ROTed: Random Oblivious Transfer for embedded devices

Manuel Goulão[1]
(mgoulao@math.tecnico.ulisboa.pt)

with P. Branco[1], L. Fiolhais[2], P. Martins[2], P. Mateus[1], L. Sousa[2]

CHES 2021

14 September 2021

[1]Instituto de Telecomunicações, Instituto Superior Técnico, Universidade de Lisboa
[2]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Introduction

Motivation

- We aim to design a highly efficient Random Oblivious Transfer (ROT) protocol.
- OT extensions use ROT as base OT (malicious setting).
- Most related art has focused on the design of 1-out-of-2 OTs.
- Directly using ROT improves efficiency without needing a black-box conversion.
- Applications in embedded systems, IoT, desktops, servers...

Introduction

Motivation

- We aim to design a highly efficient Random Oblivious Transfer (ROT) protocol.
- OT extensions use ROT as base OT (malicious setting).
- Most related art has focused on the design of 1-out-of-2 OTs.
- Directly using ROT improves efficiency without needing a black-box conversion.
- Applications in embedded systems, IoT, desktops, servers...

Introduction

Motivation

- We aim to design a highly efficient Random Oblivious Transfer (ROT) protocol.
- OT extensions use ROT as base OT (malicious setting).
- Most related art has focused on the design of 1-out-of-2 OTs.
- Directly using ROT improves efficiency without needing a black-box conversion.
- Applications in embedded systems, IoT, desktops, servers...

Introduction

Motivation

- We aim to design a highly efficient Random Oblivious Transfer (ROT) protocol.
- OT extensions use ROT as base OT (malicious setting).
- Most related art has focused on the design of 1-out-of-2 OTs.
- Directly using ROT improves efficiency without needing a black-box conversion.
- Applications in embedded systems, IoT, desktops, servers...

Introduction

Motivation

- We aim to design a highly efficient Random Oblivious Transfer (ROT) protocol.
- OT extensions use ROT as base OT (malicious setting).
- Most related art has focused on the design of 1-out-of-2 OTs.
- Directly using ROT improves efficiency without needing a black-box conversion.
- Applications in embedded systems, IoT, desktops, servers...

Introduction

Overview

- ROT protocol
  - Novel 3-round protocol based on the RLWE assumption, proved secure in the UC framework (in the ROM) in the presence of malicious adversaries.
- ROT Implementation
  - Implemented in C++ and executed in an x86 server-class processor and in four ARM application-class processors, benchmarked against the current state-of-the-art, achieving speeds at least one order of magnitude faster.
- PSI use-case
  - Evaluate the impact of the proposed protocol inside a real-world application: an open-source PSI framework [PRTY, EUROCRYPT 2020], managing to get speedups of up to 6.6 times in relation to the related art.

Introduction

Overview

- ROT protocol
  - Novel 3-round protocol based on the RLWE assumption, proved secure in the UC framework (in the ROM) in the presence of malicious adversaries.
- ROT Implementation
  - Implemented in C++ and executed in an x86 server-class processor and in four ARM application-class processors, benchmarked against the current state-of-the-art, achieving speeds at least one order of magnitude faster.
- PSI use-case
  - Evaluate the impact of the proposed protocol inside a real-world application: an open-source PSI framework [PRTY, EUROCRYPT 2020], managing to get speedups of up to 6.6 times in relation to the related art.

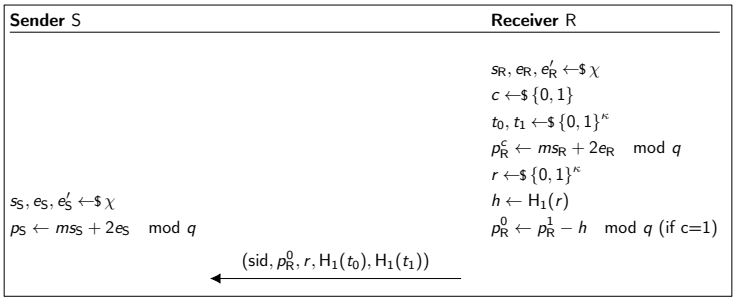
Introduction

Overview

- ROT protocol
  - Novel 3-round protocol based on the RLWE assumption, proved secure in the UC framework (in the ROM) in the presence of malicious adversaries.
- ROT Implementation
  - Implemented in C++ and executed in an x86 server-class processor and in four ARM application-class processors, benchmarked against the current state-of-the-art, achieving speeds at least one order of magnitude faster.
- PSI use-case
  - Evaluate the impact of the proposed protocol inside a real-world application: an open-source PSI framework [PRTY, EUROCRYPT 2020], managing to get speedups of up to 6.6 times in relation to the related art.

