IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 3, pp. 74–96. DOI:10.46586/tches.v2023.i3.74-96

Enabling FrodoKEM on Embedded Devices

Joppe W. Bos[1], Olivier Bronchain[1], Frank Custers[1], Joost Renes[1], Denise Verbakel[1,2] and Christine van Vredendaal[1]

[1] NXP Semiconductors, Leuven, Belgium
{firstname.lastname}@nxp.com
[2] Radboud University, Nijmegen, Netherlands
denise.verbakel@ru.nl

Abstract. FrodoKEM is a lattice-based Key Encapsulation Mechanism (KEM) based on *unstructured* lattices. From a security point of view this makes it a conservative option to achieve post-quantum security, hence why it is favored by several European authorities (e.g., German BSI and French ANSSI). Relying on unstructured instead of structured lattices (e.g., CRYSTALS-Kyber) comes at the cost of additional memory usage, which is particularly critical for embedded security applications such as smart cards. For example, prior FrodoKEM-640 implementations (using AES) on Cortex-M4 require more than 80 kB of stack making it impossible to run on some embedded systems. In this work, we explore several stack reduction strategies and the resulting time versus memory trade-offs. Concretely, we reduce the stack consumption of FrodoKEM by a factor 2–3× compared to the smallest known implementations with almost no impact on performance. We also present various time-memory trade-offs going as low as 8 kB for all AES parameter sets, and below 4 kB for FrodoKEM-640. By introducing a minor tweak to the FrodoKEM specifications, we additionally reduce the stack consumption down to 8 kB for all the SHAKE versions. As a result, this work enables FrodoKEM on more resource constrained embedded systems.

Keywords: Post-Quantum Cryptography · Small-stack · FrodoKEM

1 Introduction

The security of public-key cryptography is based on conjectured-to-be-hard mathematical problems. The most widely used examples are RSA [RSA78] and Elliptic Curve Cryptography [Kob87, Mil86] which remain secure as long as the integer factorization problem and the discrete logarithm problem are hard. Although this has been a time-tested conjecture in a pre-quantum world, both of these classes of algorithms are vulnerable to polynomial-time attacks in a post-quantum era using a quantum computer [Sho94, PZ03].

Post-Quantum Cryptography (PQC) promises to deliver new type of algorithms that are resistant against quantum (polynomial-time) attacks. There are many types of PQC algorithms based on different mathematical problems. The USA's National Institute of Standards and Technology (NIST) initiated a process in 2016 to select which algorithms will become their new public-key standard in a post-quantum world [Nat]. In 2022, this has culminated to a single Key Encapsulation Mechanism (KEM) and three Digital Signature schemes (DS) being put on track for standardization by 2024 and beyond.

Typically, NIST standards can be considered as global standards. However, it is not uncommon for other countries or regions to define their own schemes *next to* this US standard. For example, in the elliptic-curve domain several curves are standardized by the German (BSI), French (ANSSI) and multiple other governments worldwide. For PQC the situation is not different. Currently multiple European authorities have stated preferences

Received: 2023-01-15 Accepted: 2023-03-15 Published: 2023-06-09

for PQC algorithms which were not selected as the final winners by NIST (examples include the German [Fed], French [Age] and Dutch [Net] governments). The more conservative (in terms of security) choices of FrodoKEM [ABD+20] and Classic McEliece [ABC+22] are prevalent in these documents. Also outside of Europe, it is expected that some countries will choose to adapt their own standard.

In this work, we focus on FrodoKEM. Its security relies on the Learning With Errors (LWE) problem [Reg05] and it was designed to provide a practical post-quantum key exchange mechanism with conservative security. Compared to algebraically structured alternatives, e.g., the NIST winner Kyber [SAB+22], it is widely considered a conservative and secure choice in practice. The major downside is that this lack of algebraic structure also leads to larger keys and ciphertexts and makes implementations of FrodoKEM require more memory and be slower compared to the structured alternatives. For example, the most memory-efficient implementation of FrodoKEM [HOKG18, Table 6] still uses about 23 kB, 41 kB and 51 kB of stack memory for key generation, encapsulation and decapsulation respectively for the smallest parameter set. The stack usage only increases further for larger parameter sets. Other existing scientific works focus mainly on the performance (i.e., runtime) of FrodoKEM on constrained platforms: e.g. [BFM+18, BOR+21]. The main reason that the stack memory usage remains high for embedded devices is that the previous works were not willing to explore memory optimizations that (significantly) impact performance. However, this means that FrodoKEM is simply infeasible to execute on a wide variety of embedded systems. Meanwhile this range of constrained devices is of interest for the European authorities recommending FrodoKEM. Examples of their use include identity documents (passport, driver's license) and access control of high-security infrastructure.

Typical resource constrained examples are platforms which are based on ARM Cortex-M0(+) cores. Such platforms are typical for a large family of IoT applications. Products in this range include the LPC800 series by NXP (4 – 16 KiB of SRAM), STM32F0 by ST (4 – 32 KiB of SRAM), or the XMC1000 by Infineon (16 KiB of SRAM). A similar observation was made in [BRS22] where the authors explore aggressive memory optimizations for Dilithium in order to fit this into target platforms which have significantly limited memory capabilities and computational power.

Contributions. We outline multiple strategies to significantly lower the memory consumption of FrodoKEM in order to run this on resource-constrained (IoT) devices. In particular, we show that the memory usage of FrodoKEM can be reduced significantly with little loss of performance. For example, FrodoKEM-640 can be run using about 14 kB of stack memory (compared to 51 kB in [HOKG18]) with only a 5% performance loss with respect to the pqm4 implementation, making it suddenly feasible to execute this on resource constrained platforms with 16 kB of stack. We also show that all parameter sets of FrodoKEM are feasible to execute on platforms with at most 8 kB of stack memory available, with a slowdown factor of about 3–4× compared to their more memory hungry counterparts. For devices with at most 4 kB of SRAM not all parameter sets fit: however, we show that certain algorithmic strategies result in implementations for FrodoKEM-640 that can be run on such resource constrained platforms. We demonstrate the impact of these techniques in terms of computational costs and illustrate this with actual performance figures using the pqm4 benchmarking platform. We present all algorithmic techniques to reduce memory in such a way that software libraries can immediately alter the existing software and benefit from these techniques.

We note that there is a significant difference between the AES128 and SHAKE versions of FrodoKEM in terms of memory optimizations. To enable low-memory versions using SHAKE we propose a minor modification to the FrodoKEM specification that we hope to be adopted in any future standards as it will significantly improve the performance on embedded systems (with very little impact on other platforms). The memory and

Table 1: The relevant FrodoKEM parameters and matrix dimensions for the various security levels.

Parameter set	NIST security level	q	$\bar{n}$	n
FrodoKEM-640	1	2^{15}	8	640
FrodoKEM-976	3	2^{16}	8	976
FrodoKEM-1344	5	2^{16}	8	1344

Matrix size	Matrix name
$n \times n$	$\mathbf{A}$
$n \times \overline{n}$	$\mathbf{B}$, $\mathbf{E}$, $\mathbf{S}$
$\overline{n} \times n$	$\mathbf{B}'$, $\mathbf{B}''$, $\mathbf{E}'$, $\mathbf{S}'$
$\overline{n} \times \overline{n}$	$\mathbf{C}$, $\mathbf{E}''$, $\mathbf{M}$, $\mathbf{V}$

performance figures that we present already include this modification.

2 Background

In this section we provide the basics of the FrodoKEM algorithm [BCD+16, ABD+20] and its notation, with a focus on the main matrix operations and the generation of these matrices needed in this paper. For further information, we refer the interested reader to the specification of FrodoKEM [ABD+20].

In the remainder of this paper, we follow the notation as in [ABD+20]. The ring of integers modulo q is denoted as $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$. Vectors and matrices are denoted by lower and upper case boldface letters respectively. For example, $\mathbf{B} \in \mathbb{Z}_q^{m \times n}$ denotes a matrix of size $m \times n$ and the matrix element in its i-th row and j-th column is denoted by $\mathbf{B}_{i,j}$. Bit strings are written as a vector over the set $\{0,1\}$. A concatenation of bit strings is denoted by $\|$. For two bit strings $k \in \{0,1\}^m$ and $\ell \in \{0,1\}^n$ their concatenation is written as $k\|\ell \in \{0,1\}^{m+n}$.

2.1 The FrodoKEM Algorithm

FrodoKEM is derived from the Frodo key agreement protocol proposed in [BCD+16], whose security reduces to the hardness of the standard Learning With Errors (LWE) problem [Reg05]. As is typical for PQC KEM schemes, FrodoKEM is built from a public-key encryption scheme (PKE) named FrodoPKE. This PKE is transformed into a KEM via (a variant of) the Fujisaki-Okamoto (FO) transformation [FO99]. The three main algorithms are key generation FrodoKEM.KeyGen, encapsulation FrodoKEM.Encaps and decapsulation FrodoKEM.Decaps (see Algorithm 1).

Since FrodoKEM is based on the standard LWE problem (without structured matrices), its main operations are matrix sampling, multiplication and addition. All the matrices used in FrodoKEM are of size $n \times \overline{n}$, $\overline{n} \times \overline{n}$, $\overline{n} \times n$ or $n \times n$ for both n and $\overline{n}$ depending on the selected parameter set as recalled in Table 1. These matrices consist of elements from $\mathbb{Z}_q$ where q is a small power of two, which makes modular reduction very cheap on a modern computer architecture. In the following sections we first describe the matrix multiplication and addition involved in the three algorithms. Then, we detail the sampling of these matrices from a random seed.

2.1.1 Matrix Operations

In FrodoKEM.KeyGen, the secret key is first generated and then used to compute the public key. Concretely, the *secret* matrices $\mathbf{S}, \mathbf{E} \in \mathbb{Z}_q^{n \times \overline{n}}$ are generated by sampling from the small Gaussian distribution χ. The *public* matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ is generated by calling FrodoKEM.Gen (see Section 2.1.2) with a seed $\mathsf{seed}_{\mathbf{A}}$. Then, the *public* matrix $\mathbf{B} \in \mathbb{Z}_q^{n \times \overline{n}}$ is computed as $\mathbf{B} = \mathbf{A} \cdot \mathbf{S} + \mathbf{E}$ and consists of an expensive matrix multiplication and

Algorithm 1 FrodoKEM KeyGen, Encaps and Decaps algorithm descriptions.

