

# Crypto Dark Matter++

## Abstract

Abstract goes here.

## 1 Introduction

### 1.1 Applications

## 2 Background

## 3 Related Work

## 4 Technical Overview of Results

### 4.1 Protocols Implemented

In this work we implemented a few distributed protocols of the PRF introduced by Boneh et. Al [1]:

$$PRF_k(x) = R \times (K \times x \bmod 2) \bmod 3$$

where  $R$  is a randomization matrix in  $Z_3$ ,  $K$  is an  $m \times n$  key and  $x$  is an  $n$ -length vector. Specifically, we present a new protocol for implementing this PRF. The protocol uses a shared input and output which provides improved performance over the original protocol described in [1]. For context, we also implement the original protocol [1]. We also introduce an improved OPRF (Oblivious PRF) protocol, in which one party has the input and one has the key. Below are details of the protocols.

#### 4.1.1 PRF

**notations:** below are the notations we use to describe the protocols:

$K$  is the key, represented by an  $m \times n$  matrix

$x$  is the input, represented by an  $n$  size vector

$R_x, r_k$  - pre-shared randomness for key  $K$  and input  $x$

$r$  - random vector in  $Z_3$ ] We use capital letters for matrices, and  $\vec{x}$  notation for vectors

**Original Distributed Input Distributed Output (DIDO) PRF Protocol** The protocol was introduced in [1]. The main details of the protocols are described below (1). In this setting, both the input and the key are secret shared additively between the two parties. Both the protocol presented in this paper and the original protocol are two-party PRFs. Example can be shown in Figure 1

## wPRF(TCC'18)

- **Input:**
  - Private shares of Key matrix  $K \in \mathbb{Z}_2^{m \times n}$  and input vector  $x \in \mathbb{Z}_2^n$
  - Public Randomization Matrix  $R_{mat} \in \mathbb{Z}_3^{l \times m}$
- **Output:**  $((K \times x) \times R_{mat}) \in \mathbb{Z}_3^l$
- **Phase 1: Parties compute the product  $K \times x$** 
  - Parties mask their inputs and key in order to compute the term  $K_1x_2 + K_2x_1$
  - Parties locally compute their shares of input and key simultaneously,  $K_1x_1$
- **Phase 2: Share Conversion**
  - Convert the shares computed by both parties to  $\mathbb{Z}_3$
- **Phase 3: Randomization Matrix**
  - Multiply the output of both parties in Phase 2 with public randomization matrix  $R_{mat}$

Figure 1: Old wPRF protocol

## wPRF (Shared Input and Shared Key)

- **Input:**
  - Parties hold private shares of Key matrix  $K \in \mathbb{Z}_2^{m \times n}$  and input vector  $x \in \mathbb{Z}_2^n$
  - Public Randomization Matrix  $R_{mat} \in \mathbb{Z}_3^{l \times m}$
- **Output:**  $((K \times x) \times R_{mat}) \in \mathbb{Z}_3^l$
- **Preprocessing:** Generating correlated randomness( $r_x, s_w \in \mathbb{Z}_2^n$  and  $r_K \in \mathbb{Z}_2^{m \times n}$ ) used in masking the inputs.
- **Phase 1: Parties masks their input and share it with each other**
  - Parties mask their inputs and key such that  $x' = x + r_x$  and  $K' = K + r_K$
- **Phase 2: Compute  $w' = Kx + r_w$** 
  - Using the shares received in phase 1, each party i computes their share of  $K_i x_i + r_{w_i}$  where  $r_{w_i} = r_{K_i} * r_{x_i} + s_{w_i}$
- **Phase 3: Randomization Matrix**
  - Parties perform product of their shares and public randomization matrix  $R_{mat}$
  - These product are shared among the parties and final output is obtained by adding the shares

Figure 2: Distributed Input Distributed Output wPRF protocol  
Our improved (DIDO) PRF protocol

---

**Algorithm 1** 2-Party dark matter PRF

---

Input:  $K_{m \times n}(i)$  and  $Inp(i_n)$  key and user input of each user,  
( $i = 1 \dots 2, n, m = 256$ )