Protocol

Construction (1st round)



Protocol

Construction (2nd round)

Sender S	**Receiver** R
$h' \leftarrow \mathsf{H}_1(r)$	
$p_{\mathsf{R}}^1 \leftarrow p_{\mathsf{R}}^0 + h' \mod q$	
for $i \in \{0,1\}$:	
$k_{\mathsf{S}}^i \leftarrow s_{\mathsf{S}} p_{\mathsf{R}}^i + 2e'_{\mathsf{S}} \mod q$	
$\sigma_i \leftarrow \mathsf{Sig}(k_{\mathsf{S}}^i)$	
$\mathsf{sk}_{\mathsf{S}}^i \leftarrow \mathsf{Mod}_2(k_{\mathsf{S}}^i, \sigma_i)$	
$K_i \leftarrow \mathsf{H}_2(\mathsf{sk}_{\mathsf{S}}^i)$	
$a \leftarrow\$ \{0,1\}$	
$u \leftarrow\$ \{0,1\}^\kappa$	
$\xrightarrow{(\mathsf{sid}, p_{\mathsf{S}}, K_a, K_{1-a}, \sigma_0, \sigma_1, u)}$	

Protocol

Construction (3rd round)

Sender S		**Receiver** R
		$k_R \leftarrow s_R p_S + 2e'_R \mod q$ $\mathsf{sk}_R \leftarrow \mathsf{Mod}_2(k_R, \sigma_c)$ $K_c \leftarrow \mathsf{H}_2(\mathsf{sk}_R)$ $b = 0$ if $K_a = K_c$ $b = 1$ if $K_{1-a} = K_c$ $b = c$ if $K_{1-a} \neq K_c \wedge K_a \neq K_c$
	(sid, t_0, t_1) ←	
Abort if t_0, t_1 don't match $\mathsf{H}_1(t_0), \mathsf{H}_1(t_1)$		
output: $(M_0 = \mathsf{H}_2(\mathsf{sk}_S^a + t_a + u),$ $M_1 = \mathsf{H}_2(\mathsf{sk}_S^{1-a} + t_{1-a} + u))$		**output:** $(b, M_b = \mathsf{H}_2(\mathsf{sk}_R + t_c + u))$

Security

Against a corrupted Sender

- **Cannot learn the bit** b because while it holds two KE messages from the receiver (p_R^0 and p_R^1), only one of these was generated as an RLWE sample (p_R^c).

  The "other message" (p_R^{1-c}) is coerced to be a uniform random element by summing or subtracting a random value obtained from the RO, and distinguishing the two yields the bit b but means breaking the RLWE assumption.

$$p_R^c \leftarrow m s_R + 2e_R \mod q$$
$$p_R^0 \leftarrow p_R^1 - h \mod q \quad \text{(if c=1)}$$

- **Cannot bias the distribution of the messages** as the messages come from the RO and their query must include random nonces sent by the honest Receiver (t_0, t_1).

$$M_0 \leftarrow \mathrm{H}_2(\mathrm{sk}_S^a + t_a + u)$$
$$M_1 \leftarrow \mathrm{H}_2(\mathrm{sk}_S^{1-a} + t_{1-a} + u)$$

Security

Against a corrupted Sender

- **Cannot learn the bit** b because while it holds two KE messages from the receiver (p_R^0 and p_R^1), only one of these was generated as an RLWE sample (p_R^c).

