No Title Given

No Author Given

No Institute Given

Abstract. Picnic is a post-quantum digital signature scheme, where the security is based on the difficulty of inverting a symmetric block cipher and zero-knowledge proofs.
Based on our paper at the 23rd Euromicro Conference ([6]), we found an optimization, which reduces memory usage to make it usable on IoT devices. This paper focusses on approaches for the implementation of this optimization. In particular, we make use of hardware-implementations of AES and SHAKE (SHA-3) on microntrollers.
We analyze the performance in case of exchanging SHAKE to AES algorithm and vice-versa.

1 Introduction

Post-quantum cryptography is a special field of cryptography, which has the goal to be resistant against attacks from quantum computers as far as we know. In 1994 Peter Shor published an algorithm [5], which can solve the integer factorization problem (and the discrete logarithm problem) in polynomial time. Therefore, a sufficient strong quantum computer can break RSA, ElGamal and Diffie-Hellman algorithms. The main target of post-quantum cryptography is to replace classical asymmetric cryptography by post-quantum resistant algorithms.

In 2017 *National Institute of Standards and Technology* (NIST) started a post-quantum cryptography standardization process. At the end of 2017, 82 cryptographic algorithms were submitted, 69 of them were accepted for the first round. On January 30, 2019, 26 of these candidates were accepted for the second round. Most of the accepted algorithms are based on lattices or error-correcting codes. If there is a structural weakness found in lattices or codes, many schemes can be broken. Picnic is the only algorithm based on zero-knowledge proofs (see [3]). Thus, Picnic is a valuable alternative to lattice and code-based post-quantum cryptography.

As an advantage over lattice- and code-based schemes, Picnic has smaller key sizes (max. 256 bit) depending on the security level. The levels of security are defined by NIST [4]. However, Picnic signatures are very large compared to other post-quantum algorithms, e.g. for security level 5 the maximum size of a signature is 132.856 bytes. Consequently, current Picnic implementations are not suited for memory-limited devices, such as IoT nodes or smartcards. In [6], we provided an optimization, which reduces the memory consumption of the implementation. In this paper we focus on the implementation of Picnic on a microcontroller.

2 Preliminaries

Picnic signature scheme was first published in 2017 [2]. The security of the scheme relies on the hardness of inverting the LowMC [1] block cipher. Given the public key (C, p), the signer generates a Fiat-Shamir transcription proving the knowledge of y with $\text{LowMC}(p, y) = C$. For generating the Fiat-Shamir transcript the to-be-signed message is included to the hashed value for the challenge. Hence, the transcript is connected to the public key and the message and can be verified by any user knowing the public key and the message.

2.1 Picnic

There are various variants of Picnic. On our optimization, we focus on the Fiat-Shamir version of the Picnic algorithm. We does not cover Unruh transformation or Picnic 2. This section describes the parameters used to instantiate Picnic with respect to the security levels L1, L3, and L5.

Table 1 shows the parameters of the algorithm depending on the security level. S is the length of the state (i.e. the length of the input, output, and key of the LowMC cipher) in bits. Furthermore, all generated seeds also have this length. The number r represents the number of rounds needed to compute the result of one LowMC encryption. The number of repetitions of the *Multi Party Computation* (MPC) protocol is denoted as T. All commitments have the length ℓ. An *Extendable Output Function* (XOF) is a hash function with variable output length. It is used for commitments and for generating pseudorandom bit strings, such as the challenge or the seeds.

Table 1. Picnic parameters by security level [3]

security level	S	r	T	ℓ	XOF
L1	128	20	219	256	`SHAKE128`
L3	192	30	329	384	`SHAKE256`
L5	256	38	438	512	`SHAKE256`

Table 1 shows the parameters of the algorithm depending on the security level. S is the length of the state (i.e. the length of the input, output, and key of the LowMC cipher) in bits. Furthermore, all generated seeds also have this length. The number r represents the number of rounds needed to compute the result of one LowMC encryption. The number of repetitions of the MPC protocol is denoted as T. All commitments have the length ℓ. An XOF is a hash function with variable output length. It is used for commitments and for generating pseudorandom bit strings, such as the challenge or the seeds.

2.2 Picnic Structure

The structure of the Picnic algorithm consists of four steps. Figure 1 shows the main structure of the signature generation and visualizes the steps.

