \leq k \leq m$
- On k-th round, players directly send or broadcast sub-share $s_{k,j}$ to all other players
- Starting on the second round (otherwise polynomial was a constant value), players compute:
- $\mathcal{Z}_k \leftarrow \text{MHP.PEval}(\bigotimes, pp_1, \mathcal{Z}_{k,1}, ...\mathcal{Z}_{k,n})$
- $s_k \leftarrow \text{MHP.PSolve}(pp_1, \mathcal{Z}_k)$
- $f\prime(x)$ recovered by Lagrange interpolation of $s_1, ..., s_k$
- $\{s, \perp\} = \begin{cases} f\prime(0) & f\prime(y_0) \pmod N = s_0 \\ \perp & \mathcal{Z}_{k,n} = \emptyset \vee \text{list}_j = \emptyset \end{cases}$
- Secret recovered if checking share verified
- Secret deemed unrecoverable if the sub-share list is exhausted or any player quit communication in a given round

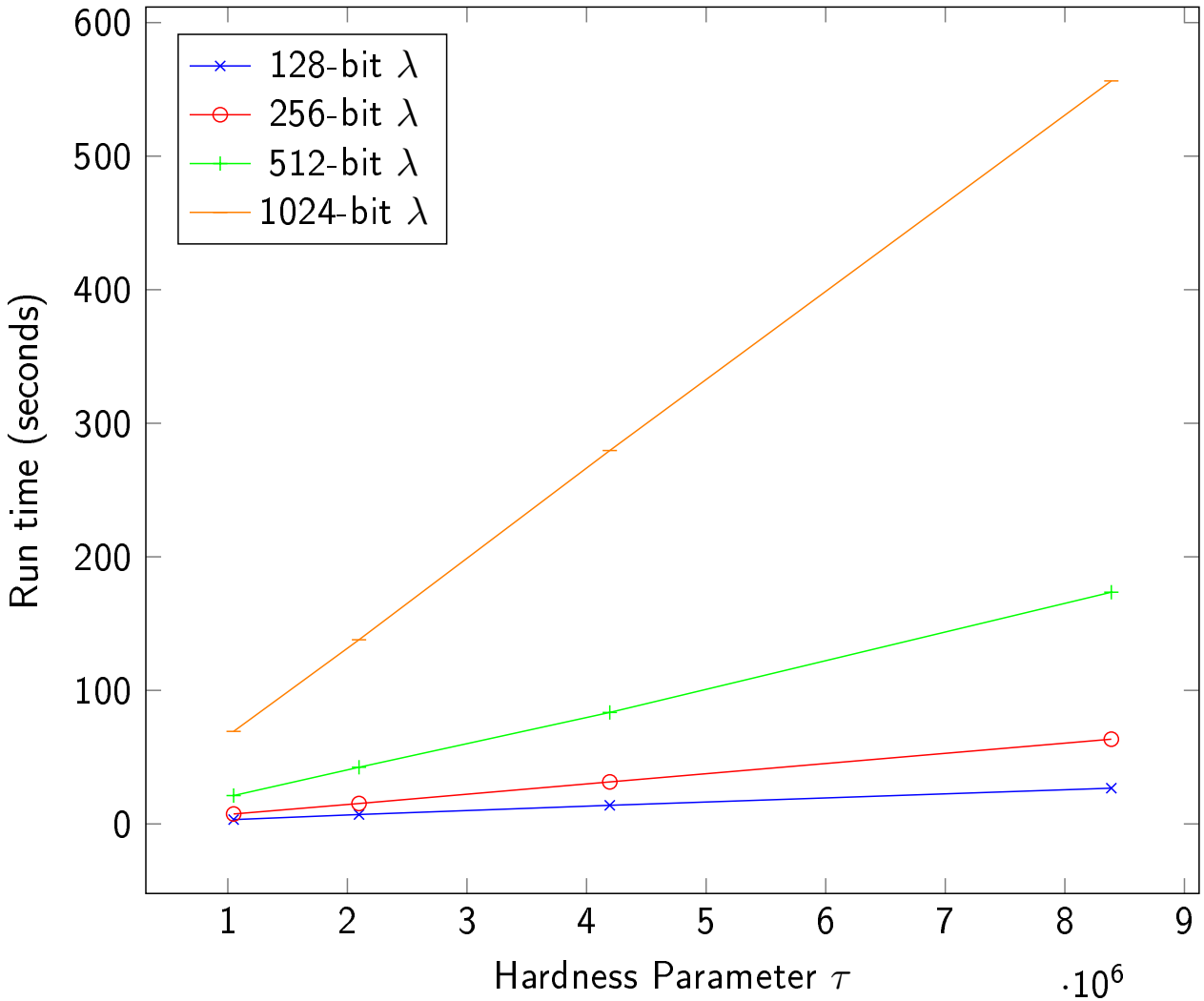
Python implementation - Protocol Simulation (cntd.)

```
s0 = si[0]
#print([mhp_psolve(pp, mhp_pgen(pp, si[i])) for i in range(m)])
sk = [mhp_psolve(pp, mhp_eval(lambda a, b: a * b, pp, puzzles[i])) for i in
     range(m)]
for k in range(m):
  polyprime = _lagrange_to_poly(y[1:1+1+k], sk[:1+k], p)
  print("Secret recovered: ", _lagrange_interpolate(0, y[1:1+1+k], sk[:1+k],
       p))
  ver = _eval_at(polyprime, y[0], p) == s0
  print("Verification check: ", ver)
  if ver == True: break
  if k >= r: assert False
```