  The "other message" (p_R^{1-c}) is coerced to be a uniform random element by summing or subtracting a random value obtained from the RO, and distinguishing the two yields the bit b but means breaking the RLWE assumption.

$$p_R^c \leftarrow ms_R + 2e_R \mod q$$
$$p_R^0 \leftarrow p_R^1 - h \mod q \quad \text{(if c=1)}$$

- **Cannot bias the distribution of the messages** as the messages come from the RO and their query must include random nonces sent by the honest Receiver (t_0, t_1).

$$M_0 \leftarrow \mathrm{H}_2(\mathrm{sk}_S^a + t_a + u)$$
$$M_1 \leftarrow \mathrm{H}_2(\mathrm{sk}_S^{1-a} + t_{1-a} + u)$$

Security

Against a corrupted Receiver

- **Cannot learn both messages**, as computing each message requires computing a shared key with the sender for p_R^0 and p_R^1.

  Only p_R^c was generated as an RLWE sample, p_R^{1-c} is uniform (as before). So **only the shared key corresponding to p_R^c may be computed** and output, otherwise the adversary needs to break the RLWE assumption and find the secrets for p_R^{1-c}.

- **Cannot bias the distribution of the output bit (b)** as it includes a random bit chosen by the honest Sender (a).

$$b \leftarrow 0 \text{ if } K_a = K_c$$
$$b \leftarrow 1 \text{ if } K_{1-a} = K_c$$

- **Cannot bias the distribution of the output message (M_b)** as it comes from the RO and its query must include a random nonce sent by the honest Sender (u).

$$M_b \leftarrow \mathrm{H}_2(\mathrm{sk}_R + t_c + u)$$

Security

Against a corrupted Receiver

- **Cannot learn both messages**, as computing each message requires computing a shared key with the sender for p_R^0 and p_R^1.

  Only p_R^c was generated as an RLWE sample, p_R^{1-c} is uniform (as before). So **only the shared key corresponding to p_R^c may be computed** and output, otherwise the adversary needs to break the RLWE assumption and find the secrets for p_R^{1-c}.

- **Cannot bias the distribution of the output bit (b)** as it includes a random bit chosen by the honest Sender (a).

$$b \leftarrow 0 \text{ if } K_a = K_c$$
$$b \leftarrow 1 \text{ if } K_{1-a} = K_c$$

- **Cannot bias the distribution of the output message (M_b)** as it comes from the RO and its query must include a random nonce sent by the honest Sender (u).

$$M_b \leftarrow \mathrm{H}_2(\mathrm{sk}_R + t_c + u)$$

Security

Against a corrupted Receiver

- **Cannot learn both messages**, as computing each message requires computing a shared key with the sender for p_R^0 and p_R^1.

  Only p_R^c was generated as an RLWE sample, p_R^{1-c} is uniform (as before). So **only the shared key corresponding to p_R^c may be computed** and output, otherwise the adversary needs to break the RLWE assumption and find the secrets for p_R^{1-c}.

- **Cannot bias the distribution of the output bit (b)** as it includes a random bit chosen by the honest Sender (a).

$$b \leftarrow 0 \text{ if } K_a = K_c$$
$$b \leftarrow 1 \text{ if } K_{1-a} = K_c$$

- **Cannot bias the distribution of the output message (M_b)** as it comes from the RO and its query must include a random nonce sent by the honest Sender (u).

$$M_b \leftarrow \mathrm{H}_2(\mathrm{sk}_R + t_c + u)$$

UC simulation

Corrupted Sender

- **Program H_1 such that it is able to recover both keys** sk_S^i obtained by the Sender (this is indistinguishable from uniform output — RLWE assumption).

$$\mathsf{H}_1(r) = p_\mathsf{R}^1 - p_\mathsf{R}^0$$

- With both keys, **extract the value a from the malicious Sender** (if a is not found, 3rd condition safeguards indistinguishability).

$$\mathsf{H}_2(\mathsf{sk}_\mathsf{R}^0) = K_a \wedge \mathsf{H}_2(\mathsf{sk}_\mathsf{R}^1) = K_{1-a} \implies a = 0$$
$$\mathsf{H}_2(\mathsf{sk}_\mathsf{R}^1) = K_a \wedge \mathsf{H}_2(\mathsf{sk}_\mathsf{R}^0) = K_{1-a} \implies a = 1$$

- **Program H_2 to output the right messages** which it received from the ideal functionality to their respective queries, which it can now compute.

$$M_0 \leftarrow \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^a + t_a + u)$$
$$M_1 \leftarrow \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^{1-a} + t_{1-a} + u)$$