Function: FrodoKEM.KeyGen
Input: None.
Output: (pk, sk')
1: $\mathbf{s}\|\mathsf{seed}_{\mathbf{SE}}\|\mathbf{z} \leftarrow\$ U(\{0,1\}^{\mathsf{len_s}+\mathsf{len}_{\mathsf{seed}_{\mathbf{SE}}}+\mathsf{len_z}})$
2: $\mathsf{seed}_{\mathbf{A}} \leftarrow \mathrm{SHAKE}(\mathbf{z}, \mathsf{len}_{\mathsf{seed}_{\mathbf{A}}})$
3: $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$ via $\mathbf{A} \leftarrow \mathsf{Frodo.Gen}(\mathsf{seed}_{\mathbf{A}})$
4: $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(2n\overline{n}-1)}) \leftarrow \mathrm{SHAKE}(\mathtt{0x5F}\|\mathsf{seed}_{\mathbf{SE}}, 2n\overline{n}\cdot\mathsf{len}_\chi)$
5: $\mathbf{S}^{\mathrm{T}} \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(n\overline{n}-1)}), \overline{n}, n, T_\chi)$
6: $\mathbf{E} \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(n\overline{n})}, \mathbf{r}^{(n\overline{n}+1)}, \ldots, \mathbf{r}^{(2n\overline{n}-1)}), n, \overline{n}, T_\chi)$
7: $\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E}$
8: $\mathbf{b} \leftarrow \mathsf{Frodo.Pack}(\mathbf{B})$
9: $\mathbf{pkh} \leftarrow \mathrm{SHAKE}(\mathsf{seed}_{\mathbf{A}}\|\mathbf{b}, \mathsf{len}_{\mathbf{pkh}})$
10: **return** $pk \leftarrow \mathsf{seed}_{\mathbf{A}}\|\mathbf{b}$, $sk' \leftarrow (\mathbf{s}\|\mathsf{seed}_{\mathbf{A}}\|\mathbf{b}, \mathbf{S}^{\mathrm{T}}, \mathbf{pkh})$

Function: FrodoKEM.Encaps
Input: $pk = \mathsf{seed}_{\mathbf{A}}\|\mathbf{b} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}+D\cdot n\cdot\overline{n}}$.
Output: $\mathbf{c}_1\|\mathbf{c}_2 \in \{0,1\}^{(\overline{m}\cdot n+\overline{m}\cdot\overline{n})D}$, $\mathbf{ss} \in \{0,1\}^{\mathsf{len_{ss}}}$.
1: $\mu \leftarrow\$ U(\{0,1\}^{\mathsf{len}_\mu})$
2: $\mathbf{pkh} \leftarrow \mathrm{SHAKE}(pk, \mathsf{len_{pkh}})$
3: $\mathsf{seed}_{\mathbf{SE}}\|\mathbf{k} \leftarrow \mathrm{SHAKE}(\mathbf{pkh}\|\mu, \mathsf{len}_{\mathsf{seed}_{\mathbf{SE}}} + \mathsf{len_k})$
4: $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(2\overline{m}n+\overline{mn}-1)}) \leftarrow \mathrm{SHAKE}(\mathtt{0x96}\|\mathsf{seed}_{\mathbf{SE}}, (2\overline{m}n+\overline{mn})\cdot\mathsf{len}_\chi)$
5: $\mathbf{S}' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(\overline{m}n-1)}), \overline{m}, n, T_\chi)$
6: $\mathbf{E}' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(\overline{m}n)}, \mathbf{r}^{(\overline{m}n+1)}, \ldots, \mathbf{r}^{(2\overline{m}n-1)}), \overline{m}, n, T_\chi)$
7: $\mathbf{A} \leftarrow \mathsf{Frodo.Gen}(\mathsf{seed}_{\mathbf{A}})$
8: $\mathbf{B}' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
9: $\mathbf{c}_1 \leftarrow \mathsf{Frodo.Pack}(\mathbf{B}')$
10: $\mathbf{E}'' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(2\overline{m}n)}, \mathbf{r}^{(2\overline{m}n+1)}, \ldots, \mathbf{r}^{(2\overline{m}n+\overline{mn}-1)}), \overline{m}, \overline{n}, T_\chi)$
11: $\mathbf{B} \leftarrow \mathsf{Frodo.Unpack}(\mathbf{b}, n, \overline{n})$
12: $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
13: $\mathbf{C} \leftarrow \mathbf{V} + \mathsf{Frodo.Encode}(\mu)$
14: $\mathbf{c}_2 \leftarrow \mathsf{Frodo.Pack}(\mathbf{C})$
15: $\mathbf{ss} \leftarrow \mathrm{SHAKE}(\mathbf{c}_1\|\mathbf{c}_2\|\mathbf{k}, \mathsf{len_{ss}})$
16: **return** $\mathbf{c}_1\|\mathbf{c}_2$, $\mathbf{ss}$

Function: FrodoKEM.Decaps
Input: $\mathbf{c}_1\|\mathbf{c}_2 \in \{0,1\}^{(\overline{m}\cdot n+\overline{m}\cdot\overline{n})D}$, $sk' = (\mathbf{s}\|\mathsf{seed}_{\mathbf{A}}\|\mathbf{b}, \mathbf{S}^{\mathrm{T}}, \mathbf{pkh}) \in \{0,1\}^{\mathsf{len_s}+\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}+D\cdot n\cdot\overline{n}} \times \mathbb{Z}_q^{\overline{n}\times n} \times \{0,1\}^{\mathsf{len_{pkh}}}$.
Output: $\mathbf{ss} \in \{0,1\}^{\mathsf{len_{ss}}}$.
1: $\mathbf{B}' \leftarrow \mathsf{Frodo.Unpack}(\mathbf{c}_1, \overline{m}, n)$
2: $\mathbf{C} \leftarrow \mathsf{Frodo.Unpack}(\mathbf{c}_2, \overline{m}, \overline{n})$
3: $\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S}$
4: $\mu' \leftarrow \mathsf{Frodo.Decode}(\mathbf{M})$
5: $pk \leftarrow \mathsf{seed}_{\mathbf{A}}\|\mathbf{b}$
6: $\mathsf{seed}_{\mathbf{SE}}'\|\mathbf{k}' \leftarrow \mathrm{SHAKE}(\mathbf{pkh}\|\mu', \mathsf{len}_{\mathsf{seed}_{\mathbf{SE}}} + \mathsf{len_k})$
7: $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(2\overline{m}n+\overline{mn}-1)}) \leftarrow \mathrm{SHAKE}(\mathtt{0x96}\|\mathsf{seed}_{\mathbf{SE}}', (2\overline{m}n+\overline{mn})\cdot\mathsf{len}_\chi)$
8: $\mathbf{S}' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(\overline{m}n-1)}), \overline{m}, n, T_\chi)$
9: $\mathbf{E}' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(\overline{m}n)}, \mathbf{r}^{(\overline{m}n+1)}, \ldots, \mathbf{r}^{(2\overline{m}n-1)}), \overline{m}, n, T_\chi)$
10: $\mathbf{A} \leftarrow \mathsf{Frodo.Gen}(\mathsf{seed}_{\mathbf{A}})$
11: $\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
12: $\mathbf{E}'' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(2\overline{m}n)}, \mathbf{r}^{(2\overline{m}n+1)}, \ldots, \mathbf{r}^{(2\overline{m}n+\overline{mn}-1)}), \overline{m}, \overline{n}, T_\chi)$
13: $\mathbf{B} \leftarrow \mathsf{Frodo.Unpack}(\mathbf{b}, n, \overline{n})$
14: $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
15: $\mathbf{C}' \leftarrow \mathbf{V} + \mathsf{Frodo.Encode}(\mu')$
16: (in constant time) $\overline{\mathbf{k}} \leftarrow \mathbf{k}'$ if $(\mathbf{B}'\|\mathbf{C} = \mathbf{B}''\|\mathbf{C}')$ else $\overline{\mathbf{k}} \leftarrow \mathbf{s}$
17: $\mathbf{ss} \leftarrow \mathrm{SHAKE}(\mathbf{c}_1\|\mathbf{c}_2\|\overline{\mathbf{k}}, \mathsf{len_{ss}})$
18: **return** $\mathbf{ss}$

Algorithm 2 Frodo.Gen using AES128 (algorithm taken from [ABD+20]).

Input: Seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$.
Output: Matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$.
1: **for** $(i = 0; i < n; i \leftarrow i + 1)$ **do**
2: **for** $(j = 0; j < n; j \leftarrow j + 8)$ **do**
3: $\mathbf{b} \leftarrow \langle i \rangle \| \langle j \rangle \| 0 \cdots 0 \in \{0,1\}^{128}$ where $\langle i \rangle, \langle j \rangle \in \{0,1\}^{16}$
4: $\langle c_{i,j} \rangle \| \langle c_{i,j+1} \rangle \| \cdots \| \langle c_{i,j+7} \rangle \leftarrow \mathrm{AES128}_{\mathsf{seed}_{\mathbf{A}}}(\mathbf{b})$ where each $\langle c_{i,k} \rangle \in \{0,1\}^{16}$
5: **for** $(k = 0; k < 8; k \leftarrow k + 1)$ **do**
6: $\mathbf{A}_{i,j+k} \leftarrow c_{i,j+k} \bmod q$
7: **return** $\mathbf{A}$

Algorithm 3 Frodo.Gen using SHAKE128 (algorithm taken from [ABD+20]).

Input: Seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$.
Output: Pseudorandom matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$.
1: **for** $(i = 0; i < n; i \leftarrow i + 1)$ **do**
2: $\mathbf{b} \leftarrow \langle i \rangle \| \mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{16 + \mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$ where $\langle i \rangle \in \{0,1\}^{16}$
3: $\langle c_{i,0} \rangle \| \langle c_{i,1} \rangle \| \cdots \| \langle c_{i,n-1} \rangle \leftarrow \mathrm{SHAKE128}(\mathbf{b}, 16n)$ where each $\langle c_{i,j} \rangle \in \{0,1\}^{16}$.
4: **for** $(j = 0; j < n; j \leftarrow j + 1)$ **do**
5: $\mathbf{A}_{i,j} \leftarrow c_{i,j} \bmod q$
6: **return** $\mathbf{A}$

addition. The public key pk is derived from $\mathbf{B}$ and $\mathsf{seed}_{\mathbf{A}}$, while the secret key contains $\mathbf{S}$. The error matrix $\mathbf{E}$ is not part of any key and is discarded.