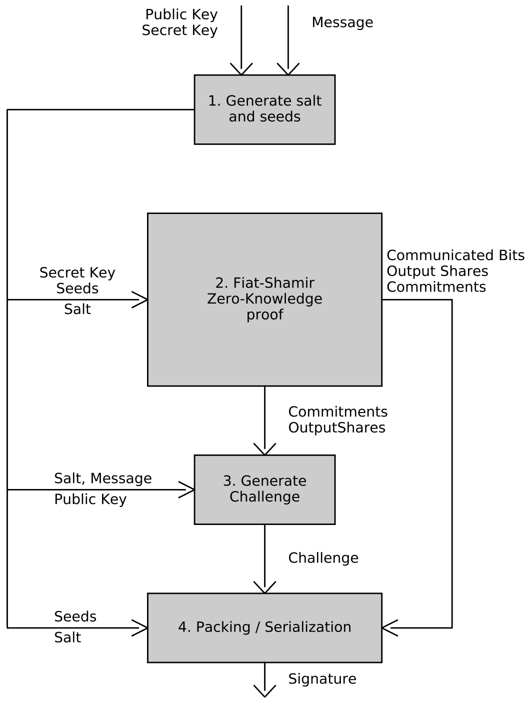

Fig. 1. Picnic algorithm data flow. In the original Version the four steps are processed sequentially.

Generate salt and seeds The seeds and the salt are generated by SHAKE-128 or SHAKE-256 and have to be kept in memory during the whole execution of Picnic. In total one salt and $3 \cdot T$ seeds are generated, 3 for each MPC repetition. As defined in [3], the output of the XOF is defined as the concatenated seeds followed by the salt. Therefore, in order to generate the salt the XOF has to generate all the seeds at first.

However, according to [3], the generation of seeds and salts can be changed, without breaking the compatibility to the verification function of the reference implementation. For security reasons, the generation of seeds and salt has to be collission-free and unpredictable.

Running the MPC repetitions This step consists of T repetitions of the MPC protocol. One repetiton consists of the generation of three tapes and the input shares using the secret key, the execution of the evaluation of LowMC using the MPC protocol, and finally the computation of the commitments for this run of the protocol.

The tapes and the first two input shares are pseudorandomly generated from the seed. The third input share is chosen such that the XOR-sum of all shares yields the secret key.

The MPC protocol computes the output shares from the input shares and the tapes and the inter-party-communication (here denoted as *communicated bits*). The length of the communicated bits per party is $30 \cdot r$ bits, where r is the number of repetitions of the LowMC computation. For further computations, the bit string is filled with zeros up to a full byte, namely a size of 75, 113, and 143 bytes for each security level respectively.

The commitments are hash values of communicated bits, input, and output shares. One commitment has $S/4$ bits.

Generate challenge When all runs of the MPC protocol are finished the challenge is computed by an XOF. The inputs for the XOF is all output shares, followed by all commitments, the salt, the public key and finally the message itself.

The challenge consists of values $\{0, 1, 2\}$ where each element is represented via 2 bits of the XOF output. In case of two consecutive ones i.e. `0b11 = 3` these bits are skipped. These guarantees uniform distributed challenges. The binary representation of the challenge is filled to a full byte with zeros at the end. Thus, the size of the challenge string is 55, 83, and 110 bytes respectively.

Packing and Serialization The signature is the concatenation of the challenge and the salt. For each round depending on the challenge $e \in \{0, 1, 2\}$ a commitment for party $e + 2$ and the inter-party communication for party $e + 1$ is appended. Additionally, if $e \neq 0$ the share of the third party is also appended.

Based on Figure 1 our optimizations target on reusing data and withdraw it as soon as possible. In the original version, the seeds have to stay in RAM for the whole execution. There is a similar situation for the commitments and the output shares. As step 2 (Fiat Shamir Zero-Knowledge proof) is run 219, 329 or 438 times depending on the security level, all intermediate results have to be stored until the generation of the challenge in step 3. Our basic idea is to run steps 1, 2 and 3 simultaneously in a way that all results can be reused and then withdrawn immediately.