UC simulation

Corrupted Sender

- **Program H_1 such that it is able to recover both keys** sk_S^i obtained by the Sender (this is indistinguishable from uniform output — RLWE assumption).

$$\mathsf{H}_1(r) = p_\mathsf{R}^1 - p_\mathsf{R}^0$$

- With both keys, **extract the value a from the malicious Sender** (if a is not found, 3rd condition safeguards indistinguishability).

$$\mathsf{H}_2(\mathsf{sk}_\mathsf{R}^0) = K_a \wedge \mathsf{H}_2(\mathsf{sk}_\mathsf{R}^1) = K_{1-a} \implies a = 0$$
$$\mathsf{H}_2(\mathsf{sk}_\mathsf{R}^1) = K_a \wedge \mathsf{H}_2(\mathsf{sk}_\mathsf{R}^0) = K_{1-a} \implies a = 1$$

- **Program H_2 to output the right messages** which it received from the ideal functionality to their respective queries, which it can now compute.

$$M_0 \leftarrow \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^a + t_a + u)$$
$$M_1 \leftarrow \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^{1-a} + t_{1-a} + u)$$

UC simulation

Corrupted Sender

- **Program H_1 such that it is able to recover both keys** sk_S^i obtained by the Sender (this is indistinguishable from uniform output — RLWE assumption).

$$\mathsf{H}_1(r) = p_\mathsf{R}^1 - p_\mathsf{R}^0$$

- With both keys, **extract the value a from the malicious Sender** (if a is not found, 3rd condition safeguards indistinguishability).

$$\mathsf{H}_2(\mathsf{sk}_\mathsf{R}^0) = K_a \wedge \mathsf{H}_2(\mathsf{sk}_\mathsf{R}^1) = K_{1-a} \implies a = 0$$
$$\mathsf{H}_2(\mathsf{sk}_\mathsf{R}^1) = K_a \wedge \mathsf{H}_2(\mathsf{sk}_\mathsf{R}^0) = K_{1-a} \implies a = 1$$

- **Program H_2 to output the right messages** which it received from the ideal functionality to their respective queries, which it can now compute.

$$M_0 \leftarrow \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^a + t_a + u)$$
$$M_1 \leftarrow \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^{1-a} + t_{1-a} + u)$$

UC simulation

Corrupted Receiver

- **Program the RO to set the bit a such that it specifies the bit b** to be the same bit from the ideal functionality.

$$b = 0 \implies \text{reply } \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^0) = K_a \vee \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^1) = K_a \text{ (if queried on both, abort)}$$
$$b = 1 \implies \text{reply } \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^0) = K_{1-a} \vee \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^1) = K_{1-a} \text{ (if queried on both, abort)}$$

- **Program the oracle H_2 to output M** when queried on the shared key (if queried on both aborts).

$$M_b \leftarrow \mathsf{H}_2(\mathsf{sk}_\mathsf{R} + t_c + u)$$

UC simulation

Corrupted Receiver

- **Program the RO to set the bit a such that it specifies the bit b** to be the same bit from the ideal functionality.

$$b = 0 \implies \text{reply } \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^0) = K_a \vee \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^1) = K_a \text{ (if queried on both, abort)}$$
$$b = 1 \implies \text{reply } \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^0) = K_{1-a} \vee \mathsf{H}_2(\mathsf{sk}_\mathsf{S}^1) = K_{1-a} \text{ (if queried on both, abort)}$$

- **Program the oracle H_2 to output M** when queried on the shared key (if queried on both aborts).

$$M_b \leftarrow \mathsf{H}_2(\mathsf{sk}_\mathsf{R} + t_c + u)$$

UC simulation

Remaining cases

- When **no party is corrupted**, the simulator has no input from the ideal functionality. So, the simulator generates and honestly executes the protocol and the adversary just observes the transcript. The transcript is indistinguishable which can be proved using the definition of RO and the RLWE assumption.

- When **both parties are corrupted**, the simulator simply runs the adversary internally which generates the messages for both parties.

UC simulation

Remaining cases

- When **no party is corrupted**, the simulator has no input from the ideal functionality. So, the simulator generates and honestly executes the protocol and the adversary just observes the transcript. The transcript is indistinguishable which can be proved using the definition of RO and the RLWE assumption.

- When **both parties are corrupted**, the simulator simply runs the adversary internally which generates the messages for both parties.

