Crypto Dark Matter++

Abstract

Abstract goes here.

1 Introduction

1.1 Applications

2 Background

3 Related Work

4 New Protocol

5 New Protocol - Additional Details

6 Preliminaries

We begin by defining some basic notations that we will use throughout this work. We use uppercase letters (e.g., A, B) to denote matrices and bold letters to denote vectors **a** . As was specified in the Boneh et al work [1], the input variables to the PRF functions are the key K which is of size $m \times n$ and each input vector **i** iof 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 R is used which is of size $t \times m = 81 \times 256$, resulting in entropy of 128 bits (as $81 = 128 \times log_2 3$).

7 Implementation

The pre-shared randomness was generated locally (as if its generated by a third trusted party). Only the online part was implemented in a distributed manner. Only the online part was implemented in a distributed manner.

7.1 representing Z_2 vector

The dark matter protocol manipulates elements of Z_2 and Z_3. However, machine words can hold up to 64 bits and we use this to represent many elements in just a few machine words, using bit slicing. Namely, we used each machine word as a vector of 64 bits and applied operations to this bit-vector in a SIMD manner. Since machine words can hold up to 64 bits, we further enhance the code by

implementing software bit-slicing optimizations. In this method, each 64 bits are represented by a word. Since each key $\mathbf{K}$ is of size $m \times n$ and each input is a vector of size n, this results in packing each key o $m/64 = 4$ words and each input vector into 4 words. This may result in time saving of up to $\times 64$ of the run-time.

7.2 Representing Z_3 vector

The last part of the dark matter function takes an intermediate vector viewed as a vector of Z_3 elements, which is then multiplied by a matrix R in Z_3 to produce an output vector in Z_3. We tried different methods for implementing this matrix-vector multiplication over Z_3.:

7.2.1 Bit slicing

Bit slicing is implemented by representing a vector over Z_3 as two binary vectors - one LSB's and one MSB's. The operations over this Z_3 vector - addition, subtraction, multiplication and MUX - were implemented in a bitwise manner as shown in table 1. We took advantage of the fact that when representing each vector as MSB and LSB, negation (which is the same as multiplying by two) is the same as swapping the most and least significant bits.

Table 1: Operations in Z_3. input: $\vec{l_1}, m_1$ - LSB and MSB vectors of first trinary number, l_2, m_2 - LSB and MSB vectors of second trinary number, s - binary select vector

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	$m_{\text{out}} := (m_2 \vee s) \wedge (m_1 \vee (-s))$; $l_{\text{out}}[i] := (l_2 \vee s) \wedge (l_1 \vee (-s))$;

7.2.2 Matrix multiplication - using a lookup table

Even with bitslicing, implementing the matrix-vector multiplication column by column via the opertions from Table 1 is still rather slow. To get better performence, we capitalized on the fact that the random matrix R is fixed, and we can therefore set up R-dependent tables and use table lookup in the implementation of the multiply-by-R operation.

Specifically, we partition the matrix R (which has $m = 256$ columns) into sixteen slices of 16 columns each, denoted $R_1, \ldots, R_{16}$. For each of these small matrices R_i, we then built a table with 2^{16} entries, holding the result of multiplying R_i by each vector from $\{0,1\}^{16}$. Namely, let $R_{i,0}, R_{i,1}, \ldots, R_{i,15}$ be the sixteen columns of the matrix R_i. The table for R_i is then defined as

follows: For each index $0 \le j < 2^{16}$, let $\vec{J} = (j_0, j_1, \ldots, j_{15})$ be the 0-1 vector holding the bits in the binary representation of j, then we have

$$T_i[j] = R_i \times \vec{J} = \sum_{k=0}^{15} R_{i,k} \cdot j_k \mod 3.$$

Recalling that R is Z_3 matrix of dimension 81×256, every entry in each table T_i therefore holds an 81-vector over Z_3. Specifically, it holds a packed-Z_3 element with four words (two for the MSBs and two for the LSBs of this 81-element vector).

Note, however, that T_i can only be used directly to multiply R_i by 0-1 vectors. To use T_i when multiplying R_i by a $\{0,1,2\}$ vector, multiply R_i separately by the MBSs and the LSBs of that vector, and then subtract one from the other using the operations from Table 1. To multiply a dimension-256 $\{0,1,2\}$ vector by the matrix R, we partition it into sixteen vectors of dimension 16, use the approach above to multiply each one of them by the corresponding R_i, then use the operations from Table 1 to add them all together. Hence we have

Multiply-by-R(input: $\vec{V} \in \{0,1,2\}^{256}$:

1. $Acc := 0$
2. For $i = 0, \ldots, 15$
3. Let $\vec{M}_i, \vec{L}_i \in \{0,1\}^{16}$ be the MSB, LSB vectors (resp.) of $\vec{V}_i = \vec{V}[16i, \ldots, 16i+15]$
4. Set the indexes $m := \sum_{k=0}^{15} 2^i \cdot \vec{M}[i]$ and $\ell := \sum_{k=0}^{15} 2^i \cdot \vec{L}[i]$
5. $Acc := Acc + T_i[\ell] - T_i[\ell] \mod 3$
6. Output Acc

We note that a sligtly faster implementation could be obtained by braking the matrix into (say) 26 slices of upto 10 columns each, and directly identify each 10-vector over $\{0,1,2\}$ with an index $i < 3^{10} = 59049$. This implementation would have only 26 lookup operations instead of 32 in the implementation above, so we expect it to be about 20% faster. On the other hand it would have almost twice the table size of the implementation from above.

8 Analysis

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

Protocol	Round Complexity	Online Communication	Preprocessing Size	Computation Operation
Centralized WPRF	-	-	-	9K
Distributed WPRF [1]	4	5	2K	45K
Our)WPRF	3	5	1M	13K
Our OPRF	3	5	1M	11K

Table 2: Communication Analysis of different protocols. The protocols were optimized using both the bit packing and the lookup table

9 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').

10 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	82K	12.12
OPRF with lookup	104K	9.52
Discrete log-based PRF	12K	**86.07**

Table 4: Optimized protocols runtime. These protocols utilize both the bit-packing and lookup table optimization

11 Conclusion and Future Work

12 Acknowledgement

13 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.