Chapter 8:
An Implementation of
Tagged Private Information Retrieval*

Adam L. Young and Moti M. Yung

Abstract

In this chapter we present an experimental implementation of the *tagged private information retrieval* protocol (TPIR). Informally, a TPIR protocol retrieves data from a database without revealing the entry that the data is taken from and that satisfies additional security properties. Let $B = ((t_0, b_0),(t_1, b_1),...,(t_{W-1}, b_{W-1}))$ be a database with W entries. Data b_i is "tagged", (i.e., uniquely identified) using the tag string t_i for $i = 0, 1, ..., W - 1$. A user wants b_i from the database but does not trust the database administrator to know the tag t_i. In a TPIR protocol, the user supplies t_i to algorithm `QueryGenerator` which outputs a private key and a query q. The query is given to the database administrator who then passes B and q to `DatabaseAlgorithm`. The output of this algorithm is a response r. The administrator sends r to the user. The user supplies r and the private key to `ResponseRetriever` which then outputs b_i. These 3 algorithms are public and the protocol satisfies the following properties: (1) q does not reveal t_i, (2) the user trusts the database administrator to run `DatabaseAlgorithm` on the correct inputs and return r to the user, and (3) b_i can only be recovered from r using the user's private key. This chapter implements a modified version of the TPIR protocol that was introduced in Section 6.4 of the book *Malicious Cryptography* [10], a protocol that is closely related to [1]. We apply TPIR to implement an experimental password-snatching cryptotrojan that does not reveal the unique login/password pair that it snatches. The TPIR algorithm that we present can be applied to numerous PIR problems.

1 Introduction

This chapter presents an experimental implementation of tagged private information retrieval. We focus on how to run and interpret the results of our experimental program and important aspects of the ANSI C code. Our goal is not to give the theory behind TPIR. Readers interested in other aspects of private information retrieval (PIR) should look elsewhere, i.e., the scientific literature that introduced and advanced PIR [3, 4, 2, 7]. This chapter and the associated appendix that contains the corresponding source code constitute *Fundamental Research* in cryptovirology.

There are numerous applications of PIR. For example, PIR can be used to conduct patent searches without revealing to the patent database administrator the intellectual property of interest. It can be used in bioinformatics to identify common symptoms of patients suffering from a particular disease without compromising the privacy of the patients.

PIR can be used for malicious purposes as well. It can form the basis of a *cryptotrojan* that covertly steals information from the host system without revealing the information that is sought. The research that we present in this chapter therefore has both positive and negative applications.

The TPIR protocol that we present, which we will call TPIR2, is a modified version of the TPIR protocol, call it TPIR1, that appears in Chapter 6, Section 6.4 of [10]. TPIR1 is closely related to the two-round computational PIR protocol of Cachin et al [1], which for brevity we will call the CPIR protocol. CPIR assumes the existence of a single database and it is based on two new complexity assumptions, the phi-hiding and phi-sampling assumptions.

TPIR2 differs from TPIR1 in several ways. Additional constraints are imposed by TPIR2 on the composite $m = q_1 q_2$ that is used in both TPIR1 and TPIR2. More specifically, the prime q_i is chosen such that $q_i - 1$ is divisible by a large prime for $i = 1, 2$. This makes m hard to factor. TPIR1 utilizes a random oracle to implement a random function. In contrast, TPIR2 utilizes SHA-256 to heuristically approximate a random function. The algorithm `ResponseRetriever` has been significantly optimized to run faster. Since we are interested in a real-world experimental application here, we do not "pass" the security parameter 1^k as an argument. There are other changes as well and Section 6.4 of [10] may be consulted for more information.

We state up front that TPIR2 is *not* an implementation of CPIR. The TPIR protocols are based on a problem that is closely related to the phi-hiding decision problem. Informally, the TPIR protocols are based on the

hardness of deciding whether or not p is in the prime power decomposition of $\phi(m)$ given (p, m). The protocols rely on a similar phi-sampling assumption as well. However, so many aspects of the protocols differ from CPIR that the proof of security of CPIR *does not apply* to TPIR2 (nor to TPIR1).

The TPIR protocols are designed to shrink the size of the query that the user sends to the database administrator. They are also designed to privately retrieve an entire bit string as opposed to a single bit. They are intended to be efficient enough to be used in practice for certain applications. They are based on known problems in computational complexity. We speculate that TPIR2 is secure. However, we have no formal proof of security to support this claim.

So, what does this mean in practical terms? What it means is that we have forgone the usual practice of formally restricting weaknesses to well-defined axioms in favor of an implementation with appreciable performance. Our claim that TPIR2 is secure could be entirely wrong. If we are wrong and an algorithm is discovered that, say, breaks TPIR2, then it may be entirely incorrect to assume that CPIR is flawed. One would need to analyze the reduction that breaks TPIR2 and see if it can be adjusted to break CPIR. We disclaim any and all flaws in the implementation, both theoretical and accidental.