Python implementation - example output

```
Secret
     177752076808708683724019157938862414660819843138117806454785143209647261648387
Secret recovered:
     429926408749594284359856000541320288567576339847176184896274878646743940684 00
Verification check: False
Secret recovered:
     47882224558090490054864839956187730250673258445210066817503657188159850231087
Verification check: False
Secret recovered:
     11967541157208361281115463917404375610361073785250643720037101448241686036710 4
Verification check: False
Secret recovered:
     177752076808708683724019157938862414660819843138117806454785143209647261648387
Verification check: True
```

Experimental Results (n=4, r=4, d=3, compute MHTLP.Eval regardless based on m=7)



Experimental Results explained

- Security parameter 1^λ non-linear likely due to quadratic polynomial and modular multiplication ($\mathcal{O}(n^2)$) and time parameter τ is linear with respect to computation time as expected
- Obviously a professional library would use all sorts of better optimized/vectorized code with better algorithms in a language like C/C++ e.g. highly optimized modular multiplication, polynomial multiplication with Kronecker substitution ($\mathcal{O}(*)$) and Karatsuba $\mathcal{O}(n^{\log_2 3})$ or Schönhage-Strassen $\mathcal{O}(n \log n \cdot \log\log n)$ or Harvey and van der Hoeven ($\mathcal{O}(n \log n)$).
- Prime number generation has large error margin in time needed, increases dramatically with bit size, not used or considered as part of measurement

```
output = []
for slambda in [128, 256, 512, 1024, 2048, 4096]:
  for tau in [1024*1024, 1024*1024*2, 1024*1024*4, 1024*1024*8]:
    pp = mhp_psetup(slambda, tau)
    output.append((slambda, tau, timeit.timeit(lambda: mhp_test(pp), number
        =1)))
print(output)
```

Weaknesses of concrete scheme

- s_0 can allow any player to compute the potential secret every round
- Suppose an oracle exists which verifies the secret, now $r-1$ rounds would be sufficient to use the checking share as an interpolation share
- If an adversary knows the factoring of the prime N then the hard problem which raises to power 2^τ can be simplified as the dealer does in the setup phase by computing the totient
- Therefore the RSA primes and modulus must be changed frequently enough based on the difficulty determined by the security parameter
- Obviously not a post-quantum algorithm as it depends upon hardness of factoring
- Cryptographically Secure Pseudo Random Number Generate (PRNG) should always be used in real code unlike in example