Implementation

ROT

- **Gaussian sampling** was implemented using NFLlib. We assume a shared region of memory periodically populated with random data, so the protocol only needs to read data off memory instead of generating random numbers.
- **Random Oracles** were implemented by hashing the inputs and using the output of the hash as a seed to a pseudo-random generator (Hash-DRBG [BK12]). This generator was then used to produce the output of the RO.
- In the case of **sampling a polynomial**, rejection sampling required extensive calls to the underlying hash function (BLAKE3 chosen).
- **NTT** implemented using NFLlib, extended to support ARM with NEON SIMD. We transmit polynomials, and consider the outputs of ROs in the NTT domain.
- Implementation uses 16% (23.9KiB) more memory than the state-of-the-art.

Implementation

ROT

- **Gaussian sampling** was implemented using NFLlib. We assume a shared region of memory periodically populated with random data, so the protocol only needs to read data off memory instead of generating random numbers.
- **Random Oracles** were implemented by hashing the inputs and using the output of the hash as a seed to a pseudo-random generator (Hash-DRBG [BK12]). This generator was then used to produce the output of the RO.
- In the case of **sampling a polynomial**, rejection sampling required extensive calls to the underlying hash function (BLAKE3 chosen).
- **NTT** implemented using NFLlib, extended to support ARM with NEON SIMD. We transmit polynomials, and consider the outputs of ROs in the NTT domain.
- Implementation uses 16% (23.9KiB) more memory than the state-of-the-art.

Implementation

ROT

- **Gaussian sampling** was implemented using NFLlib. We assume a shared region of memory periodically populated with random data, so the protocol only needs to read data off memory instead of generating random numbers.
- **Random Oracles** were implemented by hashing the inputs and using the output of the hash as a seed to a pseudo-random generator (Hash-DRBG [BK12]). This generator was then used to produce the output of the RO.
- In the case of **sampling a polynomial**, rejection sampling required extensive calls to the underlying hash function (BLAKE3 chosen).
- **NTT** implemented using NFLlib, extended to support ARM with NEON SIMD. We transmit polynomials, and consider the outputs of ROs in the NTT domain.
- Implementation uses 16% (23.9KiB) more memory than the state-of-the-art.

Implementation

ROT

- **Gaussian sampling** was implemented using NFLlib. We assume a shared region of memory periodically populated with random data, so the protocol only needs to read data off memory instead of generating random numbers.
- **Random Oracles** were implemented by hashing the inputs and using the output of the hash as a seed to a pseudo-random generator (Hash-DRBG [BK12]). This generator was then used to produce the output of the RO.
- In the case of **sampling a polynomial**, rejection sampling required extensive calls to the underlying hash function (BLAKE3 chosen).
- **NTT** implemented using NFLlib, extended to support ARM with NEON SIMD. We transmit polynomials, and consider the outputs of ROs in the NTT domain.
- Implementation uses 16% (23.9KiB) more memory than the state-of-the-art.

Implementation

ROT

- **Gaussian sampling** was implemented using NFLlib. We assume a shared region of memory periodically populated with random data, so the protocol only needs to read data off memory instead of generating random numbers.
- **Random Oracles** were implemented by hashing the inputs and using the output of the hash as a seed to a pseudo-random generator (Hash-DRBG [BK12]). This generator was then used to produce the output of the RO.
- In the case of **sampling a polynomial**, rejection sampling required extensive calls to the underlying hash function (BLAKE3 chosen).
- **NTT** implemented using NFLlib, extended to support ARM with NEON SIMD. We transmit polynomials, and consider the outputs of ROs in the NTT domain.
- Implementation uses 16% (23.9KiB) more memory than the state-of-the-art.