Output: user 1:  $\vec{K} \times x + \vec{r}$

user2:  $-\vec{r}$

**Preprocessing:**

Output: user 1:  $\vec{R}_a, \vec{r}_b$  - pre-shared randomness

user 2:  $\vec{r}_x$  - pre-share randomness

**Stage 1:** calculate  $a \times b + c$

input: user 1:  $\vec{A}, \vec{b}, \vec{R}_a, \vec{r}_b$

user 2:  $\vec{x}, \vec{r}_x, \vec{z}$

1. user 2  $\vec{m}_x = \vec{x} - \vec{r}_x \rightarrow$  user 1
2. user 1  $\leftarrow \vec{M}_a = \vec{A} - \vec{r}_A$  user 2
3. user 2  $\vec{m}_b = \vec{R}_a \times \vec{m}_x + \vec{b} - \vec{r}_b \rightarrow$  user 1

Output: user1 :  $-\vec{b}$

user2 :  $\vec{M}_a \times \vec{x} + \vec{m}_b + \vec{z} = \vec{A}\vec{x} + \vec{b}$

**Stage 2:** Oblivious Transfer input: user 1:  $\vec{r}_1, \vec{r}_2, \vec{r}_a, \vec{r}_b$  - vectors in  $Z_3$  user 2:  $\vec{x}, \vec{r}_x$  - vectors in  $Z_2$ ,  $\vec{z}$  - vector in  $Z_3$

1. user 1  $\leftarrow \vec{m}_x = \vec{x} \oplus \vec{r}_x$  user 2
2. user 1:  $\vec{m}_1 = (-m_x)\vec{r}_a + \vec{m}_x\vec{r}_b + \vec{r}_1 \rightarrow$  user 2
3. user 1:  $\vec{m}_2 = (m_x)\vec{r}_a + -\vec{m}_x\vec{r}_b + \vec{r}_2 \rightarrow$  user 2

output: user 2:  $\vec{w} := \vec{x} \times \vec{m}_2 + -\vec{x} \times \vec{m}_1 - \vec{z}$

**stage 3:**  $Z_3$  randomization

---

---

**Algorithm 2** 2-Party Distributed PRF (Shared input and Shared Key)

---

Input:  $x \in \mathbb{Z}_2^n$

Key:  $K \in \mathbb{Z}_2^{m \times n}$

Output:  $y \in \mathbb{Z}_3^t$

Preprocessing: Generate correlated randomness  $r_x, s_w \in \mathbb{Z}_2^n$ . and  $R_k \in \mathbb{Z}_2^{m \times n}$

and compute  $r_w = R_k \cdot r_x \oplus s_w$

Stage 1: Mask the inputs: Both the parties mask their shares of input and key they hold and the mask are shared.

$K'_i = K_i + Rk_i$  and  $x'_i = x_i + rx_i$  and

Stage 2: Merging the shares, each party computes  $w' = K \times x + rw$ . This is done as follows

stage 3:  $Z_3$  randomization

---

## 4.2 OPRF

This is a new OPRF protocol, which improved the performance over the original OPRF protocol described in [1]. In this setting, one party has the input and another party has the key.

### oPRF

- **Input:**
  - Server holds Key  $K \in \mathbb{Z}_2^{m \times n}$  while client holds input  $x \in \mathbb{Z}_2^n$
  - Public Randomization Matrix  $R_{mat} \in \mathbb{Z}_3^{t \times m}$
- **Output:**  $((K \times x) \times R_{mat}) \in \mathbb{Z}_3^t$
- **Preprocessing:**
  - Generating correlated randomness  $(r_q, r_w \in \mathbb{Z}_2^n \text{ and } r_k \in \mathbb{Z}_2^{m \times n})$  and used in masking the inputs.
  - Convert  $r_w$  to 0/1 value in  $\mathbb{Z}_3$
- **Phase 1: Masking the inputs**
  - Server mask the key and share it with client  $K' = K + r_k$
  - Meanwhile, client mask the input and share it with server  $x' = x + r_x$
- **Phase 2: Compute  $w' = Kx + r_w$** 
  - Both Server and client computes shares of  $w'$  which is later shared and combined by both.
- **Phase 3: Client computes the output  $F_K(x)$** 
  - Both server and client computes product of their shares and public randomization matrix  $R_{mat}$
  - Server sends its share to client and client computes the output