A similar computation occurs in FrodoPKE.Enc, where positions in the multiplication are swapped. A public matrix $\mathbf{B}' \in \mathbb{Z}_q^{\overline{n} \times n}$ is computed from the same matrix $\mathbf{A}$ as $\mathbf{B}' = \mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$ where the matrices $\mathbf{S}', \mathbf{E}' \in \mathbb{Z}_q^{\overline{n} \times n}$ are generated by sampling from χ. This is followed by another matrix multiplication to compute $\mathbf{V} \in \mathbb{Z}_q^{\overline{n} \times \overline{n}}$ as $\mathbf{V} = \mathbf{S}' \cdot \mathbf{B} + \mathbf{E}''$ where $\mathbf{E}'' \in \mathbb{Z}_q^{\overline{n} \times \overline{n}}$ is also sampled from χ. Since encryption is a subroutine of both encapsulation and decapsulation, these two matrix computations occur in both FrodoKEM.Encaps and FrodoKEM.Decaps. Finally, decapsulation is preceded by a matrix computation, computing $\mathbf{M} \in \mathbb{Z}_q^{\overline{n} \times \overline{n}}$ as $\mathbf{M} = \mathbf{C} - \mathbf{B}' \cdot \mathbf{S}$ where $\mathbf{C} \in \mathbb{Z}_q^{\overline{n} \times \overline{n}}$ is derived from the ciphertext.

2.1.2 Generation of the Public Matrix A.

Following previous work in this area [ADPS16, BCD+16], the public matrix $\mathbf{A}$ is generated dynamically and pseudorandomly for every generated key. This helps to avoid the possibility of backdoors and all-for-the-price-of-one attacks [ABD+15].

Let us recall how the matrix $\mathbf{A}$ is constructed following the FrodoKEM specification [ABD+20] since this will be relevant for the memory reduction techniques in this paper. The algorithm FrodoKEM.Gen takes as input the modulus q, a seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$ and a dimension $n \in \mathbb{Z}$, and outputs a pseudorandom matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$. There are two options for instantiating FrodoKEM.Gen. The first method uses AES128, the second instead uses SHAKE128.

When using AES128, the matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ is generated 8 elements at-a-time (of two bytes each). For each row and each block of 8 elements (in different columns), the algorithm generates a 128-bit block of predefined input based on the location in the matrix. This input is encrypted using the $\mathsf{seed}_{\mathbf{A}}$ as the AES128 key. This process is outlined in Algorithm 2. More specifically, the input blocks to AES128 are $\langle i \rangle \| \langle j \rangle \| 0 \| \cdots \| 0 \in \{0,1\}^{128}$, where i, j are encoded as 16-bit integers (see Line 3). It then splits the 128-bit AES128 output block into eight 16-bit elements, which it interprets as non-negative integers $c_{i,j+k}$

for $k = 0, 1, \ldots, 7$ (see Line 4). Finally, it sets $\mathbf{A}_{i,j+k} = c_{i,j+k} \bmod q$ for all k. Since q is always a power of two, this modular reduction is "for free" by dropping the most significant bits whenever $q < 2^{16}$ (e.g. by applying a bitmask).

The second method uses SHAKE128 instead of AES128 to generate the rows of the matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$. This process is shown in Algorithm 3. In this case, each entire row is generated with a SHAKE128 call: this reduces function call overhead compared to the AES128 approach (assuming the entire output can be stored). Its input consists of the row index encoded as a 16-bit integer, followed by $\mathsf{seed}_{\mathbf{A}}$ to produce a $16n$-bit output (see Line 3). The output is then split into 16-bit integers $c_{i,j} \in \{0,1\}^{16}$ (for $j = 0, 1, \ldots, n-1$), and used to set the corresponding matrix entries $\mathbf{A}_{i,j} = c_{i,j} \bmod q$ in Line 5.

2.1.3 Generation of Secret and Error Matrices

The secret and error matrices $\mathbf{S}$ and $\mathbf{E}$ in key generation, and their counterparts $\mathbf{S}'$, $\mathbf{E}'$ and $\mathbf{E}''$ in encapsulation and decapsulation are sampled from the distribution χ. Sampling from χ is done via a constant-time lookup table, using an inversion sampling technique. To do sampling, a random bit string of length 16 denoted by $\mathbf{r}^{(\cdot)}$ is required per coefficient in the matrix. These random bit strings are first generated using SHAKE128 (in FrodoKEM-640) or SHAKE256 (in FrodoKEM-976 and FrodoKEM-1344), by using a fixed prefix and a freshly generated random seed as input. In FrodoKEM.KeyGen, $2n\overline{n} \cdot 16$ bits are extracted via SHAKE, resulting in bit strings $(\mathbf{r}^{(0)}, \ldots \mathbf{r}^{(2n\overline{n}-1)})$. The first $n\overline{n}$ bit strings are used to sample the rows of $\mathbf{S}^{\mathrm{T}}$, while the next $n\overline{n}$ bit strings are used to sample the rows of $\mathbf{E}$. Similarly in encapsulation and decapsulation, $2\overline{n}n + \overline{n}\overline{n}$ bit strings are generated, which are used to sample the rows of $\mathbf{S}'$, $\mathbf{E}'$ and $\mathbf{E}''$ respectively.

3 Stack Reduction Strategies

The main contributor to the stack usage in key generation, encapsulation and decapsulation is the (temporary) storage of large matrices. These matrices include $\mathbf{B}, \mathbf{E}$ and $\mathbf{S}$ in the key generation, $\mathbf{B}', \mathbf{S}', \mathbf{E}'$ and $\mathbf{B}$ in encapsulation and $\mathbf{B}', \mathbf{S}, \mathbf{S}', \mathbf{E}', \mathbf{B}''$ and $\mathbf{B}$ in decapsulation. All these matrices are of size $n \times \bar{n}$ or $\bar{n} \times n$ and each entry is 2 bytes large, resulting in a size of 10 240, 15 616 and 21 504 bytes for FrodoKEM-640, FrodoKEM-976 and FrodoKEM-1344 respectively. In this section we describe our main strategies of reducing stack usage. We first focus on the AES128 versions, since this allows for more independence of how elements within $\mathbf{A}$ are generated. This in turn gives more flexibility for matrix-multiplication techniques. We highlight the main differences of these optimization techniques for the SHAKE versions in Section 3.5, for which we also propose a change to FrodoKEM.Gen in the current specification to simplify the implementation.

Matrix Multiplication Strategies. A multiplication of matrices can be seen as a collection of inner-products of two arrays, which boils down to a repeated multiplication with accumulation. Optimizing the speed of matrix multiplications has been extensively studied, including for the specific case of those in FrodoKEM. For example, in [BOR+21] the matrix multiplications in FrodoKEM are sped up via various cache-friendly techniques [HSHvdG16] as well as asymptotically faster algorithms such as Strassen [Str69]. This assumes, however, that the matrices can be stored in full. For resource constrained devices this assumption is often not valid, thus we do not consider these techniques here. Furthermore, we leave other advanced matrix tricks such as decomposition techniques out of scope, since these methods are not well-defined when the entries of the matrices are defined over a finite field.

As a starting point we consider the matrix multiplication strategies introduced in [BFM+18], who target the Cortex-M4 architecture. The first strategy is a straight-forward inner-product, which assumes a row of the left operand and a column of the right operand

are stored in full. More importantly, it assumes the coefficients are stored in memory as a contiguous array in order to use special instructions to load multiple values at once. This occurs naturally when multiplying with $\mathbf{S}$ on the right since it is sampled as $\mathbf{S}^{\mathrm{T}}$, putting the columns contiguous in memory. The inner-product of these arrays is then computed by loading multiple values of each operand into registers as halfwords, and applying a multiplication with accumulation of halfwords.

Whenever the coefficients of the right operand are not contiguous in memory, a second strategy is used that is referred to as the row-by-chunk method. This is the case when the rows of the right operand are contiguous in memory, thus the column elements have a fixed offset in memory. In this case, loading the coefficients of the right operand requires additional operations to collect them from different memory locations, and to extract the 16-bit halfwords from 32-bit memory loads. As a result only 8 coefficients can be loaded into registers at a time. Since it is more expensive to load the right operand, it is more efficient to process all row elements of the left operands whenever 8 values of the right operand are in registers.

Note that the two strategies are efficiently implemented by relying on SIMD instructions. In this section, we modify these two strategies depending on what and how parts of matrices are stored.

3.1 Key Generation

Matrix Operations. The matrix operation taking place in key generation is $\mathbf{B} = \mathbf{A} \cdot \mathbf{S} + \mathbf{E}$. Since $\mathbf{S}$ is generated by sampling $\mathbf{S}^{\mathrm{T}}$ row-by-row, the columns of $\mathbf{S}$ are naturally already contiguous in memory. The public matrix $\mathbf{A}$ is generated row-wise independent of the symmetric primitive used. Therefore, the inner-product subroutine can be applied in a straight-forward manner. Modifications to this method can be made by eliminating the requirement to store rows and columns in full and only storing a constant number of values.

Generating A On-The-Fly. The biggest pressure on the memory in key generation is the public matrix $\mathbf{A}$. To reduce the stack memory usage this can be generated on-the-fly as opposed to storing it in full. Indeed, this is recommended by the FrodoKEM team and already included in the provided optimized implementation. This strategy offers several time-memory trade-offs.

The strategy in the official FrodoKEM.KeyGen implementation stores and computes $t_k = 4$ rows of $\mathbf{A}$ simultaneously where t_k is a trade-off parameter. This enables to compute 4 inner products each time a column of $\mathbf{S}$ is available, allowing the use of the single instruction, multiple data paradigm when available. In such an implementation of matrix multiplication, the memory usage for $\mathbf{A}$ is proportional to the number of its simultaneously available coefficients $(n \cdot t_k)$. As a result, the stack consumption can be reduced by taking $t_k = 1$, hence generating a single row at the time at the cost of performance overheads as detailed in the next paragraph.