3 Our Optimizations

We provide three optimizations. The first one is for generating the seeds and the salt. In contrast to the reference implementation, we use AES instead of SHAKE128/SHAKE256. We justify this step with two reasons. At first, we need random access to the seeds and the salt. Secondly, as far as we know, most IoT devices or smartcards, are equipped with an accelerator for AES. Furthermore, for hardware implementations of Picnic, this optimization allows parallelizing the MPC rounds. As our paper is focussed on memory consumption, we will not go deeper into this topic. According to the specification, the generation of seeds and salt does not affect the verification function. Therefore, signatures generated with this optimization are fully compatible with the verification function of the reference implementation.

The second optimization is the way how the challenge is generated. By reordering the input values of the hash function, we can avoid caching the commitments of all MPC rounds. By the definition of a secure cryptographic hash function, the order of the input values does not affect security. However, as the challenge also has to be generated for the verification, this optimization breaks backward compatibility.

As a third modification, we suggest transferring parts of the signature from the IoT device to the host system. The signature itself is assembled by the host. This increases the amount of data which is transferred between the host and IoT device. On the other hand, the transferred data does not have to be stored at the IoT device. This optimization depends on the context of the IoT device.

3.1 Generation of seeds and salt

According to the specification of Picnic [3] the seeds are generated by an XOF such as SHAKE128 (for security level L1) or SHAKE256 (for higher security). By definition, the input of the XOF is

```
XOF(sk||M||pk||S)
```

where `pk` and `sk` are public and secret key, `M` is the message to be signed and `S` is the size of a seed in bit, encoded as 16 bit little endian integer. In total the XOF outputs $3 \cdot T$ seeds, each of them of size $S/8$ followed by the 32 byte salt. The output of the SHAKE function follows the pattern

```
seed[0][0] || seed[0][1] || seed[0][2] ||
seed[1][0] || seed[1][1] || seed[1][2] ||
...
seed[T-1][0] || seed[T-1][1] ||
seed[T-1][2] || salt
```

As it can be seen due to the properties of SHAKE-functions the seeds can only be generated sequentially. Additionally, the salt is obtained at last after all seeds have been generated. This prevents on-the-fly-computation in each repetition. As an optimization we recommend to change the order, i.e. generating the salt at first, and then generating the seeds. Algorithms 1 and 2 describe an independent seed and salt generation based on AES counter mode. As it's also stated in the Picnic specification, the generation of seed and salt does not affect the interoperability of the Picnic scheme. Therefore, signatures generated by our optimized version can be verified by the official reference implementation.

With this modification, we do not need to store $3 \cdot T$ (one for each of the three parties and for T rounds) seeds, where each seed has S bits. Depending on the security level the amount of reduced space is

$$3 \cdot 219 \cdot 16 = 10,512 \text{ bytes for L1},$$
$$3 \cdot 329 \cdot 24 = 23,688 \text{ bytes for L3},$$
$$3 \cdot 438 \cdot 32 = 42,048 \text{ bytes for L5}.$$

Figure 2 shows the optimized version. If operated on a device with a multi-core processor, the operation can be parallelized.

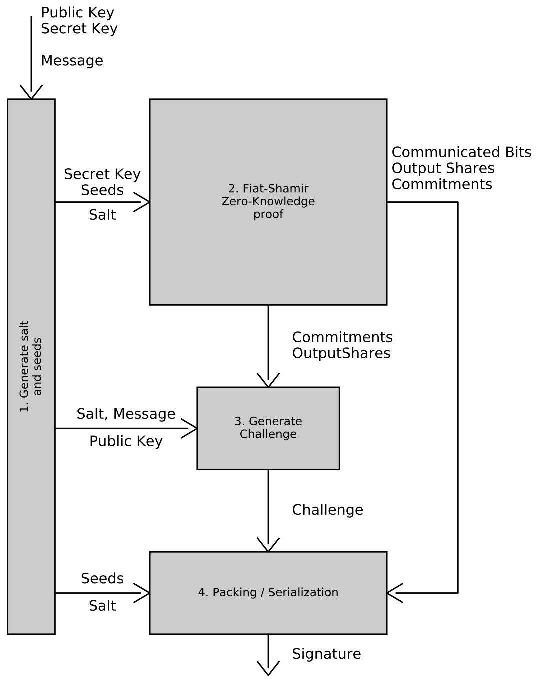

Fig. 2. Picnic algorithm data flow with modified seed/salt generation.