Why would we go through the trouble to design and implement an experimental TPIR scheme with no formal proof of security? Because we believe that a heuristic scheme with no apparent weaknesses, that is computationally *efficient* enough to be used to solve real-world cryptovirology problems and otherwise, merits both scientific exploration and open discussion.

2 Background on Password Snatching

A fundamental problem in cryptovirology is to devise malware that steals information from a host machine in such a way that the attacker is protected insofar as possible. Some of the issues that an attacker must consider are as follows.

Property 1. Deploying the malware without being traced.

Property 2. Aggregating the stolen information without being noticed.

Property 3. Aggregating the information (plaintext) without it being available to others (e.g., malicious interlopers, sysadmins).

Property 4. Obtaining the information from the malware without being traced.

Other properties include preventing others from interfering with the operation of the malware. Types of interference include: introducing faults, altering the stolen data, forging command and control information to/from the malware, and so on.

The classic type of information that is stolen is login/password pairs of users of the host system. In this context we may imagine two other specific attack scenarios:

Problem 1: The attacker is interested in the most recently used (MRU) login/password pairs since users may periodically change their passwords and new users may be added to the system.

Problem 2: The attacker is interested in the login/password pair of a particular user.

Cryptovirology views information theft via malware *as a scientific problem* and brings to bear both new notions and existing notions in modern cryptology to solve it.

2.1 Deniable Password Snatching

The problem of devising a cryptovirus, cryptoworm, or cryptotrojan to steal information with properties that are desirable to the attacker is well-known in cryptovirology. The paper [9] introduces an attack called *deniable password snatching* that we now review:

Problem 1: An attacker is interested in obtaining indiscriminately the MRU login/password pairs on an isolated (non-networked) machine M. The attacker has access to M and the machine is used by many users.

Solution: The attacker generates a public key y and corresponding private key x. The key y is placed in a cryptotrojan T. The attacker writes a virus V that has for a payload a function that installs T. A program is infected with V and is placed on a disk. The attacker inserts the disk into M, runs the program, and walks away. V installs T in M. T operates as follows. It captures each login/password pair that is entered. It encrypts the pair with y to get ciphertext c. It stores c in a circularly linked list data structure L that is saved to a hidden password file. The Trojan uses the linked list to implement a MRU login/password storage method. So, when a new ciphertext c is obtained, the Trojan T overwrites the oldest ciphertext in L with c.

Consider the case that a writable disk D is inserted into M. If there is enough free space on D and the user initiates a write to the disk (e.g., for a

text file) then T copies L into the highest unused sectors on D. T ensures that the sectors are left marked as "unused" by the underlying file system.

The attacker returns to M sometime later. He inserts a disk, creates an innocuous text file on M and saves it to the disk. The Trojan "pushes" L onto D automatically as previously described. The attacker walks away, carefully extracts L from the "unused" sectors, and decrypts the elements of L using x.

Security: The four properties are satisfied as follows. If the attacker is caught with the disk before T is installed then he or she claims to be an innocent victim of V (Property 1). The circularly linked list ensures that the hidden password file remains fixed in size, as opposed to growing monotonically larger over time. This helps lower the chance that the hidden file is found (Property 2). The use of an asymmetric cryptosystem to encrypt the login/password pairs prevents everyone except the attacker from decrypting the ciphertexts (Property 3). The Trojan T effectively broadcasts L in a covert fashion to everyone that inserts a writable disk with sufficient free space into M. So, the attacker cannot be readily distinguished from other users when he obtains L (Property 4).

The attack is *deniable* because in every phase the attacker can deny being the attacker. During deployment the attacker can claim to be a victim of the self-propagating program V. In the broadcast phase the attacker can claim to be one of hundreds or even thousands of people that inadvertently received the broadcast.

The paper [9] covers extensions to the above attack as well. It notes the possibility of having T steganographically encode L into media and posting it to a bulletin board (e.g., Usenet). It also notes the utility of probabilistically re-encrypting[1] the elements of L (e.g., using ElGamal). This enables T to change all entries of L periodically, thereby obfuscating the history of the login/password pairs that have been stored in L.

Section 6.7 of [10] extends deniable password snatching by incorporating a *Malware Loader*. Suppose that instead of being isolated, the machine M is on a network. The deployment phase of the attack can be strengthened be minimizing the logic in the Trojan T, the logic that reveals the password-stealing nature of T. We will now review Section 6.7.

The attacker places a digital signature verification public key y_s in T. The Trojan T "listens" to broadcasts made to a public bulletin board B (or other public channel). The attacker crafts a password snatching Trojan T_p and code signs it using the private signing key x_s corresponding to y_s.

