Crypto Dark Matter++

Abstract

Abstract goes here.

1 Introduction

1.1 Applications

2 Background

3 Related Work

4 New Protocol

5 Implementation

5.1 Parameters

The input variable to the PRF functions are the key K which is of size $m \times n$ and each input is a vector of size n. To save storage space, the key was implemented as a Toeplitz matrix, requiring $2 \cdot n$ bits of storage space. In phase 3 of the algorithm, a randomization matrix is used which is of size $r \times n = 81 \times 256$, resulting in entropy of 128 bits.

5.2 representing Z_2 vector

Bit slicing: this bit-wise packing technique was used to optimize the run-time, each 64 bits were represented by a word. Since each key is of size $m \times n$ and each input is a vactor of size n, it is possible to pack each 64 rows into m words. This may result in time saving of up to $\times 64$ of the run-time.

5.3 Representing Z_3 vector

To optimize the randomization in Z_3, which requires a matrix-vector multiplication in the last phase of the PRF calculation, we implement two optimization methods:

5.3.1 Bit slicing and operations:

this was done by representing a vector over Z_3 as two binary vectors - one LSB's and one MSB's.

A few operations, such as addition, subtraction and multiplication mod 3 required extra calculations. One of the properties that were utilized to implement this was:
mult-by-2 mod 3 $\leftrightarrow$ negation mod 3 $\leftrightarrow$ swap the MSB and LSB

Such a vector is sometimes being represented as a vector of $2 \times n$ bits. One example is when this vector is sent as an input to the Oblivious Transfer mechanism. The algorithms can be found in **????????**.

input: $\vec{l_1}, \vec{m_1}$ - LSB and MSB vectors of value 1
$\vec{l_2}, \vec{m_2}$ - LSB and MSB vectors of value 2 $\vec{select}$ - binary vector

Table 1: Operations in Z_3

Operations	**Methods**
Addition	$\vec{T} = (\vec{l_1} \mid \vec{m_2}) \oplus (\vec{l_2} \mid \vec{m_1});$ $MSB_{res} = (\vec{l_1} \mid \vec{l_2}) \oplus \vec{T};$ $LSB_{res} = (\vec{m_1} \mid \vec{m_2}) \oplus \vec{T};$
Subtraction	$\vec{T} = (\vec{l_1} \mid \vec{m_2}) \oplus (\vec{l_2} \mid \vec{m_1});$ $MSB_{res} = (\vec{l_1} \mid \vec{m_2}) \,\&\, \vec{T};$ $LSB_{res} = (\vec{m_1} \mid \vec{l_2}) \,\&\, \vec{T};$
Multiplication	$MS\vec{B}_{res} = ((\vec{l_1} \,\&\, (\vec{m_2}) \,\&\, ((\vec{m_1} \,\&\, (\vec{l_2});$ $LS\vec{B}_{res} = ((\vec{l_1} \,\&\, (\vec{l_2}) \,\&\, ((\vec{m_1} \,\&\, (\vec{m_2});$
MUX	$MS\vec{B}[i] = (\vec{m_2} \,\&\, s\vec{[i]}) \mid (\vec{m_1} \,\&\, -\vec{s}[i]);$ $LS\vec{B}[i] = (\vec{l_2} \,\&\, s\vec{[i]}) \mid (\vec{l_1} \,\&\, -\vec{s}[i]);$

5.3.2 Integer packing and operations:

this was explored as an alternative to the bit slicing. Since each number is a trinary number, the multiplication of each row by the output vector can be any number between -256 to 256. To explore this option, we represented each number by 9 bits, and packed 7 numbers into each word. This was expected to result in time saving of up to $\times 7$. Testing this option indeed proved to be significantly slower than the first method of packing. We therefore continued using the first packing method of packing instead. Note: Since each trinary number can be either 0,1 or 2, we can add up to 255 numbers and never exceed 511. Moreover, if we take a random sample, we can add 256 numbers and the probability of carryover is negligible. We use this to multiply a packed trinary vector with a ternary matrix with 256 columns (see algorithm below).