Furthermore, when AES128 is used to generate a row of $\mathbf{A}$, $2n$-byte arrays for both the input and the output are used. The motivation for this is that for every iteration of a row of $\mathbf{A}$, only the row indices need to be updated of the input array. Instead, we can re-use the same array for both the input and output (saving $2n$ bytes), at the cost of having to reconstruct the input array for every iteration.

Reducing Storage for S. The matrix $\mathbf{S}$ of size $n \times \overline{n}$ also represents a significant part of the memory consumption if stored in full. The values in $\mathbf{S}$ are sampled from the distribution χ which has a support of size $\{25, 21, 13\}$ (i.e. the number of values with a non-zero probability) when using $n \in \{640, 976, 1344\}$: hence, only 5, 5 and 4 bits are needed to store the values for the respective parameter sets. For convenience, these coefficients

can be stored in a two-byte representation (as done in all previous implementations). In this way, the values are sampled from 2-byte values $\mathbf{r}^{(\cdot)}$ and later also multiplied with 2-byte values. To reduce memory consumption, we instead compress the values of $\mathbf{S}$ down to whole bytes or nibbles: using an 8-, 8- and 4-bit representation respectively depending on the parameter set. This reduces the stack usage by 5 120, 7 808, and 16 128 bytes respectively for the full matrix $\mathbf{S}$.

Computing S On-The-Fly. Alternatively, we detail the trade-offs for an on-the-fly generation of $\mathbf{S}$ in the matrix multiplication $\mathbf{A} \cdot \mathbf{S}$ for the parameter t_k. First if both $\mathbf{S}$ and $\mathbf{A}$ are generated on-the-fly, then $\mathbf{S}$ would need to be recomputed exactly n times if a single row of $\mathbf{A}$ is stored in memory ($t_k = 1$). As the number of times $\mathbf{S}$ needs to reconstructed is equal $\frac{n}{t_k}$, storing more rows of $\mathbf{A}$ leads to faster (but larger) implementations. One can optimize the storage of $\mathbf{S}$ even further: instead of storing a full row at once, process 10 coefficients at-a-time (this is the maximum number of elements that can be processed in registers simultaneously on Cortex-M4). This multiplication is depicted in Figure 1a. Computing $\mathbf{S}$ on-the-fly reduces its storage from $2n\overline{n}$ bytes down to 20 bytes.

Computing B On-The-Fly. Another significant improvement in memory usage in key generation can be achieved by reducing the storage of the matrix $\mathbf{B}$. In all current implementation strategies, $\mathbf{B}$ is generated and stored in full. At the end of key generation, the matrix $\mathbf{B}$ is packed and written to the output as well as hashed as part of the secret key for the public-key hash **pkh**.

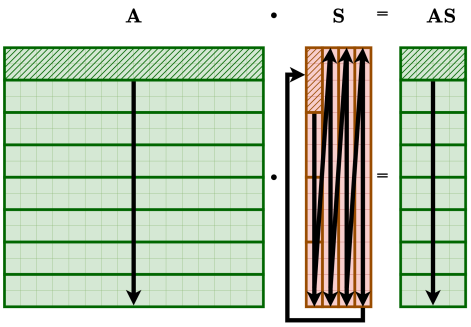
Let us describe a memory reduction technique using the on-the-fly pack-and-hash method. Instead of first completely storing $\mathbf{B}$, we propose to directly pack and hash coefficients of $\mathbf{B}$ as soon as they are produced. In order to be compliant with FrodoKEM specifications, this strategy ensures that that the elements of $\mathbf{B}$ are computed row-wise. Concretely, we generate t_k rows of $\mathbf{B}$ by leveraging the matrix multiplication on-the-fly strategy. These coefficients are then packed (when $q = 2^{15}$), written to the output buffer, absorbed into the SHAKE state and then one moves to the following coefficients. Note that we absorb $\mathsf{seed}_{\mathbf{A}}$ before we start processing $\mathbf{B}$, and squeeze out **pkh** after processing all of $\mathbf{B}$. This technique reduces the storage of $\mathbf{B}$ from $2n\overline{n}$ bytes down to $2t_k\overline{n}$ bytes.

This strategy completely eliminates the need for a memory buffer for $\mathbf{B}$. Hence, one cannot use the memory allocated to $\mathbf{B}$ as scratch space to store $\mathbf{E}$ as done in the implementation used in pqm4. We overcome this by storing the appropriate SHAKE state from which the values of $\mathbf{E}$ can be generated on-the-fly instead.

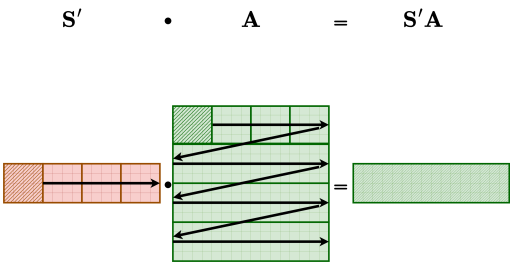
3.2 Encapsulation

Matrix Multiplication. The main memory consuming computations in encapsulation are the matrix operations to compute $\mathbf{B}' = \mathbf{S}'\mathbf{A} + \mathbf{E}'$ and $\mathbf{V} = \mathbf{S}'\mathbf{B} + \mathbf{E}''$. Note that the public matrix $\mathbf{A}$ is the right operand in the computation of $\mathbf{B}'$. Since this is generated row-wise and the elements are addressed column-wise for $\mathbf{S}'\mathbf{A}$, the layout in memory is not contiguous for this arithmetic operation. The same observation holds for the matrix $\mathbf{B}$ in the computation of $\mathbf{V}$. Therefore the row-by-chunk strategy is required for the multiplications in encapsulation. Modifications to this method are made depending on how many elements or rows of the matrix $\mathbf{S}'$ can be stored.

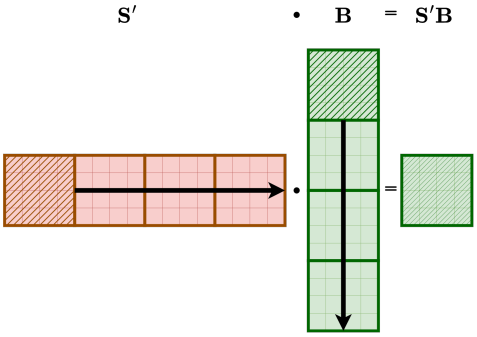
Generating A and S′ On-The-Fly. Since we need to compute inner-products with columns of $\mathbf{A}$, we use the row-by-chunk multiplication method that operates on eight values in a column of $\mathbf{A}$ at once. Because AES128 generates eight values in a row, one now stores an 8×8 block of $\mathbf{A}$. This implies one needs to store at least eight values in a row of $\mathbf{S}'$, but to avoid re-computation of any matrix we also store the eight values of every $\overline{n}$



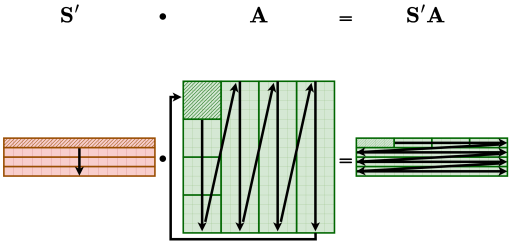
(a) Matrix-multiplication to compute the product $\mathbf{A} \cdot \mathbf{S}$ on-the-fly.



(b) Matrix-multiplication to compute the product $\mathbf{S}' \cdot \mathbf{A}$ on-the-fly, with storing result in full.



(c) Matrix-multiplication to compute the product $\mathbf{S}' \cdot \mathbf{B}$ on-the-fly.



(d) Matrix-multiplication to compute the product $\mathbf{S}' \cdot \mathbf{A}$, without storing result in full.

Figure 1: A summary of the matrix multiplication techniques that we use for the various operations in FrodoKEM.

rows, resulting also in an 8×8 block for $\mathbf{S}'$. With a block of both $\mathbf{S}'$ and $\mathbf{A}$, all $8 \cdot 8 = 64$ size-8 inner-products are computed and accumulated to the respective values in $\mathbf{B}'$, which is initialized as $\mathbf{E}'$. For every block of $\mathbf{S}'$, eight rows of $\mathbf{A}$ need to be processed, meaning that we go through eight rows of $\mathbf{A}$ as blocks: this strategy is depicted in Figure 1b.

Note that $\mathbf{S}'$ is generated row-wise from a single seed via SHAKE, thus one cannot immediately generate an 8×8 block from a single state. In order to generate consecutive blocks, one needs to prepare eight SHAKE states first. This is done by running through all of $\mathbf{S}'$ once with an incremental SHAKE state, and store the state every time it reaches the start of a row. The values of $\mathbf{S}'$ are not computed at this point. To generate a new 8×8 block, each stored SHAKE state is used to squeeze out eight values (i.e., eight values of two bytes $\mathbf{r}^{(\cdot)}$ which are then used to sample from χ). Since $\mathbf{S}'$ is not fully stored, but needed again to compute $\mathbf{V}$, one can compute both $\mathbf{B}'$ and $\mathbf{V}$ in parallel. This is done by also preparing an 8×8 block of $\mathbf{B}$ for every block of $\mathbf{S}$, as depicted in Figure 1c. This block of $\mathbf{B}$ is obtained by unpacking the corresponding coefficients from $\mathbf{b}$ which is part of the public key. Again, with the row-by-chunk method all $8 \cdot 8 = 64$ size-8 inner-products between the blocks of $\mathbf{S}'$ and $\mathbf{B}$ are computed and accumulated to $\mathbf{V}$, which initially contains $\mathbf{E}''$. After $\mathbf{B}'$ is computed it has to be packed to $\mathbf{c}_1$, and to avoid having to fully store $\mathbf{c}_1$ we apply the on-the-fly pack and hash method.

In previous implementations, 4 rows of $\mathbf{A}$ were stored, requiring $8n$ bytes. This is now reduced to a block of 128 bytes. The storage of matrices $\mathbf{S}'$ and $\mathbf{B}$ are reduced from $2\overline{n}n$ bytes each down to 128 bytes each.