[1] This is sometimes referred to as "blinding" the encryption.

The result is then encrypted with x and is then steganographically encoded into media. The attacker posts the media (e.g., through a mix-net) to B. T obtains the broadcast, steganographically decodes it, decrypts it,[2] and verifies the code-signed Trojan using y_s. If the signature is valid then T_p is installed in M by T. T_p also zeroizes all traces of T (and V if present).

So, the purpose of T is ultimately to install T_p that is obtained from the network. If T is found within V on the attacker's person during the deployment phase then no password-stealing logic will be found.

2.2 The Targeted Deniable Password Snatching Attack

The work in [9] is ideally suited for stealing login/password pairs indiscriminately using a MRU priority. But it does not ideally address the problem of having the Trojan steal the login/password pair of a particular user where the login is known a priori. This is because, through bad luck, the list L may not contain the desired login/password pair when L is obtained by the attacker.

TPIR provides an elegant solution to Problem 2. The abstract and introduction of this chapter provides a fair amount of background on TPIR. The only additional properties that need to be presented in order to describe the solution to Problem 2 are as follows.

1. For our TPIR protocol, the data structure for the query q is identical to that of the response r.

2. For each entry of the database $B = ((t_0, b_0), (t_1, b_1), ..., (t_{W-1}, b_{W-1}))$ that is processed by the database administrator, the response r is updated.

3. The TPIR protocol is designed in such a way that `DatabaseAlgorithm` is executed on the database entries $0, 1, 2, ..., W-1$ in order. Let S_1 be subset of $T = \{0, 1, 2, ..., W-1\}$. Let S_2 be a multiset of S_1. For example, if $T = \{0, 1, 2, 3, 4, 5, 6, 7\}$ then S_2 could be $\{4, 1, 2, 1, 6\}$. The TPIR protocol is compatible with the following operation. The database administrator runs the database algorithm on a sub-multiset of B in order. In our example, the database algorithm would be run on (t_4, b_4),(t_1, b_1),(t_2, b_2),(t_1, b_1),(t_6, b_6), in this order. TPIR2 (and TPIR1) can be correctly run in this fashion. The only issue is that if a database entry is not processed then it cannot be retrieved.

[2] There is an implicit integrity check during decryption for chosen-ciphertext secure cryptosystems.

Let the database B be the login/password pairs of users that are authorized to use M. The attacker wants the password b_i belonging to the user having login t_i. The attacker supplies t_i to `QueryGenerator`. The output is a private key and query q. The password snatching Trojan includes `DatabaseAlgorithm` and q.

Suppose a user logs into M by supplying (t_j, b_j) to M. When this happens, the Trojan runs `DatabaseAlgorithm` on input (q, t_j, b_j), resulting in response r. The Trojan overwrites q with r.

Now suppose that the pair (t_k, b_k) is entered into M. The previous response r is used as the query. The Trojan runs `DatabaseAlgorithm` on input (r, t_k, b_k). The Trojan then overwrites the old r with the new response r that is output by `DatabaseAlgorithm`. The new r is used as the query for the next login/password pair, and so on.

The Trojan covertly broadcasts the current value of r in the same fashion as in the deniable password snatching attack that is reviewed in Subsection 2.1. This can be a regular occurrence or due to some stimulus (e.g., a writable disk being inserted into the machine). The attacker must somehow obtain this broadcast to complete the password snatching attack.

The attacker supplies the private key and the response r to the algorithm `ResponseRetriever`. This results in the output b_i provided that user t_i has logged in since the time that the Trojan was installed.

The security properties are straightforward. The Trojan in no way exposes the login of interest t_i. This protects the interests of the attacker. Also, the response r remains fixed in size. This helps ensure that Property 2 holds, since r will not grow in size over time.

3 Creating the Program

The program consists of the ANSI C source file `csis.c`. We constructed a simple makefile for this file. The program was compiled using `gcc`. We used the Minimalist GNU for Windows development suite (MinGW). The program utilizes OpenSSL for the underlying crypto and big number primitives. It also utilizes the Microsoft Cryptographic API (MS CAPI) to seed the OpenSSL random number generator. Compiling the program results in the MS DOS command-line program `csis.exe`.