Algorithm 1 Picnic's salt generation

```
1: procedure GenerateSalt(sk, M, pk, S)
2:     g = sk||M||pk||S
3:     r = SHAKE-256(SHA3-256(g), 128)
4:     k = SHAKE-256(g, S)
5:     return salt = AES(r,k) || AES(r+1,k)
6: end procedure
```

Algorithm 2 Picnic's seed generation

```
procedure GenerateSeed(sk, M, pk, S, seclevel, repetition)
    g = sk||M||pk||S
    r = SHAKE-256(SHA3-256(g), 128)
    k = SHAKE-256(g, S)
    r += 2
    if seclevel == L1 then
        k += 3 * repetition
        seed1 = AES(r,k)
        seed2 = AES(r+1,k)
        seed3 = AES(r+2,k)
    else if seclevel == L3 then
        k += 5 * repetition
        seed1 = AES(r,k) || AES(r+1,k).substr(0,63)
        seed2 = AES(r+1,k).substr(64,127) || AES(r+2,k)
        seed3 = AES(r+3,k) || AES(r+4,k).substr(0,63)
    else if seclevel == L5 then
        k += 6 * repetition
        seed1 = AES(r,k) || AES(r+1,k)
        seed2 = AES(r+2,k) || AES(r+3,k)
        seed3 = AES(r+4,k) || AES(r+5,k)
    end if
    return (seed1,seed2,seed3)
end procedure
```

3.2 Computation of Challenge

The challenge depends on all output shares (denoted as `out`), all commitments (denoted as `com`), the salt, the public key (denoted as `pk`), and the message (denoted as `msg`). For computing the challenge a specially designed `H3` hash function is used. This function is defined in [3]. In contrast to other hash functions `H3` maps on $\{0, 1, 2\}^T$, which represents a list of challenge for each repetition. The order of the input data as specified in [3] is

```
e = H3(
out[0][0] || out[0][1] || out[0][2]
out[1][0] || out[1][1] || out[1][2]
...
out[T-1][0] || out[T-1][1] || out[T-1][2]
com[0][0] || com[0][1] || com[0][2]
com[1][0] || com[1][1] || com[1][2]
...
com[T-1][0] || com[T-1][1] || com[T-1][2]
salt, pk, msg).
```

As a way of saving memory, the output shares obtained in one particular MPC repetition can be hashed immediately after the repetition and then be discarded.

However, the commitments have to be stored in memory until all output shares are hashed. Our approach is to hash three output shares and three commitments per repetition.

```
e = H3(
out[0][0] || out[0][1] || out[0][2]
com[0][0] || com[0][1] || com[0][2]
out[1][0] || out[1][1] || out[1][2]
com[1][0] || com[1][1] || com[1][2]
...
out[T-1][0] || out[T-1][1] || out[T-1][2]
com[T-1][0] || com[T-1][1] || com[T-1][2]
salt, pk, M)
```

It follows that we do not need to keep commitments or output shares in memory during the signing operation.

With this modification, we do not need to store $3 \cdot T$ commitments, where each seed has ℓ bits. Depending on the security level the amount of reduced space is

$$\begin{aligned} 3 \cdot 219 \cdot 32 &= 21,024 \text{ bytes for L1,} \\ 3 \cdot 329 \cdot 48 &= 47,376 \text{ bytes for L3,} \\ 3 \cdot 438 \cdot 64 &= 84,096 \text{ bytes for L5.} \end{aligned}$$

Figure 3 shows the first two optimizations. It can be seen that step 1 and step 3 of Picnic can be parallelized.

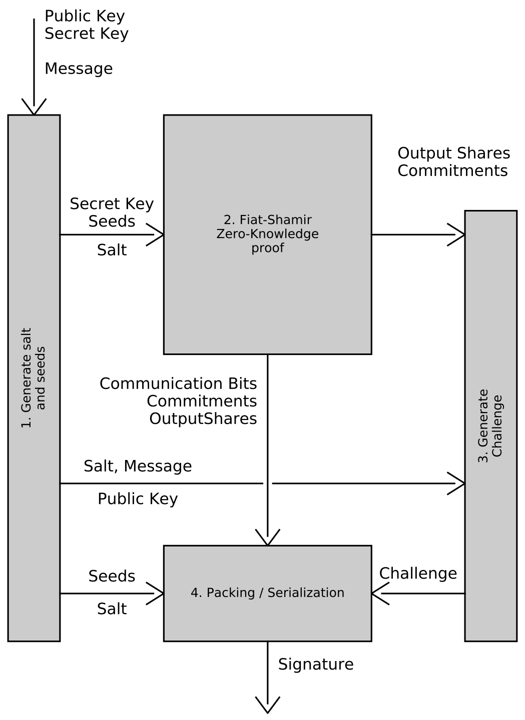

Fig. 3. Picnic algorithm data flow with modified seed/salt/challenge generation.