Generating $\mathbf{B}'$ On-The-Fly. With $\mathbf{A}$ and $\mathbf{S}'$ generated as 8×8 blocks, the remaining memory allocation mainly consists of storing the matrix $\mathbf{B}'$. Similar to $\mathbf{B}$ in FrodoKEM.KeyGen, the on-the-fly pack-and-hash method is applicable with the constraint that $\mathbf{B}'$ is computed row-wise and is directly used as input to SHAKE. Again, if both inputs are not fully stored, the right operand has to be recomputed, which is $\mathbf{A}$ in this case. Concretely, as described in Figure 1d, a single row of $\mathbf{S}'$ is stored and the columns of $\mathbf{A}$ are generated as 8×8 blocks. One can then sequentially multiply eight columns of $\mathbf{A}$ as 8×8 blocks, after which the corresponding eight values of $\mathbf{E}'$ are added. This results in a row chunk of 8 coefficients of $\mathbf{B}'$, which is packed and absorbed to a SHAKE state for the shared secret $\mathbf{ss}$. This process needs to be repeated for every row of $\mathbf{S}$, and therefore all of $\mathbf{A}$ is need to be re-computed $\overline{n}$ times.

The re-computation of $\mathbf{A}$ gives performance overheads: one can perform a time-memory tradeoff using the parameter t_{ed}. Storing $t_{ed}+1$ rows of $\mathbf{S}'$ means that $n(t_{ed}+1)$ coefficients must be stored: however, this reduces the number of re-computations of $\mathbf{A}$ down to $\frac{\overline{n}}{t_{ed}+1}$. It is important to note that the first row of $\mathbf{B}'$ can be directly packed and hashed, but this is not the case for the t_{ed} extra ones as $\mathbf{B}'$ must be hashed row-wise. As a result, when this trade-off is used, a full row of $\mathbf{B}'$ needs to be stored for every of the t_{ed} extra rows stored in $\mathbf{S}'$. This reduces the storage of $\mathbf{B}'$ from $2\overline{n}n$ bytes down to $16 + 2t_{ed}n$ bytes. Note that the rows of $\mathbf{B}'$ can also be stored packed. This saves only 80 bytes for FrodoKEM-640 and nothing for the other parameter sets. Hence, we do not consider this optimization further.

With this method it is no longer beneficial to compute $\mathbf{B}'$ and $\mathbf{V}$ in parallel. Indeed, for every a row of $\mathbf{S}'$, a column of $\mathbf{B}$ needs to be accessed. Since $\mathbf{B}$ is packed row-wise in the public key, such access pattern is not trivial. Additionally, for every row of $\mathbf{S}'$, $\mathbf{B}$ must be fully loaded. Therefore $\mathbf{V}$ is computed after $\mathbf{B}'$, and the block method as in Figure 1c is used. The computation of $\mathbf{V}$ can now re-use memory from the computation of $\mathbf{B}'$.

3.3 Decapsulation

Matrix Multiplication. Since decapsulation includes a re-encryption, FrodoKEM.Decaps benefits from the same optimizations as in FrodoKEM.Encaps. For computations preceding the re-encryption step, the main memory consuming computation consists of the matrix operation $\mathbf{M} = \mathbf{C} - \mathbf{B}'\mathbf{S}$. This is what we focus on in this subsection.

Generating $\mathbf{B}'$ On-The-Fly. The first option is to fully store both the matrices $\mathbf{M}$ and $\mathbf{S}$ (with $\mathbf{S}$ being copied from the secret key). One can re-use the memory to later store the matrices $\mathbf{V}$ and $\mathbf{B}''$, respectively. Additionally, up to a full row of $\mathbf{B}'$ is unpacked from $\mathbf{c}_1$ and stored. With the availability of full rows of $\mathbf{S}^{\mathrm{T}}$ and $\mathbf{B}'$ one can simply compute an inner-product to compute $\mathbf{M}$.

After the re-encryption step is completed, one still needs to compare $\mathbf{B}'$ with $\mathbf{B}''$. When this strategy is used, a memory block is available for the entire matrix $\mathbf{B}''$. Hence, we leverage a similar approach as for FrodoKEM.Encaps as described in Figure 1b. Unfortunately, $\mathbf{B}'$ is not available in memory hence one needs to (in an on-the-fly manner) unpack a single row of $\mathbf{B}'$ from $\mathbf{c}_1$ and compare this to its respective part of $\mathbf{B}''$. Using these techniques, the stack usage of decapsulation matches the requirement of the encapsulation because the same memory blocks can be re-used, except for a small caveat for the FrodoKEM-1344 parameter set. Here 64 bytes more are needed for the storage of a row of $\mathbf{B}'$ as this becomes the memory bottleneck of the subroutines.

Generating $\mathbf{B}''$ On-The-Fly. For decapsulation we can generate $\mathbf{B}''$ on-the-fly by using the same strategy as in encapsulation at the cost of re-computation of $\mathbf{A}$ (see Figure 1d). This means this memory block of $\mathbf{B}''$ cannot be shared with $\mathbf{S}$. As a result, the computation of $\mathbf{M}$ is performed with the block technique similar as for the generation of $\mathbf{V}$ in

FrodoKEM.Encaps (Figure 1c). Note however, that now a straight-forward inner-product of size eight can be used instead of the row-by-chunk method. This only requires an 8×8 block of both $\mathbf{S}$ and $\mathbf{B}$ to be stored.

In this case neither $\mathbf{B}'$ nor $\mathbf{B}''$ are stored, which need to be compared for equality at the end of the decapsulation as part of the FO transform. This can be done by loading in the necessary respective coefficients of $\mathbf{B}'$ on-the-fly whenever the analogous coefficients of $\mathbf{B}''$ are computed. It is important that the computation continues whenever inequality is encountered, since the comparison has to be performed in constant time. This is fairly straightforward by maintaining a flag that signals whether any coefficients have been different, and using this flag to reject only after all coefficients have been compared. A similar approach can be taken to compare $\mathbf{C}$ against $\mathbf{C}'$.

Alternatively, we can compare *hashes* of the packed matrices using SHAKE. The packed matrices $(\mathbf{B}''\|\mathbf{C}')$ are absorbed on-the-fly during their computation, while the hash of the matrices $(\mathbf{B}'\|\mathbf{C})$ is computed alongside the computation of the shared secret. This latter part is done by copying the SHAKE state after absorbing $\mathbf{c}_1\|\mathbf{c}_2$, where one copy is finalized and the comparison hash is squeezed out, while the other state first gets $\overline{\mathbf{k}}$ absorbed and then the shared secret is derived from it by finalizing and squeezing from the state. This strategy leads to a little additional memory to maintain the SHAKE state, and a computational trade-off between additional memory loads for $\mathbf{B}'$ and $\mathbf{C}$, and SHAKE absorbs. The comparison is not exact and equality is only guaranteed up to collision resistance of SHAKE (SHAKE128 for FrodoKEM-640 and SHAKE256 for FrodoKEM-976 and FrodoKEM-1344). We take the former of the two approaches in our implementation, though the difference is performance is minor.

3.4 Implementation Strategies

We introduced several memory reduction techniques in Sections 3.1–3.3, but it remains to decide which ones to select for implementation. In this section we present various implementation strategies for all parameter sets of FrodoKEM. We first give details of a low-cost stack-optimized implementation strategy. In this strategy we select the techniques that significantly reduce the stack usage, while the performance (i.e., runtime) does not increase significantly. This implementation is characterized by still storing one large (i.e. of size $n \times \overline{n}$ or $\overline{n} \times n$) matrix throughout the algorithm.

For a more drastic reduction of the stack usage we provide a second memory reduction strategy, which offers specific time-memory trade-offs based on a selected target. This strategy is characterized by not storing any large matrix in full, which comes at the cost of re-computations.

Both strategies are further optimized by optimizing placement of memory blocks. Any variables that do not have an overlapping lifetime will share a memory block. To fully utilize this method, a single block of memory is allocated for all subroutines. This is then passed to all subroutines but also used to store short-lifetime variables in the main function. This requires careful optimization of the various routines. We show an example for FrodoKEM-AES-640.KeyGen in Figure 2.

In Table 2 we show our results by breaking down the memory usage of every optimization technique. We compare our two implementation with the pqm4 implementation to highlight the differences.

Low-Cost Memory Reduction. In key generation, both matrices $\mathbf{A}$ and $\mathbf{B}$ can be processed on-the-fly instead of storing them fully, without any need for re-computation of other matrices. This is *not* the case for the matrix $\mathbf{S}$, which we store entirely. Note that it is possible to reduce the stack usage of key generation even further, by reducing the storage for $\mathbf{S}$, as explained in Section 3.1. We opted not to implement it since key generation is already in the same range of encapsulation and decapsulation.

Table 2: Breakdown of stack memory per optimization technique, measured in bytes.