4 Running the Program

The program starts by printing out copyright information and some bibliographic references. RAND_status() is an OpenSSL function that returns 1 if the OpenSSL library believes that it has been seeded with a sufficient number of random bytes.

```
"csis" Copyright (c) 2005-2006 by
Moti Yung and Adam L. Young. All rights reserved.
This is an experimental implementation of the
Tagged Private Information Retrieval (TPIR)
algorithm (with modifications) described in
Section 6.4 of Chapter 6 entitled
"Computationally Secure Information Stealing"
in the book:
  "Malicious Cryptography: Exposing Cryptovirology"
  by Adam Young & Moti Yung, Wiley, 2004.
See also:
  C. Cachin, S. Micali, M. Stadler, "Computationally
  Private Information Retrieval with Polylogarithmic
  Communication", Proc. of Eurocrypt, 1999,
  Springer-Verlag, pp. 402-414, 1999.

RAND_status() returned 1.
Computationally Secure Info Stealing Functions:
Type (a) to generate a query.
   This creates/overwrites "query.bin" and "privkey.txt".
Type (b) to process the query (this may be
   re-invoked multiple times for processing).
   This updates "query.bin".
Type (c) to recover the answer to the
   query using the private key.
   This reads in "query.bin" and "privkey.txt".
Type (d) to conduct 200 stress tests.

Enter command (a-d) :
```

This chapter will only cover commands (a) through (c). Command (d) is used for software testing purposes.

For very long numbers expressed in hexadecimal, the middle part of the number is omitted and instead shows ellipsis (...). A decimal number printed

between "[" and "]" is the length in bits of the number written immediately to the left.

Command (a) prompts the user for a login name. This string is used as the input tag t_i to the query generator. Tag t_i corresponds to the database entry (t_i, b_i) in the TPIR2 protocol. The login is passed to the algorithm `QueryGenerator` which outputs a private key and query q.

The query and private key are stored in the output files `query.bin` and `privkey.txt`, respectively. In this example we use the tag t_i = "ayoung". The output of the program explains what some of these quantities are and how they are related.

```
Enter command (a-d) : a

The search string for the query in this example program is the
login of the user. (tstr,bstr) is the (login,password) pair.
The login (tstr) is the tag that is used to identify the
associated password. Query generation can take SEVERAL MINUTES.

Enter login: ayoung

Input to RandPrime() is the tag tstr = "ayoung"
RandPrime() output the probable prime p =
 E598BF5B847F0297328C2F6F77CEC88B [128]
Now Phi-Hiding p in m = q_1 * q_2...
Generating 320-bit divisor r_1 of q_1-1...
probable prime r_1 = D82A9FA248F45A28...72AD21AAC50290B5 [320]
Generating 512-bit probable prime q_1 = c_1*p*r_1 + 1...
c_1 = F3F8C1362EAAC706 [64]
probable prime q_1 = B8C329BD2629ADF5...C5026402ED430EAB [512]
Generating 320-bit divisor r_2 of q_2-1...
probable prime r_2 = F5280CC8F0C15148...8FF4935E87A5AA11 [320]
Generating 512-bit probable prime q_2 = c_2*r_2 + 1...
c_2 = F35CF05DD46E38A2D548A3888386EFF1D645719BBB65654E [192]
probable prime q_2 = E90E04B6BBB5C45E...D9FF33333447862F [512]
Computing u = (q_1-1)/p...
m = A833BF105EF465E1...F0684E5F766A3365 [1024]
Generating x[0],x[1],...,x[95] for query...
x[i] is a random element in Z_m^* for i = 0,1,...,95.
Wrote q_1 and u to the file named "privkey.txt".
Wrote m,x[0],x[1],...,x[95] to the file named "query.bin".
Query generation complete for tstr.

--------Memory Leaks displayed below--------
```

```
--------Memory Leaks displayed above--------
```

The file `privkey.txt` contains (q_1, u) in ASCII. The file `query.bin` encodes the query in binary. A hex editor can be used to view the numbers that comprise the query.

The last portion of the output indicates the presence or absence of memory leaks. We utilize OpenSSL's memory leak checking functionality in our program. As long as there is no output between these two lines, there should not be any memory leaks in the implementation.

The query and the response to the query have the same file format, m followed by $x[0], x[1], ..., x[W-1]$. Command (b) prompts the user for a login/password pair. In this example, we entered the login "ayoung" and the password "12344321". Command (b) passes this pair and the contents of `query.bin` to `DatabaseAlgorithm` which outputs a response r. The file `query.bin` is overwritten with r.

```
Enter command (a-d) : b

This command implements the database algorithm that privately
retrieves the password. At most 12 bytes of the password will
be stored. If the correct login is entered then the password
that follows will be stored in bits b_0,b_1,...,b_95 of bstr.
Note that entering (login1,pw1) followed by (login1,pw2) where
pw1 is not equal to pw2 will cause the retrieved password to
to be the bitwise logical OR of pw1 with pw2. Here login1 is
the tag that is encoded in the query.

Enter login   : ayoung
Enter password: 12344321

Read m,x[0],x[1],...,x[95] from the file named "query.bin".
Input to RandPrime() is the tag tstr = "ayoung"
RandPrime() output the probable prime p =
 E598BF5B847F0297328C2F6F77CEC88B [128]
Let b[j] denote bit j of bstr = "12344321".
During updating x[j] = x[j]^(p^b[j]) mod m is computed
for each j in which b[j] = 1. Here j ranges from 0
to 95.

Updating byte 1.
Updating byte 2.
```

```
Updating byte 3.
Updating byte 4.
Updating byte 5.
Updating byte 6.
Updating byte 7.
Updating byte 8.
Updating byte 9.
Updating byte 10.
Updating byte 11.
Updating byte 12.

Wrote m,x[0],x[1],...,x[95] to the file named "query.bin".

--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------
```