3.3 Stream encrypted temporary results

Due to the limited memory of IoT devices, some intermediate results have to be transferred to the host system. However, deciding which information can be sent outside is a non-trivial task from security perspective.

Encryption As an encryption method, we suggest AES-128, AES-192, or AES-256 depending on the security level. The security of these algorithms exactly matches to the NIST security levels 1, 3, and 5. In order to avoid generating a random initialization vector, we use *Electronic Codebook* (ECB) mode. We do not expect any problems with ECB mode, as our data is pseudorandom and has a fixed length.

We suggest to derive the encryption keys from the seeds. Therefore, we are not required to generate keys randomly and store them. As a result, we do not have to transfer the key from the device to the host.

Seeds Transferring the seeds could be critical, as the seed of the first two parties also generates the input share. For any challenge $e \neq 0$, the input share of the last party is given. By construction, XORing all input shares give the secret key. Therefore, the IoT device must not send all three seeds to the host.

As the seeds are withdrawn in each repetition, they have to be generated again. According to our proposal, the generation of the seeds is AES-based and, therefore, it is faster than the generation by SHAKE XOF and the seeds can be randomly accessed.

Commitments As the commitments are needed for the final signature, we suggest transfering all three commitments to the host in each repetition. As only one of three commitment per repetition is necessary for the signature, we increase the amount of data which is transferred between the IoT device and the host system. However, as the commitments can be transferred by the end of each repetition, no speed losses occur. From the security perspective, the commitments can be computed by anyone knowing the signature and the public key, and therefore our approach does not affect the security of the scheme.

Input share of 3rd party The input share of the 3rd party has to be included in the signature if, and only if, the challenge is not 0. On the other hand, it must not be included for any other challenge. Our approach is to derive the encryption key from the seed of the 3rd party. It can be easily seen, that the seed for encryption is only known, if the challenge is 1 or 2.

In comparison to the classical Picnic, the third (encrypted) input share is sent for each repetition of the MPC protocol and, therefore, also increases the size of the communication.

As a consequence, the communication stream between host and TPM has a fixed length.

Communication Bits The communication bits are the result of the AND gates of a party. As there are $30 \cdot r$ AND gates per repetition, there are 75 bytes, 113 bytes, and 143 bytes for each party at a security levels L1, L3, and L5 respectively. For one repetition with challenge e, only the communcation bits of party $e + 1$ are needed. As the challenge is unknown while running the MPC protocol, all communications bits have to be transferred to the host system. As the communication bits can leak information about the secret key, they have to be encrpyted. The key for decryption is derived from seed $e + 1$.

Serialization of the signature To parallelize computation and communication, parts of the signature are sent to host. Therefore, the host has to convert all parts of the signature to a valid format according to the specification. A valid Picnic signature has to start with the challenge followed by the salt. For each repetition of the MPC protocol, following values have to be included to the signature: commitment for party $e + 2$, communication for party $e + 1$ (decrypted by the host), seeds for party e and $e+1$. The serialization of the signature will be executed on the host system. The result of this procedure is a valid signature according to [3] (except the changes of optimization 2).

This modification avoids storing $3 \cdot T$ communication bits with the length of $\{75, 113, 143\}$ bytes depending on the security level. Furthermore, it avoids storing the share of the third party, which has S bits. For the security levels L1, L3 and L5, the reduced amount of space is

$$\begin{aligned} 219 \cdot (3 \cdot 75 + 16) &= 52,779 \text{ bytes}, \\ 329 \cdot (3 \cdot 113 + 24) &= 119,427 \text{ bytes}, \\ 438 \cdot (3 \cdot 143 + 32) &= 201,918 \text{ bytes}, \end{aligned}$$

respectively.

Algorithm 3 is the code of the algorithm which generates the signature from the data which is transferred from the IoT device.