Implementation	**pqm4**[1]	**Ours (Low-Cost)**	**Ours (Trade-Off)**
		FrodoKEM-640.KeyGen (AES)	
A	10,240	1,280	$1{,}280 \cdot t_k$
B, **E**	10,240	16	$16 \cdot t_k$
S	10,240	10,240	20
Total[2]	31,489	12,448	$1{,}556 + 1{,}296 \cdot t_k$
		FrodoKEM-640.Encaps (AES)	
A	20,480	128	128
B	10,240	$128^\dagger$	$128^\dagger$
V, $\mathbf{E}''$	128	128	$128^\dagger$
C	128	$128^\dagger$	$128^\dagger$
$\mathbf{B}'$, $\mathbf{E}'$	10,240	10,240	$16 + 1{,}280 \cdot t_{ed}$
$\mathbf{S}'$	10,240	$128 + 1{,}664$	$1{,}280 \cdot (t_{ed} + 1) + 1{,}664^\dagger$
c	9,720	$15^\dagger + 120^\dagger$	$15^\dagger + 120^\dagger$
Total[2]	61,977	13,024	$2{,}801 + 2{,}560 \cdot t_{ed}$
		FrodoKEM-640.Decaps (AES)	
A	20,480	128	128
B	10,240	$128^\dagger$	$128^\dagger$
$\mathbf{B}'$	10,240	$1{,}280^\dagger$	$128^\dagger$
S	10,240	10,240	$128^\dagger$
M, **V**, $\mathbf{E}''$	128	$128^\dagger$	$128^\dagger$
C	128	$128^\dagger$	$128^\dagger$
$\mathbf{C}'$	128	$128^\dagger$	$128^\dagger$
$\mathbf{B}''$, $\mathbf{E}'$	10,240	$10{,}240^\dagger$	$16 + 1{,}280 \cdot t_{ed}$
$\mathbf{S}'$	10,240	$128 + 1{,}664$	$1{,}280 \cdot (t_{ed} + 1) + 1{,}664^\dagger$
c	9,720	$15^\dagger + 120^\dagger$	$15^\dagger + 120^\dagger$
Total[2]	82,585	13,024	$2{,}593 + 2{,}560 \cdot t_{ed}$

[1] https://github.com/mupq/pqm4/tree/master
commit: `685fbbb4059b882cad00f3fb4a345bf36b37ef4b`

[2] This also includes smaller variables such as seeds, but also stack usage of the symmetric subroutines (i.e. AES and SHAKE)

[†] These variables do not increase peak stack usage in this implementation by reusing memory from previous variables

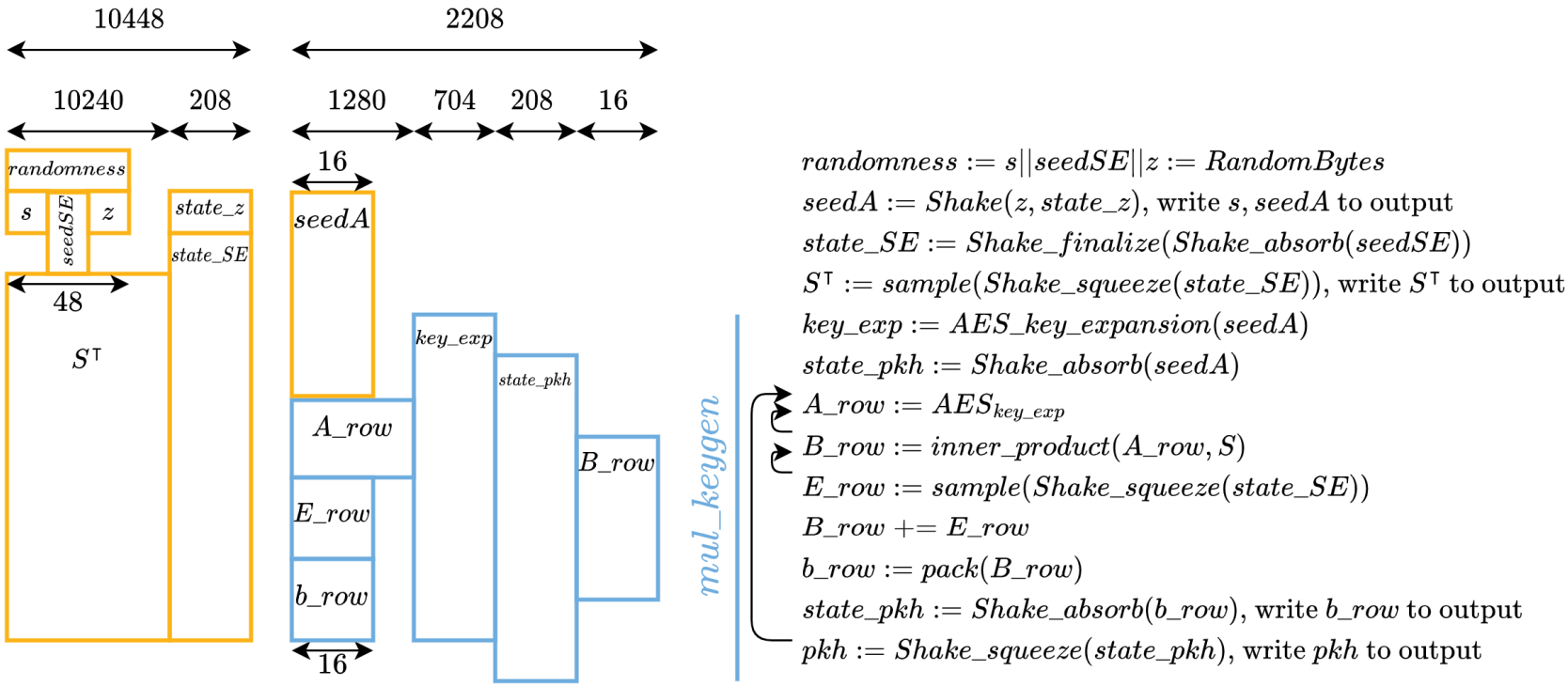


Figure 2: Memory allocation in bytes for the FrodoKEM-AES-640.KeyGen algorithm.

During encapsulation avoiding re-computation means having to store the entire matrix $\mathbf{B}'$. Both $\mathbf{A}$ and $\mathbf{S}'$ can be generated on-the-fly by only storing an 8×8 block, requiring 128 bytes for each block. This will result in a small overhead in runtime to construct the additional SHAKE states of $\mathbf{S}'$, as explained in Section 3.2.

Decapsulation uses the same strategy as in encapsulation for its re-encryption subroutine. For FrodoKEM-640-AES, FrodoKEM-976-AES, the re-encryption preceding computations require no additional stack allocation compared to encapsulation since the encapsulation stack is the bottleneck (whereas FrodoKEM-1344-AES uses an additional 64 bytes as explained in Section 3.3).

Time-Memory Trade-Offs. If further memory reduction is required, re-computation of certain matrices is required. The second strategy leverages time-memory tradeoffs.

In key generation, we choose to compute the matrix $\mathbf{S}$ on-the-fly and re-compute when needed. We select the trade-off parameter t_k as a divisor of n for convenience. Increasing t_k increases the number of stored rows of $\mathbf{A}$ (at the cost of $2n$ bytes per row) as well as the stored rows of $\mathbf{B}$ (at the cost of 16 bytes per row).

Similar trade-offs exist in encapsulation, where one can process $\mathbf{B}'$ on-the-fly instead of storing the full matrix. Depending on the stack available, the trade-off parameter t_{ed} can be increased to store additional rows of $\mathbf{S}'$ (at the cost of $2n$ bytes per row) and $\mathbf{B}'$ (at the cost of $2n$ bytes per row), which reduces the number of required re-computations of $\mathbf{A}$. For convenience of implementation, we only consider the values $t_{ed} \in \{0, 1, 3, 7\}$.

When applying the same strategies for the re-encryption in decapsulation, there is less stack available for the preceding computation of $\mathbf{M}$. This is solved by using the strategy of loading both $\mathbf{S}$ and $\mathbf{B}$ on the fly as blocks, as explained in Section 3.3. Similar stack issues arise for the comparison after the re-encryption step of the decapsulation. By using the on-the-fly comparing strategy as described in Section 3.3, storing the entire matrices can be avoided. In contrast to encapsulation, during the on-the-fly computation of $\mathbf{B}''$, no hash of it has to be computed. Hence, in total decapsulation requires roughly 200 bytes less (i.e. one SHAKE state).

3.5 FrodoKEM with SHAKE: Efficient Matrix Generation

Finally, we highlight a significant difference between the FrodoKEM parameter sets using AES128 and SHAKE128. Whereas when generating $\mathbf{A}$ using AES128, each eight values in a row are generated independently from one another, when using SHAKE128 instead a complete row is squeezed from a single seed (see Algorithm 3). In high-performance settings this has the advantage that it reduces SHAKE128 function call overhead, but in low memory implementations it comes at the cost of multiple re-computations.

For example, in the memory reduction technique for FrodoKEM.Encaps illustrated in Figure 1b, the rows of $\mathbf{A}$ are computed in 8×8 blocks. This requires coefficients from eight rows at once, which can be achieved by initializing the eight corresponding SHAKE states and squeezing coefficients when needed. This gives significant memory overheads since storing a SHAKE state requires at least 200 bytes, hence this method uses at least 1400 bytes more than using only a single state.

Another example of the drawback of squeezing the full rows of $\mathbf{A}$ from a single SHAKE128 state is illustrated by the strategy Figure 1d. There $\mathbf{A}$ is accessed column-wise with 8×8 blocks, hence the impact is even larger. Indeed, the first block contains values from row 0–7 and the first eight columns while the second block we process contains values from row 8–15 and still the first eight columns. As a result, in order to access the block containing the coefficient $\mathbf{A}_{i,j}$, for each of the 8 rows, a SHAKE state must be initialized according to the row i and then squeezed to obtain all the coefficients j' such that $j' < j$ despite not being used in this block. In the worst case, to generate coefficients in the right-most columns of $\mathbf{A}$, the full rows must be generated from a SHAKE state. This is

Algorithm 4 Frodo.Gen using SHAKE128 using customization

Input: Seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$.
Output: Matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$.