It is instructive to run command (b) multiple times. For instance, the login "ayoung" and password "12344321" can be entered multiple times. Other login and password pairs can also be entered. As long as "12344321" is always entered when the login "ayoung" is used, command (c) will properly retrieve "12344321" in this example.[3]

A particularly good test is to run command (b) on (t_{i-1}, b_{i-1}) then on (t_i, b_i), and then on (t_{i+1}, b_{i+1}). The main constraint on these 6 strings for the test is that $t_{i-1} \neq t_i \neq t_{i+1}$. Recall that the query was generated based on t_i. The reason that this test is good is because it modifies the query/response both before and after (t_i, b_i) is processed by the database algorithm. The value b_i should be privately retrieved correctly in this test. Command (d) invokes the function `StressTest` that does exactly this test many times over.

Command (c) reads in the files `query.bin` and `privkey.txt`. The contents of `query.bin`, which is the response to the database algorithm, is supplied to algorithm `ResponseRetriever` along with the private key. As shown in the example output, the password "12344321" is properly retrieved and printed to the screen.

```
Enter command (a-d) : c

Read m,x[0],x[1],...,x[95] from the file named "query.bin".
```

[3]Technically, this only holds with overwhelming probability.

```
Read q_1 and u from the file named "privkey.txt".
Computing v = ((x[i] mod q_1)^u mod q_1) == 1)
for i = 0,1,...,95. The == operator listed above is the ANSI C
equality testing operator. So, v is 1
only when (x[i] mod q_1)^u mod q_1 is 1.

The recovered data (given in hexadecimal) is:
  313233343433323100000000
First 8 characters of password (if it was recovered) is:
  "12344321"
Note: If the listed password is 12 characters then
possible additional characters could be missing.

--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------
```

5 The Query Generator

We start our explanation of the query generator by covering the function that transforms the tag string (i.e., the login string) into a probable prime. The function `RandPrime` takes as input the tag string and returns a number p that is `PHI_HIDDEN_PRIME_LEN` $= 128$ bits in length. The probability that the transformation returns p where p is prime is overwhelming.

`RandPrime` operates as follows. The tag string `tstr` is hashed using the SHA-256 hash algorithm [8]. 128 contiguous bits are taken from the hash as indicated in the code for `RandPrime` below. The uppermost 3 bits of these 128 bits are overwritten with 111 and the least significant bit is overwritten with 1. The resulting value is a candidate value for p. The number p is then subjected to the Miller-Rabin probabilistic primality test. The number of iterations for the test is determined by the function `BN_prime_checks_for_size` that is based on [6]. If p is determined to be composite then the algorithm sets $p \leftarrow p + 2$ and Miller-Rabin is run again, and so on.

This incremental search has the potential to result in a value p that is greater than `PHI_HIDDEN_PRIME_LEN` bits in length. For this reason the loop checks the size of the candidate value p and if it grows to more than `PHI_HIDDEN_PRIME_LEN` bits in length then the program halts with failure.

```
void RandPrime(BIGNUM *p,const char *tstr)
{
```

```
unsigned char hash[32];
SHA256_CTX shactx;

printf("Input to RandPrime() is the tag tstr = \"%s\"\n",tstr);
BN_CTX *ctx = BN_CTX_new();
SHA256_Init(&shactx);
size_t len = strlen(tstr);
SHA256_Update(&shactx,tstr,len);
SHA256_Final(hash,&shactx);
hash[0]  |= 0xE0;
hash[15] |= 0x01;
BN_bin2bn(hash,16,p);
int checks = BN_prime_checks_for_size(PHI_HIDDEN_PRIME_LEN);
for (;;)
   {
   if (BN_is_prime_fasttest(p,checks,NULL,ctx,NULL,1) == 1)
      break;
   BN_add_word(p,2); /* do incremental search for prime */
   if (BN_num_bits(p) > PHI_HIDDEN_PRIME_LEN)
      {
      printf("ERROR: preimage maps to a value p that is\n");
      printf("too large. This is a very rare occurrence.\n");
      exit(1);
      }
   }
PrintBigS("RandPrime() output the probable prime p =\n ",p);
BN_CTX_free(ctx);
}
```