$x_0 \ldots x_6 \epsilon Z_3 = 0, 1, 2$
$\rightarrow x = \sum_{i=0}^{6} (512^i) \times c_i, x_i \epsilon [0 \ldots (2^(64) - 1)]$

Matrix multiplication - using a lookup table: List of things to be written

Comment 1: What is lookup table, explain with context?
Comment 2: Why is it required and what does it replace, are there any advantage
Comment 3: How does it work, please explain with a diagram, if possible
Comment 4: What are the disadvantage(s) of having/using a lookup table

Phase 2 of the protocols(DIDO, oPRF) outputs 256 bits $\in \mathbb{Z}_2$. Phase 3 takes output of phase 2 and performs a matrix vector multiplication with a public randomization matrix $Rmat \in \mathbb{Z}_3^{t\times m}$ to output $\mathbb{Z}_3^t$
Naive method of performing this matrix-vector multiplication will be to multiply and add individual bits, which will take a long time. An optimized and faster version was also implemented as a function using packed bits. This packed version of multiplication runs way faster than the naive implementation. In order to further optimize the run-time, a lookup table is used to perform the task. Implementing protocol using the lookup method consists of two substages in phase 3:

- **Preprocessing Stage:** Creation of lookup table
- **Multiplication Stage:** Usage of lookup table

Lookup table is a matrix whose elements are precomputed product value of packed inputs(output of phase 2) with all possible combinations of similar sized packed values. Suppose, if inputs are packed as a 8-bit word, the matrix will contain all possible combination of product of 8-bit input and all values from 0 to $2^8 - 1$.

The prime advantage of replacing matrix vector multiplication with lookup table is speed improvement. This is due to creation of lookup table at the preprocessing stage of the protocol. This means the lookup table can be created beforehand and during the protocol just use the
To this end, the randomization matrix is assumed to be constant and divided into 16 matrices of size 81×16. A lookup table of size 16×2^{256} is created during the pre-processing stage. During runtime, the input is divided to MSB and LSB, and each consecutive 16 bits (16 rows in the matrix) are used as separate input to the lookup table.
There are two challenges of using a lookup table instead of matrix-vector multiplication

- Huge storage requirement/ more time in preprocessing
- Find a perfect value that can balance between less storage and speedy lookup: Finding

5.4 Putting it all together

6 Analysis

Table 6 includes the computation and communication results for the different protocol.

Protocol	**Packed**	**Z_3 lookup table**	**Computation**	Communication Costs
Centralized	N	N		0
Centralized	Y	N		0
Original (DIDO) PRF protocol	Y	N		
Our (DIDO) PRF algorithm	Y	N		(4n + 2m)
Our (DIDO) PRF algorithm	Y	Y		(4n + 2m)
Our OPRF	Y	N		
Our OPRF	Y	Y		
Discrete log-based PRF	-	-		

Table 2: Run-time of different protocols

Protocol	**Rounds**	**Message Counts**	**Preprocessing Size**	**Comm Costs**
Old Protocol	3	4	0	
DIDO PRF protocol	Y	N		
OPRF	3	N		
Discrete log-based PRF	-	-		

Table 3: Communication Analysis of different protocols

7 Benchmarking

We compare our run-time to discrete-log based PRFs. To this end, we use the lib sodium library [2]. The library uses elliptic curve 252 bits, and includes a function that performs scalar multiplication ('crypto_scalarmult_ed25519').

8 Experimental Results

The system was tested using Ubuntu Server 18.04 on a t2.medium AWS environment. To record the timings, the code was run in a loop 1000 times. Below are run-time results for running a single instance of PRF in microsecond. The results include both centralized and distributed versions of the PRF. In order to increase efficiency, packing and lookup tables were used. The packing indicates both the Z_2 and Z_3 packing.

9 Conclusion and Future Work

10 Acknowledgement

11 References

References

[1] Dan Boneh et al. *Exploring Crypto Dark Matter: New Simple PRF Candidates and Their Applications*. Cryptology ePrint Archive, Report 2018/1218. `https://eprint.iacr.org/2018/1218`. 2018.

Protocol	**Packed**	Z_3 **lookup table**	**Rounds/sec**	Runtime(μ sec)
Centralized weak PRF	N	N	50K	20.2
Centralized weak PRF	Y	N	65.4K	18.5
Centralized weak PRF	Y	Y	165K	6.08
Original distributed dark matter	Y	N	24K	40.56
DIDO	Y	N	49K	20.20
DIDO	Y	Y	82K	12.12
oPRF	Y	N	53K	18.66
oPRF	Y	Y	104K	9.52
Discrete log-based PRF	-	-	35K	28.69

Table 4: Run-time of different protocols

[2] *libsodium 1.0.18-stable*. `https://libsodium.gitbook.io/doc/`. Online; December 31 2020. 2020.