1: $n' \leftarrow 84 \cdot \lfloor \frac{n}{84} \rfloor$
2: $n'' \leftarrow n \mod 84$
3: **for** $(i = 0; i < n; i \leftarrow i + 1)$ **do**
4: **for** $(j = 0; j < n'; j \leftarrow j + 84)$ **do**
5: $\mathbf{b} \leftarrow \langle i \rangle \| \langle j \rangle \| \mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{32+\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$ where $\langle i \rangle, \langle j \rangle \in \{0,1\}^{16}$
6: $\langle c_{i,j} \rangle \| \langle c_{i,j+1} \rangle \| \cdots \| \langle c_{i,j+83} \rangle \leftarrow \text{SHAKE128}(\mathbf{b}, 16 \cdot 84)$ where each $\langle c_{i,k} \rangle \in \{0,1\}^{16}$
7: $\mathbf{b} \leftarrow \langle i \rangle \| \langle n' \rangle \| \mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{32+\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$ where $\langle i \rangle, \langle n' \rangle \in \{0,1\}^{16}$
8: $\langle c_{i,n'} \rangle \| \langle c_{i,n'+1} \rangle \| \cdots \| \langle c_{i,n-1} \rangle \leftarrow \text{SHAKE128}(\mathbf{b}, 16n'')$ where each $\langle c_{i,k} \rangle \in \{0,1\}^{16}$
9: **for** $(j = 0; j < n; j \leftarrow j + 1)$ **do**
10: $\mathbf{A}_{i,j} \leftarrow c_{i,j} \mod q$
11: **return** $\mathbf{A}$

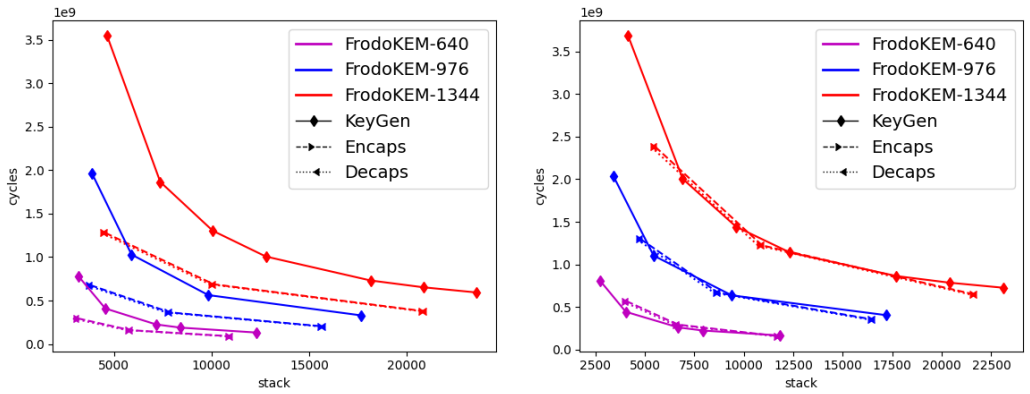
much different to AES128, where each 8×8 block can be generated using eight seeds at the cost of 128 bytes per seed.

In order to reduce the impact of SHAKE128 compared to AES128 on low-memory implementations, we propose a modification to the Frodo.Gen routine as shown in Algorithm 4 (compared to the original Algorithm 3). There, we propose to generate the rows as blocks of 168 bytes which corresponds to 84 coefficients in $\mathbb{Z}_q$. This choice of blocksize is motivated by the size of rate of SHAKE128 which is equal to $1600 - 256 = 1344$ bits which corresponds to 168 bytes. As this block-size fully exploits the rate of SHAKE128, the number of calls to the underlying permutation (which is a dominating cost factor) is unchanged compared to the specifications of FrodoKEM. We note that this solution still induce some slight overheads in the absorbing phase, since this is done for each 84-byte block instead of only once per row. However since this only consists of initializing a 1600-bit state the overhead is minimal as detailed in the benchmarks of Section 4. Concretely, to ensure uniqueness of every row, the input to SHAKE128 now requires both a row and column index as indicated in Line 5 of Algorithm 4. Since $n = 640$ and 976 do not have 84 as a divisor, the remaining bytes of the last block are discarded. With this change, memory-friendly implementations of FrodoKEM are in scope of the (adapted) SHAKE parameter sets as well.

The new Frodo.Gen for SHAKE enables efficient low-memory implementations without unused recomputations of rows, similarly as for AES128, but with slight differences in terms of block sizes used for the matrix multiplications. For the strategy described in Figure 1d, our SHAKE version processes blocks of $\mathbf{A}$ of size 8×84 instead of 8×8 for the AES128 (and still a full row of $\mathbf{S}'$). This means that 84 columns of $\mathbf{A}$ are processed simultaneously instead of 8. Hence, the memory allocation for these blocks increases by a factor 10.5. Eventually, it means also that the memory allocation for row chunks of $\mathbf{B}'$ also slightly increases.

3.6 Low-Memory Trade-Offs

We conclude this section by illustrating the possible trade-offs for a more granular stack-budgets described in Section 3.4. For this purpose, we use an STM32F4 discovery clocked at 24 MHz. See Section 4 for further details on this board and more extensive implementation results. Figure 3a contains the data for FrodoKEM with AES, while Figure 3b displays similar results when using SHAKE with the modified FrodoKEM.Gen from Section 3.5 (as regular version leads to prohibitive overheads). In these figures only the markers are achievable trade-offs. Namely, the parameter t_k in FrodoKEM.KeyGen is selected



(a) FrodoKEM-AES **(b)** FrodoKEM-SHAKE

Figure 3: Possible stack/cycle tradeoffs using low-memory strategies for FrodoKEM.

to be a divisor of the parameter n, where we only consider values small enough such that the stack memory is below the low-cost implementation. For FrodoKEM.Encaps and FrodoKEM.Decaps, the trade-off parameter considered are $t_{ed} \in \{0, 1, 3\}$. Here $t_{ed} = 7$ is excluded since this always produces worse results for both stack and performance compared to the low-cost implementation. From these figures, several observations can be made.

First, decreasing the trade-off parameter (t_k or t_{ed}) implies a reduction in memory, but comes at the cost of additional computing. Indeed, the right bullets on the plots corresponds to large(r) trade-off parameters, hence with large(r) memory but small(er) cycle counts. By decreasing the trade-off parameters, the stack reduces, but the cycle count increases. The cycle count impact is less than a factor five between using the maximum stack and the minimal stack for all cases, but many trade-offs are possible in-between.

Second, for all parameter sets one can observe that stack values of FrodoKEM.Encaps and FrodoKEM. Decaps are very close. Indeed, as Decaps contains a re-encryption, memory optimization and memory allocation techniques can be applied to both (see Section 3.3 for details).

4 Implementation and Evaluation

In this section we implement the presented memory reduction techniques in order to validate the exact performance impact and compare against the literature. The implementations target the Arm Cortex-M4: this allows us to re-use and extend some of the assembly routines present in existing implementations. The initial development was performed on an NXP FRDM-K28FA board (equipped with a Cortex-M4) containing a Kinetis MCU running at a maximum frequency of 150 MHz and containing 1 MB of SRAM. It was selected due to its large memory size that would fit any parameter set of FrodoKEM without problems. No special features of this board were used which allows direct execution on other platform which contain microcontrollers based on a Cortex-M4 CPU.

To simplify evaluation and comparison with existing literature, we target the development boards used by default in pqm4 for benchmarking and reporting. This default board is the STM32F4 Discovery, containing an STM32F407 microcontroller operating at a maximum frequency of 168 MHz and containing 192 kB of RAM. Unfortunately, this is not sufficient RAM to execute the implementations of the two largest parameter sets of FrodoKEM. This explains why pqm4 has only included the smallest parameter set ($n = 640$) into its framework. To get around this limitation and get meaningful result using pqm4 we also benchmark our implementations on the STM32 Nucleo-144, containing

Table 3: Stack memory consumption in kilobytes (kB) on the STM32F407VG (STM32F4 Discovery) platform. The implementation of [HOKG18] targets the Round 2 version of FrodoKEM.

Algorithm	Impl.	Keygen	Encaps	Decaps
FrodoKEM-640 (AES)	[HOKG18]	23.4	41.2	51.7
	ours	13.0	13.0	13.0
FrodoKEM-976 (AES)	[HOKG18]	35.5	63.5	63.6
	ours	19.0	18.8	18.9
FrodoKEM-640 (SHAKE)	[HOKG18]	22.4	37.8	48.2
	ours	12.5	14.5	14.5
FrodoKEM-976 (SHAKE)	[HOKG18]	33.8	58.0	58.1
	ours	18.6	19.9	19.9

640 kB of RAM and running at a maximum frequency of 120 MHz. As is typical for pqm4, all benchmarks were obtained at 24 MHz to avoid memory-related wait cycles. We measure only a single execution of the algorithms, but since they are implemented in constant-time and fairly slow there is only very little variance. All numbers were obtained using `arm-none-eabi-gcc 11.3.rel1` using the default optimization flags from the pqm4 build environment. For the ROM code size our implementations have little to no impact compared to the pqm4 implementations.

4.1 Comparison

We compare our results to the ones reported in pqm4 as well as the numbers reported by [HOKG18]. As mentioned, the pqm4 framework only includes an implementation of the smallest parameter set. In order to make a comparison for the larger parameter sets, we extend the pqm4 implementation such that it also supports these larger parameter sets. This was achieved by adapting the assembly for the matrix multiplication to work for all parameter sets. We also include a slightly modified version of the pqm4 implementation that includes the adaptation to the SHAKE parameter sets as suggested in Section 3.5. As one can see in Table 4 and Table 5, this has virtually no impact on the stack memory consumption for the unoptimized version and a minor increase in runtime of at most 5% compared to the FrodoKEM implementations in pqm4. However, it has major advantages for further reducing the stack memory.

We also note that [HOKG18] does not implement the FrodoKEM submission of Round 3 but rather an older version. Since the differences are fairly small, we think a direct comparison here is still meaningful. It is clear that their memory optimizations avoided impact on the runtime. For example, the cycle counts reported in [HOKG18, Table 5] are on par with pqm4. However, the memory reductions with our techniques are significant with little loss of performance. We summarize the improvements in Table 3.

STM32F4 Discovery. We ran both the pqm4 implementation of FrodoKEM as well as the implementation of the techniques from this work on the STM32F4 Discovery board. For the low-stack implementation, we benchmark the low-performance-impact techniques of Section 3.4 as well as the trade-offs that reduce the stack usage below 16 kB, 8 kB and 4 kB respectively. These sizes were chosen because they are common amounts of available stack memory on commercial (secure) microcontrollers. The results are in Table 4.

We observe that for the FrodoKEM-640 parameter set, the stack reduction techniques lead to a 53% and 84% reduction in stack memory, respectively, for the AES and SHAKE

implementations with a minimal increase in the cycle count of 2–6%. For the two larger parameter sets we are unable to compare to pqm4 as their stack usage is too large to run on the Discovery board. Note that the duplicate lines for the FrodoKEM-640 parameter sets are not an error; in these cases the low-performance-impact techniques of Section 3.4 were already under 16 kB in stack and outperformed those where the time-memory tradeoffs of Section 3.4 were set to achieve this limit.

Further reduction to 16 kB and 8 kB is possible for all parameter sets, and a reduction to 4 kB is even possible for the FrodoKEM-640 and FrodoKEM-976 parameter sets. Of course, the performance impact in this case is more significant. For example, in the decapsulation of FrodoKEM-640 (AES) the stack memory usage of pqm4 can be reduced by (approximately) a factor 28.6× at the expense of increasing the runtime by a factor 6.3×.