`QueryGenerator` supplies the tag string to `RandPrime` to obtain p. The value p is then phi-hidden in the composite $m = q_1 q_2$. The phi-hiding is accomplished by passing p to `PhiHide` which returns the probable primes q_1 and q_2. The values $u = (q_1 - 1)/p$ and $m = q_1 q_2$ are computed. Following this the values $x[0], x[1], ...,$ and $x[W-1]$ are chosen randomly from $\mathbb{Z}_m^*$. The query is $(m, x[0], x[1], ..., x[W-1])$. The private key is (q_1, q_2, u). However, only (q_1, u) is stored to `privkey.txt` since these are the only values that are needed to retrieve b_i from the response.

```
void QueryGenerator(querystruct *query,csisprivkey *privkey,
   const char *tstr)
{
BN_CTX *ctx = BN_CTX_new();
```

```
BIGNUM *tmp = BN_new();
BIGNUM *p = BN_new();
RandPrime(p,tstr);
PhiHide(privkey->q1,privkey->q2,p);
printf("Computing u = (q_1-1)/p...\n");
BN_sub(tmp,privkey->q1,BN_value_one());
BN_div(privkey->u,tmp,tmp,p,ctx);
BN_mul(query->m,privkey->q1,privkey->q2,ctx);
PrintBigS("m = ",query->m);
printf("Generating x[0],x[1],...,x[%d] for query...\n",W-1);
printf("x[i] is a random element in Z_m^* for i = 0,1,...,%d.\n",
   W-1);
int i=0;
for (;i<W;i++)
   for (;;)
      {
      BN_rand_range(query->x[i],(BIGNUM *) query->m);
      if (IsInZmstar(query->x[i],query->m,ctx))
         break;
      }
BN_clear_free(p);BN_clear_free(tmp);
BN_CTX_free(ctx);
}
```

`PhiHide` generates q_1 and q_2 using the algorithms `HideInPrime` and `GenerateQ2`. `HideInPrime` generates q_1 and `GenerateQ2` generates q_2. The value p is passed to `HideInPrime` so that it can generate a q_1 such that p divides $q_1 - 1$ evenly.

```
void PhiHide(BIGNUM *q1,BIGNUM *q2,const BIGNUM *p)
{
printf("Now Phi-Hiding p in m = q_1 * q_2...\n");
HideInPrime(q1,p);
PrintBigS("probable prime q_1 = ",q1);
GenerateQ2(q2);
PrintBigS("probable prime q_2 = ",q2);
}
```

The algorithm `HideInPrime` generates a number r_1 randomly that is (`kover2` - `PHI_HIDDEN_PRIME_LEN` - 64) bits in length. The 2 most significant bits of r_1 are set to 1. If Miller-Rabin indicates that r_1 is prime then r_1 is

determined and the generation of q_1 continues. The number q_1 that is output will have r_1 as a factor of $q_1 - 1$.

After r_1 is found the algorithm proceeds to generate a random factor c_1. This number is generated such that the 3 most significant bits of it are 1 and the least significant bit is 0. `HideInPrime` tests the primality of $q_1 = c_1 r_1 + 1$. If q_1 is `kover2` bits in length and Miller-Rabin indicates that it is prime then q_1 is the final return value. Otherwise, another candidate factor c_1 is chosen randomly and q_1 is tested again, and so on.

```
void HideInPrime(BIGNUM *q1,const BIGNUM *p)
{
int retval,r1len,safe=0;
BIGNUM *r1,*c1,*tmp;

BN_CTX *ctx = BN_CTX_new();
r1 = BN_new();c1 = BN_new();tmp = BN_new();
r1len = kover2 - PHI_HIDDEN_PRIME_LEN - 64;
int checks = BN_prime_checks_for_size(kover2);
printf("Generating %d-bit divisor r_1 of q_1-1...\n",r1len);
for (;;)
   {
   BN_generate_prime(r1,r1len,safe,NULL,NULL,NULL,NULL);
   if (BN_is_bit_set(r1,r1len-2))
      break;
   }
PrintBigS("probable prime r_1 = ",r1);
BN_mul(q1,r1,p,ctx);
printf("Generating %d-bit probable prime q_1 = c_1*p*r_1 + 1...\n",
   kover2);
for (;;)
   {
   BN_rand(c1,64,1,0);
   BN_set_bit(c1,61);
   BN_clear_bit(c1,0);
   BN_mul(tmp,q1,c1,ctx);
   BN_add_word(tmp,1);
   if (BN_num_bits(tmp) == kover2)
      {
      retval = BN_is_prime_fasttest(tmp,checks,NULL,ctx,NULL,1);
      if (retval == 1)
         break;
      }
   }
```

```
PrintBigS("c_1 = ",c1);
BN_copy(q1,tmp);
BN_clear_free(r1);BN_clear_free(c1);BN_clear_free(tmp);
BN_CTX_free(ctx);
}
```

The security parameter `PHI_HIDDEN_PRIME_LEN` is set to be 128 and the security parameter `kover2` is set to be 512. For the moment, consider the problem of factoring m with the other security properties of TPIR aside. In this case we may assume that the phi-hidden value p is known without ambiguity to the cryptanalyst. This means that $(\frac{1}{4} + \delta)|q_1|$ bits of q_1 are publicly known (and hence known to the cryptanalyst). Here $|q_1|$ denotes the length in bits of q_1. The value δ is very small and takes into account the fixed upper and lower bits of c_1, etc.