Implementation

ROT results

ARM Cortex-A7 @ 900MHz	CLK (k)	Time (μs)	ROTs/s
[PVW08] ROTted	18258	20287	50
[BDGM19] ROTted (Serial)	843	936.6	1068
[BDGM19] ROTted (NEON)	666	739.7	1352
This work **(Serial)**	829.08	921.2	1086
This work **(NEON)**	644.94	716.6	1396
ARM Cortex-A53 @ 1.4GHz			
[PVW08] ROTted	12864.6	9189	109
[BDGM19] ROTted (Serial)	589.3	420.9	2376
[BDGM19] ROTted (NEON)	450	321.4	3112
This work **(Serial)**	574.98	410.7	2435
This work **(NEON)**	429.52	306.8	3260
ARM Cortex-A72 @ 1.5GHz			
[PVW08] ROTted	7378.5	4919	204
[BDGM19] ROTted (Serial)	374.6	249.7	4005
[BDGM19] ROTted (NEON)	299.4	199.6	5011
This work **(Serial)**	362.7	241.8	4136
This work **(NEON)**	286.5	191	5236

Apple M1 @ 3.2GHz			
[PVW08] ROTted	1407.7	439.9	2274
[BDGM19] ROTted (Serial)	164.8	51.5	19418
[BDGM19] ROTted (NEON)	129.6	40.5	24692
This work **(Serial)**	154.6	48.3	20704
This work **(NEON)**	120	37.5	26667
Intel i9-10980XE @ 3GHz			
[PVW08] ROTted	1310.7	436.9	2289
[BDGM19] ROTted (Serial)	151.5	50.5	19802
[BDGM19] ROTted (SSE4)	97.2	32.4	30865
[BDGM19] ROTted (AVX2)	96.3	32.1	31153
[BDGM19] ROTted (AVX512)	99.6	33.2	30121
This work **(Serial)**	147	49	20409
This work **(SSE4)**	91.2	30.4	32895
This work **(AVX2)**	81.6	27.2	36765
This work **(AVX512)**	90.3	30.1	33223

Implementation

PSI

- The proposed ROT integrated in the framework of [PRTY, EUROCRYPT2020].
- The framework is for x86, so the results are only provided for that platform.
- Achieve a 6.6x speedup against ROTted version of [PVW, CRYPTO2008], and a 2.1x speedup against the ROTted version of [BDGM, IMACC2019].

Intel @ i9-10980XE 3GHz	Time (ms)
[PVW08] ROTted	932
[BDGM19] ROTted (Serial)	328
[BDGM19] ROTted (AVX2)	304
[BDGM19] ROTted (AVX512)	318
This work (Serial)	166
This work (AVX2)	142
This work (AVX512)	151

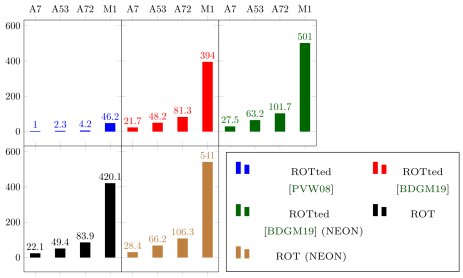
Implementation

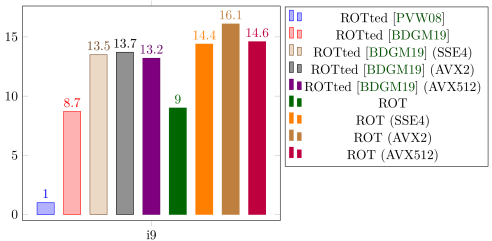
PSI

- The proposed ROT integrated in the framework of [PRTY, EUROCRYPT2020].
- The framework is for x86, so the results are only provided for that platform.
- Achieve a 6.6x speedup against ROTted version of [PVW, CRYPTO2008], and a 2.1x speedup against the ROTted version of [BDGM, IMACC2019].

Intel @ i9-10980XE 3GHz	**Time (ms)**
[PVW08] ROTted	932
[BDGM19] ROTted (Serial)	328
[BDGM19] ROTted (AVX2)	304
[BDGM19] ROTted (AVX512)	318
This work (Serial)	166
This work (AVX2)	142
This work (AVX512)	151