STM32 Nucleo-144. In order to run all FrodoKEM parameter sets we also consider the STM32 Nucleo-144 which is supported by pqm4 and does have a much larger stack memory. The conclusions are very similar as for the STM32F4 Discovery: the stack memory usage is reduced by a factor 2–3× with virtually no impact on the runtime, while the memory can be reduced further with more significant performance impact. The full results are in Table 5.

5 Conclusion

In this work we focus on memory optimization techniques for FrodoKEM. We present various optimization techniques for FrodoKEM that further reduce the stack memory usage with little impact on performance. We also showed that while for previous implementations it was not possible to execute any FrodoKEM parameter set on a device less than 72 kB SRAM, using the presented memory reduction strategies AES parameter sets can be implemented even within 8 kB SRAM constraints, and some even within 4 kB SRAM. The same can be achieved for the FrodoKEM SHAKE parameter sets, but it requires a minor tweak to the FrodoKEM specification. As this tweak does not lead to a (significant) loss of performance, we propose this to be adopted into any future FrodoKEM standard to enable implementation on embedded systems.

There are a few areas to investigate for further research. First, we did not investigate the re-use of memory buffers for inputs and outputs as workarea memory (i.e., keys and ciphertext). Since large buffers are required for FrodoKEM, the memory could be significantly reduced with such optimizations. We did not consider this optimization in this work since the small devices with only 4–16 kB of SRAM are unlikely to have such large buffers available. Instead, they would likely stream out keys and ciphertexts or write them to Flash memory that cannot be used as stack.

Second, for many constrained microcontroller applications, protection against fault and side-channel attacks is also a main priority. As a follow-up to this work we would like to investigate the impact of masking on the memory requirements of FrodoKEM.

Lastly, there are more NIST and non-NIST schemes which are a challenge to run on constrained devices. For instance Falcon [PFH+22] and Classic McEliece [ABC+22] both require rather large memory requirements as well. Future work could also investigate whether stack reductions are feasible there as well.

Table 4: Performance benchmarks for the FrodoKEM implementation (AES and SHAKE) on the STM32F407VG (STM32F4 Discovery) platform.

Implementation	Keygen		Encaps		Decaps	
	stack [bytes]	*cycle* [10^6]	*stack* [bytes]	*cycle* [10^6]	*stack* [bytes]	*cycle* [10^6]
FrodoKEM-640 (AES)						
pqm4[1]	31,964	43	62,476	47	83,100	47
ours	12,940	44	13,436	49	13,436	49
ours <16kB	12,940	44	13,436	49	13,436	49
ours <8kB	7,164	226	5,844	160	5,660	160
ours <4kB	3,204	776	3,188	294	2,908	294
FrodoKEM-976 (AES)						
pqm4[1]	-	-	-	-	-	-
ours	18,988	100	18,828	110	18,836	109
ours <16kB	9,844	563	15,708	203	15,556	203
ours <8kB	5,908	1,029	7,884	364	7,692	364
ours <4kB	3,884	1,962	3,812	677	3,620	677
FrodoKEM-1344 (AES)						
pqm4[1]	-	-	-	-	-	-
ours	25,636	186	24,732	203	24,804	202
ours <16kB	12,788	1,007	10,116	687	9,932	687
ours <8kB	7,380	1,865	4,572	1,278	4,372	1,279
FrodoKEM-640 (SHAKE)						
pqm4[1]	26,428	73	51,804	79	72,428	78
pqm4[2]	26,428	76	51,844	82	72,468	81
ours	12,516	75	14,468	85	14,476	84
ours <16kB	12,516	75	14,468	85	14,476	84
ours[2] **<8kB**	7,948	223	6,668	293	6,460	294
ours[2] **<4kB**	2,732	811	-	-	3868	568
FrodoKEM-976 (SHAKE)						
pqm4[1]	-	-	-	-	-	-
pqm4[2]	-	-	-	-	-	-
ours	18,572	169	19,860	186	19,868	185
ours[2] **<16kB**	9,388	637	8,700	668	8,500	667
ours[2] **<8kB**	5,444	1,103	4,796	1,296	4,596	1,296
FrodoKEM-1344 (SHAKE)						
pqm4[1]	-	-	-	-	-	-
pqm4[2]	-	-	-	-	-	-
ours	25,196	309	25,764	345	25,772	344
ours[2] **<16kB**	12,332	1,139	10,924	1,223	10,716	1,223
ours[2] **<8kB**	6,916	2,003	5,532	2,380	5,324	2,379

[1] https://github.com/mupq/pqm4/tree/master
commit: `685fbbb4059b882cad00f3fb4a345bf36b37ef4b`
[2] This includes a modification to the specification for the generation of **A** (see Section 3.5)

Table 5: Performance benchmarks for the FrodoKEM implementation (AES and SHAKE) on the STM32L4R5ZI (STM32 Nucleo) platform.

Implementation	Keygen		Encaps		Decaps	
	stack [bytes]	*cycle* [10^6]	*stack* [bytes]	*cycle* [10^6]	*stack* [bytes]	*cycle* [10^6]
FrodoKEM-640 (AES)						
pqm4[1]	31,976	47	62,496	51	83,112	50
ours	12,984	48	13,448	53	13,440	53
ours <16kB	12,984	48	13,448	53	13,440	53
ours <8kB	7,112	239	5,728	170	5,552	170
ours <4kB	3,216	821	3,192	317	2,992	317
FrodoKEM-976 (AES)						
pqm4[1]	48,144	106	95,440	116	126,808	114
ours	19,032	108	18,864	119	18,872	118
ours <16kB	9,808	607	15,600	220	15,416	220
ours <8kB	5,864	1,111	7,784	390	7,568	390
ours <4kB	3,896	2,118	3,848	731	3648	731
FrodoKEM-1344 (AES)						
pqm4[1]	65,832	198	130,832	214	173,960	212
ours	25,632	200	24,744	220	24,840	219
ours <16kB	12,744	1,092	9,992	732	9,800	732
ours <8kB	7,336	1,989	4,608	1,378	4,384	1,378
FrodoKEM-640 (SHAKE)						
pqm4[1]	26,416	80	51,800	85	72,416	84
pqm4[2]	26,416	84	51,840	88	72,456	88
ours	12,528	81	14,472	91	14,472	90
ours <16kB	12,528	81	14,472	91	14,472	90
ours[2] **<8kB**	7,944	237	6,688	315	6,488	315
FrodoKEM-976 (SHAKE)						
pqm4[1]	39,888	181	79,368	191	110,736	189
pqm4[2]	39,888	189	79,416	199	110,784	197
ours	18,576	182	19,864	204	19,872	203
ours[2] **<16kB**	9,328	690	8,728	717	8,520	717
ours[2] **<8kB**	5,392	1,194	4,800	1,389	4,592	1,389
FrodoKEM-1344 (SHAKE)						
pqm4[1]	54,632	330	108,872	347	152,000	345
pqm4[2]	54,632	345	108,912	362	152,040	360
ours	25,200	331	25,768	371	25,776	370
ours[2] **<16kB**	12,272	1,240	10,952	1,314	10,744	1,313
ours[2] **<8kB**	6,864	2,137	5,560	2,551	5,328	2,550

[1] https://github.com/mupq/pqm4/tree/master
commit: 685fbbb4059b882cad00f3fb4a345bf36b37ef4b
[2] This includes a slight modification to the specification for the generation of **A** (see Section 3.5)

References

[ABC+22] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions.

[ABD+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 5–17. ACM Press, October 2015.

[ABD+20] Erdem Alkim, Joppe W. Bos, Léo Ducas, Karen Easterbrook, Patrick Longa Brian LaMacchia, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 327–343. USENIX Association, August 2016.

[Age] Agence nationale de la sécurité des systèmes dínformation. Anssi views on the post-quantum cryptography transition. https://www.ssi.gouv.fr/uploads/2022/01/anssi-technical_position_papers-post_quantum_cryptography_transition.pdf.

[BCD+16] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1006–1018. ACM Press, October 2016.

[BFM+18] Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. Fly, you fool! Faster Frodo for the ARM Cortex-M4. Cryptology ePrint Archive, Paper 2018/1116, 2018. https://eprint.iacr.org/2018/1116.

[BOR+21] Joppe W. Bos, Maximilian Ofner, Joost Renes, Tobias Schneider, and Christine van Vredendaal. The matrix reloaded: Multiplication strategies in frodokem. In Mauro Conti, Marc Stevens, and Stephan Krenn, editors, *Cryptology and Network Security - 20th International Conference, CANS 2021, Vienna, Austria, December 13-15, 2021, Proceedings*, volume 13099 of *Lecture Notes in Computer Science*, pages 72–91. Springer, 2021.

[BRS22] Joppe W. Bos, Joost Renes, and Amber Sprenkels. Dilithium for Memory Constrained Devices. In Lejla Batina and Joan Daemen, editors, *AFRICACRYPT 2022*, pages 217–235, Cham, 2022. Springer Nature Switzerland.

[Fed] Federal Office for Information Security, Bonn, Germany. Migration to Post Quantum Cryptography, Recommendations for action by the BSI. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Crypto/Migration_to_Post_Quantum_Cryptography.pdf?__blob=publicationFile&v=2.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554. Springer, Heidelberg, August 1999.

[HOKG18] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. Standard Lattice-Based Key Encapsulation on Embedded Devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):372–393, 2018.

[HSHvdG16] Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. van de Geijn. Strassen's algorithm reloaded. In John West and Cherri M. Pancake, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis – SC*, pages 690–701. IEEE Computer Society, 2016.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[Mil86] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 417–426. Springer, Heidelberg, August 1986.

[Nat] National Institute of Standards and Technology. Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[Net] Netherlands National Communications Security Agency. Prepare for the threat of quantum computers. https://english.aivd.nl/binaries/aivd-en/documenten/publications/2022/01/18/prepare-for-the-threat-of-quantumcomputers/Prepare+for+the+threat+of+quantumcomputers.pdf.

[PFH+22] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[PZ03] J. Proos and C. Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Inf. Comput.*, 3:317–344, 2003.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck,

Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.