We feel that this ratio provides strong enough protection against cryptanalysis at this time (but these security parameters are arguably cutting it *close*). There has been a significant amount of research on factoring composites when certain bits of the private key are known [5]. We remark that `PHI_HIDDEN_PRIME_LEN` must be large enough to avoid collisions in p.

Algorithm `GenerateQ2` generates a number r_2 that has a bit length of (`kover2` - `PHI_HIDDEN_PRIME_LEN` - 64). The 2 highest order bits of r_2 are 1. If Millier-Rabin indicates that r_2 is prime then r_2 is determined and the generation of q_2 continues. The number q_2 that is output will have r_2 as a factor of $q_2 - 1$.

`GenerateQ2` then randomly generates a candidate even factor c_2 that is `PHI_HIDDEN_PRIME_LEN`+64 bits long with the 3 highest order bits set to 1. If $q_2 = c_2 r_2 + 1$ is a `kover2`-bit number and Miller-Rabin indicates that it is prime then q_2 is returned by `GenerateQ2`. Otherwise another candidate for c_2 is generated, and so on.

6 The Database Algorithm

Recall that p is phi-hidden by $m = q_1 q_2$. Also, the user wants to retrieve the data string b_i corresponding to the tag t_i. So, (t_i, b_i) is an entry in the database B.

The principle behind TPIR2 is as follows. The array element $x[i]$ is an element in $\mathbb{Z}_m^*$. If the order of $x[i]$ is divisible by p then we *define* $x[i]$ to represent a binary 0. If the order of $x[i]$ is not divisible by p then we *define* $x[i]$ to represent a binary 1. When $x[i]$ represents a binary 0 it can be altered to represent a binary 1 by setting $x[i] \leftarrow x[i]^p \bmod m$.

So, the idea is to initially have all W values $x[0], x[1], ..., x[W-1]$ *represent* binary zeros. This will be the case with overwhelming probability by simply selecting these elements at random from $\mathbb{Z}_m^*$.

Consider the event that entry (t, b) of B is passed to the database algorithm. The database algorithm passes t to `RandPrime`. The output is a number p_c. For all $j \in \{0, 1, 2, ..., W-1\}$, if bit j of b is 1 then the database algorithm updates $x[j]$ by setting $x[j] \leftarrow x[j]^{p_c} \bmod m$. There are 2 possible cases:

Case 1. Suppose that $t \neq t_i$. Then with overwhelming probability $p_c \neq p$. Suppose that $p_c \neq p$. Then if bit j of b is 1 then the update operation will not remove p from the order of $x[j]$. So, the binary digit that $x[j]$ represents will not change.

Case 2. Suppose that $t = t_i$. Then with overwhelming probability $p_c = p$. Suppose that $p_c = p$. Then if bit j of b is 1 then the update operation will remove p from the order of $x[j]$ if p has not already been removed. So, $x[j]$ is certain to represent a binary 1 after the update operation.

So, as long as no (t_i, b_j) is supplied to the database algorithm where $b_j \neq b_i$, we would expect the database algorithm to be able to correctly retrieve b_i for the user where b_i is encoded in the response (i.e., the updated query). The next section explains how the user retrieves b_i from the response.

```
void DatabaseAlgorithm(querystruct *query, const char *tstr,
   const char *bstr)
{
unsigned char buff[NUM_DATA_BYTES],array[W];
BIGNUM *p,*m,*xval;

BN_CTX *ctx = BN_CTX_new();
p = BN_new();
RandPrime(p,tstr);
m = query->m;
int j=0;
for (;j<NUM_DATA_BYTES;j++)
   buff[j] = 0;
int len = strlen(bstr);
if (len > NUM_DATA_BYTES)
   len = NUM_DATA_BYTES;
strncpy(buff,bstr,len);
BitString2BitArray(array,buff);
printf("Let b[j] denote bit j of bstr = \"%s\".\n",bstr);
printf("During updating x[j] = x[j]^(p^b[j]) mod m is computed\n");
printf("for each j in which b[j] = 1. Here j ranges from 0\n");
```

```
printf("to %d.\n\n",W-1);
for (j=0;j<W;j++)
   {
   if (j % 8 == 0)
      printf("Updating byte %d.\n",(j>>3)+1);
   if (array[j])
      {
      xval = query->x[j];
      BN_mod_exp(xval,xval,p,m,ctx);
      }
   }
printf("\n");
BN_clear_free(p);
BN_CTX_free(ctx);
}
```

7 The Response Retriever

Recovering the data string b_i for database entry (t_i, b_i) amounts to investigating the orders of the integers $x[0], x[1], ..., x[W-1]$ that are included in the response that is sent to the user by the database administrator. The private key values (q_1, u) are used to analyze these orders.