Algorithm 3 Packing and Serialization (Host system)

```
procedure SERIALIZATION(challenge, salt, commitments, seeds, communication, ishare3)
    sig = challenge || salt
    for k = 0 to T do
        e = challenge.getBit(k)
        key1 = deriveKey(seeds[0])
        key2 = deriveKey(seeds[1])
        sig.append(commitments[k][e + 2])
        sig.append(decrypt(communication[k][e + 1], key2))
        sig.append(seeds[0])                              ▷ only 2 seeds incl.
        sig.append(seeds[1])
        if $e = 1$ then
            sig.append(decrypt(ishare3[k]), key2)
        else if $e = 2$ then
            sig.append(decrypt(ishare3[k]), key1)
        end if
    end for
end procedure
```

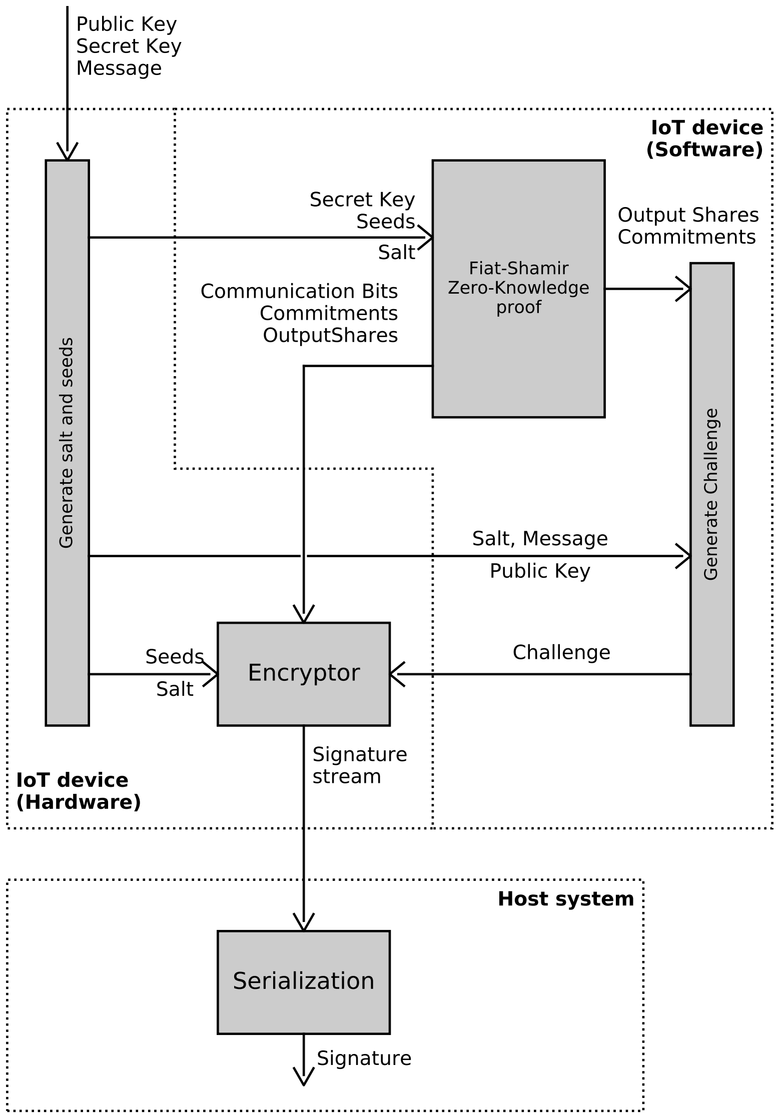

Fig. 4. Picnic algorithm on IoT device partitioned in a hardware and a software part. The algorithm for serialization is on the host system and, therefore, does not have to be optimized.

4 Results

We provided a concept for implementing the Picnic algorithm on a hardware plattform with an AES-coprocessor. For this reason, we partitionate the algorithm from the IoT device to a hardware- and a software part.

In this section, we list how much memory can be saved by our optimizations. Table 2 shows the exact amount of reduced memory. These numbers have been determined analytically by comparing our optimization to the reference implementation. Therefore, we focus on the content-depending memory i.e. keys, states, seeds, random numbers and not on the generic memory such as loop indices or constants.

The first optimization reduces the memory consumption, for storing the seeds. Depending on the level of security, this saves 10 kB to 42 kB. There is no drawback regarding backward compatibility or computation power.

