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.

The Addition, subtraction, multiplication and MUX mod 3 were implemented using the protocols defined in table reftab:multicol. 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: $\vec{l_1}, m_1$ - LSB and MSB vectors of value 1
l_2, m_2 - LSB and MSB vectors of value 2 s - binary select vector

Table 1: Operations in Z_3

Operations	**Methods**
Addition	$t := (l_1 \wedge m_2) \oplus (l_2 \wedge m_1)$ $m_{\text{out}} := (l_1 \wedge l_2) \oplus t$ $l_{\text{out}} := m_1 \wedge m_2) \oplus t$
Subtraction	$t := (l_1 \wedge l_2) \oplus (m_2 \wedge m_1)$; $m_{\text{out}} := (l_1 \wedge m_2) \oplus t$; $l_{\text{out}} := (m_1 \wedge l_2) \oplus t$;
Multiplication	$m_{\text{out}} := (l_1 \vee m_2) \wedge (m_1 \vee l_2)$; $l_{\text{out}} := (l_1 \vee l_2) \wedge (m_1 \vee m_2)$;
MUX	$m_{\text{out}} := (m_2 \vee s) \wedge (m_1 \vee (-s))$; $l_{\text{out}}[i] := (l_2 \vee s) \wedge (l_1 \vee (-s))$;

5.3.2 Integer packing and operations:

As an alternative to bit splicing, integer packing was explored. 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).

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

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

Phase 2 of the protocols(DIDO, oPRF) outputs m 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.

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 $2^8 - 1$.

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

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. While creation of a lookup table takes a prolong period of time, the multiplication stage is as simple as accessing an element from a matrix. This speeds up the last stage of the protocol, hence significantly improving the overall speed of execution.

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^{16} 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. Similarly, we explored dividing the table into 8 matrices of size 81×32, resulting in a lookup table of size 8×2^{32}. However, due to the large table size, this crashed the system during runtime and we were not able to get results for it.

There are two challenges of using a lookup table as compared to 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

- **Implemented DIDO protocol was 50.7% faster and oPRF was 54% than implemented wPRF protocol described in TCC'18.**
- **Using lookup table, the speed increase was significant at 70% and 76.5% for DIDO and oPRF respectively.**

6 Analysis

Table 2 includes the computation and communication results for the different protocol.

Protocol	Round Complexity	Online Communication	Preprocessing Size	Computation Operation
Central wPRF	-	-	-	27K
Central wPRF (packed)	-	-	-	18K
Centralized wPRF(lookup)	-	-	-	9K
Fully Distributed[1] (packed)	4	3490	2K	45K
Fully Distributed (packed)	3	2210	6K	22K
Fully Distributed (lookup)	3	2210	1M	13K
OPRF (packed)	3	1361	3K	20K
OPRF (lookup)	3	1361	1M	11K
~~**Discrete log-based PRF**~~	-	-	-	-

Table 2: 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.

Protocol	**Packed**	**lookup**	**Runs/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

Table 3: Comparison of the run-time and computation of the different Optimized Centralized wPRF.

Protocol	**Runs/sec**	**Runtime(μ sec)**
Fully Distributed wPRF[1]	24K	40.56
Fully Distributed wPRF	49K	20.20
Fully Distributed wPRF(lookup)	82K	12.12
oPRF	53K	18.66
oPRF with lookup	104K	9.52
Discrete log-based PRF	35K	~~28.69~~ **9.56**

Table 4: Optimized protocols runtime with parameters.

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.

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