Properties

- Due to side information (protocol induced auxiliary information) related to the secret must prove properties
- HTLP scheme
  - Correctness (assumed from MHTLP paper proof)
  - Security (output of scheme is indistinguishable from random to an eavesdropping adversary)
  - Compactness (the size of the circuit to evaluate the puzzles)
    - Non-trivial
    - Complexity of decrypting ciphertext does not depend on the function used to evaluate the ciphertext
    - Intuitively, ciphertext size should not grow through homomorphic operations and the output length only depends on security parameter
- SS scheme (both assumed, proven in prior papers)
  - Correctness
  - Secrecy
- Checking share side information correctness

Assumptions

- Strong RSA Modulus
  - Given two λ-bit safe primes $p = 2p\prime + 1, q = 2q\prime + 1$, and $N = p * q$
  - Given e a randomly chosen prime s.t. $2^{\lambda} < e < 2^{\lambda+1} - 1$, and $h \in \mathbb{QR}_N$
  - $\mathbb{QR}_N$ is the group of quadratic residues in $\mathbb{Z}_N^*$ of order $p\prime * q\prime$
  - It is hard to compute $x^e \equiv h \pmod{N}$
- Sequential Squaring
- Given N is a strong RSA modulus, g is a generator of $\mathbb{J}_N$, $\tau(\cdot)$ is a polynomial
- There exists $0 < \epsilon < 1$ s.t. every polynomial-size adversary $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$ who's depth is bounded from above by $\tau^{\epsilon}(\lambda)$ there exists negligible function $\mu(\cdot)$ s.t.:
- $$Pr\left[b \leftarrow \mathcal{A}(N, g, \tau(\lambda), x, y) : \begin{cases} x \rightarrow_{\$} \mathbb{J}_N; b \leftarrow_{\$} \{0,1\} \\ \text{if } b = 0 \text{ then } y \leftarrow_{\$} \mathbb{J}_N \\ \text{if } b = 1 \text{ then } y := x^{2^{\tau}} \end{cases} \right] \leq \frac{1}{2} + \mu(\lambda)$$
- $x \in \mathbb{J}_N, y \in \mathbb{J}_N$ avoids trivial attacks involving computing the Jacobi symbol of the group element

Assumptions (cntd.)

- Decisional Diffie-Hellman (DDH)
- Given N is a strong RSA modulus
- For every polynomial-size adversary $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$ there exists negligible function $\mu(\cdot)$ s.t.:
- $$Pr\left[b \leftarrow \mathcal{A}(N, g, g^x, g^y, g^z) : \begin{cases} (x,y) \leftarrow_\$ \{1, ..., \frac{\phi(N)}{2}\}; b \leftarrow_\$ \{0,1\} \\ \text{if } b = 0 \text{ then } z \leftarrow_\$ \{1, ..., \frac{\phi(N)}{2}\} \\ \text{if } b = 1 \text{ then } z := x \cdot y \pmod{\frac{\phi(N)}{2}} \end{cases} \right] \leq \frac{1}{2} + \mu(\lambda)$$
- Intuitively g^{xy} computationally indistinguishable from g^z
- Related to the discrete log assumption - namely if r is recovered from $u = g^r \pmod N$ then $v = h^r * s \pmod N$ can be trivially solved for s

Concluding Remarks

- Elegant scheme for using Time Lock Puzzles in context of Rational Secret Sharing
- Important primitive in several protocols for secure multiparty computation
- RSS is actually used in the field to secure servers even in the case of multiple server failures.
- By the dealer acting as several participants, distributing shares among participants.
- If a server is hacked, the secret is not known as long as fewer than t shares are stored on the server.
- Security by de-centralization
- Decentralized voting protocols - though TLP not really constructive here

Thank you for your attention!

- Thanks to Professor Peter Ligeti for excellent paper suggestions
- Please find this presentation and code on GitHub
- https://github.com/GregoryMorse/mhtlpss
- We have a bit of a polynomial/Secret Sharing theme:
  - Professor Laci Csirmaz "Homogeneous secret sharing"
  - Professor Gyarmati Mate "Regular bipartite secret sharing schemes"
  - Professor Seres Pisti "Polynomial commitment schemes and their applications in cryptography"