The second optimization saves 21 kB to 84 kB for security levels L1 and L5, respectively. However, signatures generated with this optimization are not compatible with the verification function of the reference implementation.

The third optimization transfers some intermediate results (commitments, communication bits and 3rd party input share) to the host system. This increases the traffic between the IoT device and the host system. The exact amount of traffic is listed in Table 3. The traffic which is transferred between the IoT device and the host is approximately twice to three times as high as the signature which has to be transferred in the original version. However, as some data can be transferred during the generation of the signature.

Table 2. Secret depending memory in bytes after our optimization.

	Without Opt.			With Opt.		
	L1	L3	L5	L1	L3	L5
MPC repetition:						
seeds	11k	24k	42k	48	72	96
tapes / comm.	49k	112k	188k	225	339	429
commitments	21k	47k	84k	96	144	192
I/O shares	≈ 0	≈ 0	≈ 0	96	144	192
3rd party share	3k	8k	14k	–	–	–
total per rep.	–	–	–	**465**	**699**	**909**
total for all rep.	**84k**	**191k**	**328k**	–	–	–
salt	32	32	32	32	32	32
challenge	55	83	110	55	83	110
total	**84k**	**191k**	**328k**	**552**	**814**	**1.051**

Table 3. Picnic signature size by security level compared to traffic size of our third optimization

security level	min in bytes	max in bytes	transferred data in bytes
L1	$30,528$	$34,032$	$80,898$
L3	$68,876$	$76,772$	$182,710$
L5	$118,840$	$132,856$	$314,188$

5 Implementation

We provide an implementation for our optimizations on Github. For comparison, we also provide a version without our implementations. Both variants are implemented with the with the optimizations [7] and [8]. The calculation of the LowMC cipher requires predefined constants, which are hardcoded in our implementations. The code as a software-implementation can be found at:
`url blinded for review`

We expect to finalize our proof-of-concept implementation by the camera-ready date.

Related Work

In [6] we presented three optimizations for reducing the space consumption of the Picnic algorithm.

1. Generating seeds and salt on demand, which saves up to 42 kB RAM.
2. Modified generation of the challenge to avoid caching the commitments. This saves up to 84 kB.
3. Streaming parts of the signature to client while generating. This saves memory for communication bits and input shares. This can save memory up to 200 kB.

Our optimizations focus on the generation of the signature, specifically on the secret-key depending part. The verification does not use the secret key and, therefore, it need not be done on a hardware-secured device, such as an IoT device or a smartcard.

Future Work

The message is needed for the generation of seeds, salt, and challenge. However, for the functionality of Picnic it is only necessary to include the message in the generation of the challenge. Therefore, the seeds and the salt can be generated independently from the message. However, the output of the random number generator has to be unpredictable and the seeds have to be collision-free.

Another aspect is the implementation on a system with a contactless interface. Due to our optimizations the traffic increases by the factor ≈ 2.5. This leads to a delay and to higher power consumption. This topic highly depends on the use-case. For a given use-case the tradeoff between space consumption and traffic has to be discussed individually.

Acknowledgment

This work was supported by the FutureTPM project of Horizon H2020 Framework Programme of the European Union, under GA number 779391.

References

1. Martin Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. Cryptology ePrint Archive, Report 2016/687, 2016. https://eprint.iacr.org/2016/687.
2. Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. Cryptology ePrint Archive, Report 2017/279, 2017. https://eprint.iacr.org/2017/279.
3. Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. The picnic signature algorithm: Specification version 2.1, 2019. last retrieved August 20, 2019.
4. National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.
5. Peter Shor. Algorithms for quantum computation: discrete logarithms and factoring. Proceedings 35th Annual Symposium on Foundations of Computer Science, 1994. DOI: 10.1109/SFCS.1994.365700.
6. Blinded for review
7. Itai Dinur, Daniel Kales, Angela Promitzer, Sebastian Ramacher, and Christian Rechberger. Linear equivalence of block ciphers with partial non-linear layers: Application to lowmc. Cryptology ePrint Archive, Report 2018/772, 2018. https://eprint.iacr.org/2018/772.
8. Daniel Kales, Léo Perrin, Angela Promitzer, Sebastian Ramacher, and Christian Rechberger. Improvements to the linear operations of lowmc: A faster picnic. Cryptology ePrint Archive, Report 2017/1148, 2017. https://eprint.iacr.org/2017/1148.