Figure 3: oPRF protocol

---

#### Algorithm 3 2-Party oPRF

---

Preprocessing  
Stage 1: calculate  $a \times b + c$   
Stage 2: Oblivious Transfer  
stage 3:  $\mathbb{Z}_3$  randomization

---

## 4.3 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.

## 4.4 Optimizations

### 4.4.1 packing in $\mathbb{Z}_2$

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 vector 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.

#### 4.4.2 Packing in $Z_3$

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:

**Bit slicing:** 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:  
 $\text{mult-by-2 mod 3} \leftrightarrow \text{negation mod 3} \leftrightarrow \text{swap the MSB and LSB}$

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 Algorithms 4 to 7.

---

##### Algorithm 4 Addition in $Z_3$

---

Item 1: 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

1.  $\vec{T} = (\vec{l}_1 \vee \vec{m}_2) \oplus (\vec{l}_2 | \vec{m}_1);$
  2.  $MSB_{res} = (\vec{l}_1 \vee \vec{l}_2) \oplus \vec{T};$
  3.  $LSB_{res} = (\vec{m}_1 \vee \vec{m}_2) \oplus \vec{T};$
- 

---

##### Algorithm 5 Subtraction in $Z_3$

---

Item 2:  
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

1.  $\vec{T} = (\vec{l}_1 | \vec{l}_2) \wedge (\vec{m}_2 | \vec{m}_1);$
  2.  $MSB_{res} = (\vec{l}_1 | \vec{m}_2) \wedge \vec{T};$
  3.  $LSB_{res} = (\vec{m}_1 | \vec{l}_2) \wedge \vec{T};$
- 

**Integer packing:** 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 510. Moreover, if we take a random sample,

---

**Algorithm 6** Multiplication of two trinary vectors  $Z_3$ 

---

Item 3:

input:  $\vec{l}_1, \vec{m}_1$  - LSB and LSB vectors of value 1

$\vec{l}_2, \vec{m}_2$  - LSB and LSB vectors of value 2

1.  $LS\vec{B}_{res} = ((\vec{l}_1 \wedge \vec{l}_2) \wedge ((\vec{m}_1 \wedge \vec{m}_2))$
  2.  $MS\vec{B}_{res} = ((\vec{l}_1 \wedge \vec{m}_2) \wedge ((\vec{m}_1 \wedge \vec{l}_2))$
- 

---

**Algorithm 7** Mux implementation of two trinary vectors  $Z_3$ 

---

input:  $\vec{l}_1, \vec{m}_1$  - LSB and LSB vectors of value 1

$\vec{l}_2, \vec{m}_2$  - LSB and LSB vectors of value 2  $\vec{select}$  - binary vector

1. for each word  $i$
  2.  $LS\vec{B}[i] = (\vec{l}_2 \wedge s[\vec{i}] | (\vec{l}_1 \wedge -s[\vec{i}]))$
  3.  $MS\vec{B}[i] = (\vec{m}_2 \wedge s[\vec{i}] | (\vec{m}_1 \wedge -s[\vec{i}]))$
- 

we can add 256 numbers and the probability of carryover is negligible. We use this to multiply a packed trinary vector with a trinary matrix with 256 columns (see algorithm below).

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

**Using a lookup table:** To further optimize the run-time, we created a lookup table to replace the matrix-vector multiplication. To this end, the randomization matrix is assumed to be constant and divided into 16 matrices of size  $81 \times 16$ . A lookup table of size  $16 \times 2^{256}$  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.

## 5 Analysis

Table 5 includes the computation and communication results for the different protocol.

## 6 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').

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 1: Run-time of different protocols

## 7 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	$Z_3$ lookup table	Rounds/sec	Runtime( $\mu$ 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 2: Run-time of different protocols

## 8 Conclusion and Future Work

## 9 Acknowledgement

## 10 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.