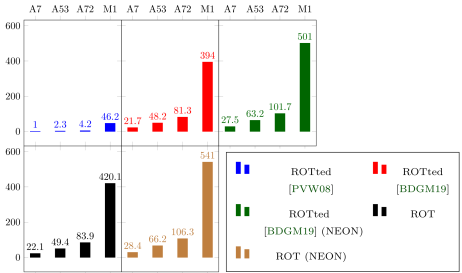
Comparison
ROT

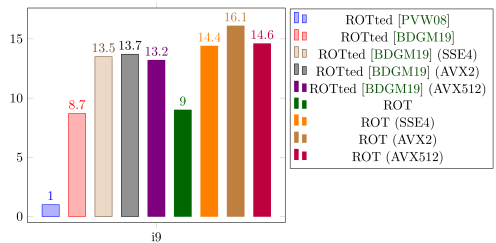




- Profiling [PVW08] shows almost 50% of the execution performing point multiplications.
- RLWE benefits both from issuing multiple instructions and from SIMD.
- Speedups from vector instruction are around 30%; the change of the execution backend provides speedups over 100%.
- The RLWE AVX512 implementations were not able to fill the vector. Therefore, length checks had to be added in the NTT loop, leading to more missed branch predictions.

Comparison
ROT

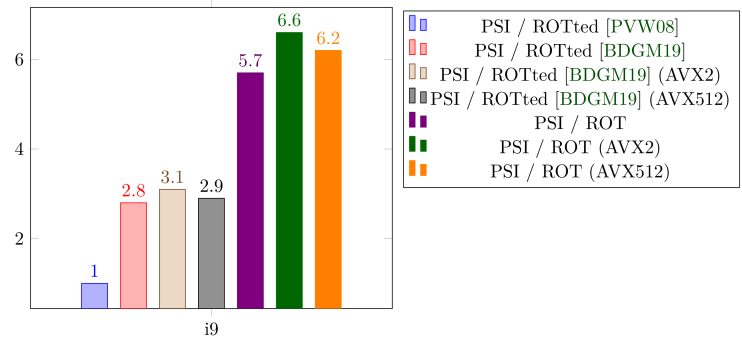




- Profiling [PVW08] shows almost 50% of the execution performing point multiplications.
- RLWE benefits both from issuing multiple instructions and from SIMD.
- Speedups from vector instruction are around 30%; the change of the execution backend provides speedups over 100%.
- The RLWE AVX512 implementations were not able to fill the vector. Therefore, length checks had to be added in the NTT loop, leading to more missed branch predictions.

Comparison

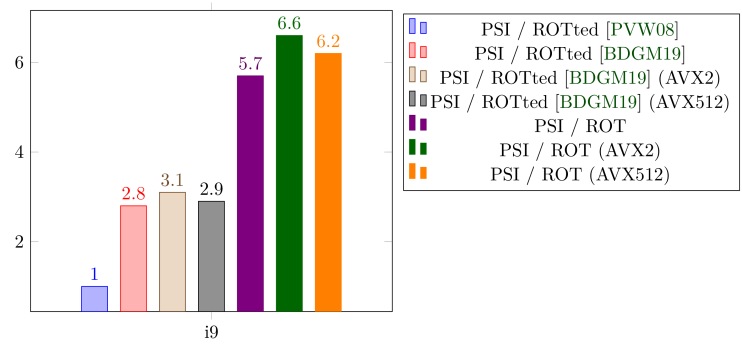
PSI



- The proposed ROT use in a PSI provides a 6.6x speedup when compared with the ROTted version of [PVW08] and a 2.1x speedup with the ROTted version [BDGM19].
- Again, AVX2 implementation is the fastest. The vector implementations show the same speedup when compared to the serial implementation.
- The memory requirements by the PSI framework far exceed those of the proposed ROT.

Comparison

PSI



- The proposed ROT use in a PSI provides a 6.6x speedup when compared with the ROTted version of [PVW08] and a 2.1x speedup with the ROTted version [BDGM19].
- Again, AVX2 implementation is the fastest. The vector implementations show the same speedup when compared to the serial implementation.
- The memory requirements by the PSI framework far exceed those of the proposed ROT.

Conclusion

- **Highly efficient UC-secure ROT** from the RLWE assumption in the ROM.
- Up to **37k ROTs/s** for the Intel server-class processor and up to **5k ROTs/s** in an ARM application-class processor.
- Usage of vector instructions provides on average a **40% speedup**.
- **One order of magnitude faster** than state-of-the-art, and suitable for a wide range of architectures in embedded systems, IoT, desktops and servers.
- **Practical interest** shown in a PSI framework with applications in contact discovery, remote diagnosis, contact tracing, among others.
- Usage of our ROT in a PSI application is up to **6x faster than related art**.
- **Future work** will address ultra-low power devices, e.g. ARM Cortex-M, RISC-V.