More specifically, the response to the query is processed as follows. Consider integer $x[i]$ where $0 \leq i \leq W-1$. The integer $z = x[i]^u \bmod q_1$ is computed. The value v is set to 1 if z is 1 and is set to 0 otherwise. The value v is the binary digit that $x[i]$ represents.

Consider integer $x[i]$ where $0 \leq i \leq W-1$. In this explanation we will assume that the phi-hidden value p is prime (Miller-Rabin assures this with overwhelming probability). There are 2 possible cases.

Case 1. Suppose that $x[i]$ represents a binary 0. Then $x[i]^u \bmod q_1$ will still have an order that is divisible by p. So, in this case $z = x[i]^u \bmod q_1$ will not[4] be 1 and v will correctly be 0.

Case 2. Suppose that $x[i]$ represents a binary 1. Then $x[i]^u \bmod q_1 = 1$. This follows from Fermat's little theorem. So, in this case $z = x[i]^u \bmod q_1 = 1$ and v will correctly be 1.

[4]The chance that $x[i] \equiv 1 \bmod q_1$ in the response that the database administrator sends to the user is negligible.

```
void ResponseRetriever(unsigned char buff[NUM_DATA_BYTES],
   const querystruct *query,const csisprivkey *privkey)
{
unsigned char array[W];

BN_CTX *ctx = BN_CTX_new();
BIGNUM *tmp = BN_new();
BIGNUM *z = BN_new();
printf("Computing v = ((x[i] mod q_1)^u mod q_1) == 1)\n");
printf("for i = 0,1,...,%d. The == operator listed above ",W-1);
printf("is the ANSI C\nequality testing operator. So, v is 1\n");
printf("only when (x[i] mod q_1)^u mod q_1 is 1.\n");
BIGNUM *q1 = privkey->q1;
BIGNUM *u = privkey->u;
int i = 0;
for (;i<W;i++)
   {
   BN_mod(tmp,query->x[i],q1,ctx);
   BN_mod_exp(z,tmp,u,q1,ctx);
   int v = BN_is_one(z);
   array[i] = v;
   }
BitArray2BitString(buff,array);
BN_clear_free(tmp);BN_clear_free(z);
BN_CTX_free(ctx);
}
```

8 Conclusion

We presented an experimental implementation of our tagged private information retrieval protocol that we call TPIR2. Example output was provided and the core crypto, expressed in ANSI C, was given. Care was taken to position the protocol realistically within the context of a provable security argument vs. a heuristic security argument and we stated very clearly that TPIR2 falls into the latter category. We encourage readers to send feedback and bug reports to `feedback@cryptovirology.com`.

References

[1] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In

J. Stern, editor, *Advances in Cryptology—Eurocrypt '99*, pages 402–414. Springer-Verlag, 1999. Lecture Notes in Computer Science No. 1592.

[2] Benny Chor and Niv Gilboa. Computationally private information retrieval. In *Proceedings of the* 29*th ACM Symposium on Theory of Computing*, pages 304–313. ACM, 1997.

[3] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of the* 36*th IEEE Symposium on Foundations of Computer Science*, pages 304–313. IEEE, 1995.

[4] David A. Cooper and Kenneth P. Birman. Preserving privacy in a network of mobile computers. In *Proceedings of the* 16*th IEEE Symposium on Security and Privacy*, pages 26–38. IEEE, 1995.

[5] Don Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In Ueli Maurer, editor, *Advances in Cryptology—Eurocrypt '96*, pages 178–189. Springer, 1996. Lecture Notes in Computer Science No. 1233.

[6] Ivan Damgård, Peter Landrock, and Carl Pomerance. Average case error estimates for the strong probable prime test. *Mathematics of Computation*, 61(203):177–194, 1993.

[7] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the* 38*th IEEE Symposium on Foundations of Computer Science*, pages 364–373. IEEE, 1997.

[8] National Institute of Standards and Technology (NIST). FIPS Publication 180-2: Secure Hash Standard. *Federal Register*, August 1, 2002.

[9] Adam L. Young and Moti M. Yung. Deniable password snatching: On the possibility of evasive electronic espionage. In *Proceedings of the* 18*th IEEE Symposium on Security and Privacy*, pages 224–235. IEEE, May 1997.

[10] Adam L. Young and Moti M. Yung. *Malicious Cryptography: Exposing Cryptovirology*. Wiley, February 2004.