Conclusion

- **Highly efficient UC-secure ROT** from the RLWE assumption in the ROM.
- Up to **37k ROTs/s** for the Intel server-class processor and up to **5k ROTs/s** in an ARM application-class processor.
- Usage of vector instructions provides on average a **40% speedup**.
- **One order of magnitude faster** than state-of-the-art, and suitable for a wide range of architectures in embedded systems, IoT, desktops and servers.
- **Practical interest** shown in a PSI framework with applications in contact discovery, remote diagnosis, contact tracing, among others.
- Usage of our ROT in a PSI application is up to **6x faster than related art**.
- **Future work** will address ultra-low power devices, e.g. ARM Cortex-M, RISC-V.

Conclusion

- **Highly efficient UC-secure ROT** from the RLWE assumption in the ROM.
- Up to **37k ROTs/s** for the Intel server-class processor and up to **5k ROTs/s** in an ARM application-class processor.
- Usage of vector instructions provides on average a **40% speedup**.
- **One order of magnitude faster** than state-of-the-art, and suitable for a wide range of architectures in embedded systems, IoT, desktops and servers.
- **Practical interest** shown in a PSI framework with applications in contact discovery, remote diagnosis, contact tracing, among others.
- Usage of our ROT in a PSI application is up to **6x faster than related art**.
- **Future work** will address ultra-low power devices, e.g. ARM Cortex-M, RISC-V.

Conclusion

- **Highly efficient UC-secure ROT** from the RLWE assumption in the ROM.
- Up to **37k ROTs/s** for the Intel server-class processor and up to **5k ROTs/s** in an ARM application-class processor.
- Usage of vector instructions provides on average a **40% speedup**.
- **One order of magnitude faster** than state-of-the-art, and suitable for a wide range of architectures in embedded systems, IoT, desktops and servers.
- **Practical interest** shown in a PSI framework with applications in contact discovery, remote diagnosis, contact tracing, among others.
- Usage of our ROT in a PSI application is up to **6x faster than related art**.
- **Future work** will address ultra-low power devices, e.g. ARM Cortex-M, RISC-V.

Conclusion

- **Highly efficient UC-secure ROT** from the RLWE assumption in the ROM.
- Up to **37k ROTs/s** for the Intel server-class processor and up to **5k ROTs/s** in an ARM application-class processor.
- Usage of vector instructions provides on average a **40% speedup**.
- **One order of magnitude faster** than state-of-the-art, and suitable for a wide range of architectures in embedded systems, IoT, desktops and servers.
- **Practical interest** shown in a PSI framework with applications in contact discovery, remote diagnosis, contact tracing, among others.
- Usage of our ROT in a PSI application is up to **6x faster than related art**.
- **Future work** will address ultra-low power devices, e.g. ARM Cortex-M, RISC-V.

Conclusion

- **Highly efficient UC-secure ROT** from the RLWE assumption in the ROM.
- Up to **37k ROTs/s** for the Intel server-class processor and up to **5k ROTs/s** in an ARM application-class processor.
- Usage of vector instructions provides on average a **40% speedup**.
- **One order of magnitude faster** than state-of-the-art, and suitable for a wide range of architectures in embedded systems, IoT, desktops and servers.
- **Practical interest** shown in a PSI framework with applications in contact discovery, remote diagnosis, contact tracing, among others.
- Usage of our ROT in a PSI application is up to **6x faster than related art**.
- **Future work** will address ultra-low power devices, e.g. ARM Cortex-M, RISC-V.

Conclusion

- **Highly efficient UC-secure ROT** from the RLWE assumption in the ROM.
- Up to **37k ROTs/s** for the Intel server-class processor and up to **5k ROTs/s** in an ARM application-class processor.
- Usage of vector instructions provides on average a **40% speedup**.
- **One order of magnitude faster** than state-of-the-art, and suitable for a wide range of architectures in embedded systems, IoT, desktops and servers.
- **Practical interest** shown in a PSI framework with applications in contact discovery, remote diagnosis, contact tracing, among others.
- Usage of our ROT in a PSI application is up to **6x faster than related art**.
- **Future work** will address ultra-low power devices, e.g. ARM Cortex-M, RISC-V.