Classic McEliece: conservative code-based cryptography: guide for implementors

23 October 2022

Contents

1 Introduction

This document is aimed at readers upgrading cryptographic applications to use Classic McEliece.

The reader's top goal is assumed to be long-term security, ensuring that application data is solidly protected for the foreseeable future. Security includes implementation security, ensuring that bugs and side channels do not compromise the mathematical security provided by the Classic McEliece specification. Section 2 reviews typical security goals for cryptographic implementations.

Sometimes public-key cryptography is a large enough component of an application that there are also constraints on the public-key cost.[1] In such applications, the obvious issue for Classic McEliece is the large size of its public keys: 1MB for the recommended high-security parameter sets. However, there are techniques that typical Internet applications can use to share the cost of each public key across many ciphertexts. Systematically applying these techniques can turn Classic McEliece into the most cost-effective post-quantum system available today, because Classic McEliece has very low costs per ciphertext.

Section 3 states the sizes of public keys, private keys, ciphertexts, and session keys for all of the selected Classic McEliece parameter sets, and explains how to pick a Classic McEliece parameter set. The parameters themselves are given in the separate "cryptosystem specification" document along with descriptions of the mathematical functions. Section 4 looks at Classic McEliece from a network-engineering perspective.

In most cases, rather than writing implementations of Classic McEliece key generation, encapsulation, and decapsulation from scratch, implementors can simply reuse the implementations that are already available for various platforms. Section 5 reviews the existing implementations, including their performance and security features. Finally, Section 6 provides advice for situations where new implementations of these operations are needed.

2 Security goals for implementations

Classic McEliece is a key-encapsulation method (KEM). This means that it provides the following three operations:

- Key generation. Alice—for example, an Internet server—randomly generates a private key and a corresponding public key.
- Encapsulation. Bob—for example, a client contacting a server—uses the public key to randomly generate a ciphertext and a session key.

[1] These are not the same as constraints on *total* cryptographic costs. For example, if an application uses a Classic McEliece ciphertext to communicate an AES-256-GCM key used to protect N gigabytes of video data, then the costs of symmetric cryptography increase linearly with N while the costs of Classic McEliece are a constant independent of N.

- Decapsulation. Alice uses the ciphertext and the private key to generate the same session key as Bob.

The session key can then be used as, e.g., an AES-256-GCM key to authenticate and encrypt any number of messages being exchanged between Alice and Bob. Note that using signatures for authentication can typically be replaced with using KEMs for authentication (often at lower cost), as long as the signer is reachable; see, e.g., [6], [7, Section 8.1], and [35].

2.1 Mathematical security

Security analysis of the mathematical specifications of KEMs focuses primarily on ensuring "IND-CCA2" security. Reviewing the IND-CCA2 security of Classic McEliece is the main goal of the separate "guide for security reviewers" document.

"IND-CCA2" refers to "indistinguishability under chosen-ciphertext attacks". This means that the attacker seeing the public key and ciphertext cannot distinguish the resulting session key from random garbage of the same length, even if the application exposes session keys for some other ciphertexts to the attacker.

The definition of IND-CCA2 allows the attacker to freely ask for the session key for *any* ciphertext other than the target ciphertext. This may sound unreasonable, since applications should not provide so much power to the attacker; but achieving IND-CCA2 security seems feasible for cryptographic designers and guarantees mathematical security in a wide range of applications. Various cryptographic protocols have been proven to meet their security goals under the assumption that the underlying KEMs provide IND-CCA2 security, that the underlying stream ciphers provide PRF security, etc.

Sometimes "IND-CCA2" is written "IND-CCA". Beware that a weaker security goal, "IND-CCA1", is sometimes also written "IND-CCA". "IND-CCA2" is unambiguous.

2.2 Implementation correctness

Sometimes a bug in an implementation of a cryptographic operation means that the mathematical function computed by the implementation is not the same as the specified mathematical function. This can compromise security.

Tests are useful in catching many bugs. Keys and ciphertexts are generated randomly, but can still be compared across implementations if the implementations use random bytes in the same way, since a test suite can provide the same random bytes to each implementation. Classic McEliece specifies exactly how randomness is converted into keys and ciphertexts, and implementation outputs for the same randomness are checked by the SUPERCOP test suite [16] and by NIST's Known Answer Tests ("KATs").

It is important for tests to include not just the behavior of cryptographic operations on *correct* inputs (e.g., keys produced by KeyGen and ciphertexts produced by Encap), but

also the behavior of cryptographic operations on other inputs that might be produced by an attacker. KEM tests carried out by SUPERCOP automatically include some random variations of each ciphertext.

Sometimes bugs occur only for inputs beyond the scope of existing tests. This does not imply that attackers will have trouble triggering those inputs. Advances in tools for "formal verification" are making it feasible to have a computer check that multiple implementations produce the same results for *all* inputs. See generally [4].

Even when all implementations, including reference implementations, are formally verified to produce identical results as each other for all inputs, there is still a risk that all of the implementations deviate from the specified cryptographic operations. This problem would disappear if specifications were written in computer languages understood by verification tools. However, specifications are typically required to be written in English and/or pseudocode, since this is commonly believed to be better than real code as a foundation for cryptanalysis and proofs. Deviations between specifications and reference implementations often occur, and are typically caught by implementors manually checking specifications.

2.3 Security against timing attacks

Many implementations of various cryptographic operations have been broken by attacks inspecting not just the public keys and ciphertexts considered in mathematical security analyses but also the *time* taken by the implementations.

The time taken by a computation is often influenced by secret data inside the computation, and is usually visible to the attacker. The attacks work backwards to compute the secret data. Extremely small variations in timings often turn out to be exploitable: see, e.g., [2] and, for time variations as small as a single clock cycle, [44].

Implementors can try to hide timings from attackers, but this is difficult to accomplish and difficult to review. What has been much more successful is "constant-time programming", cutting off data flow from secrets to timings. See [13] for a survey and further references.

SUPERCOP includes TIMECOP, an easy-to-use tool that automatically checks for two important sources of timing variations: branches based on secret data, and memory addresses (array indices) based on secret data. The official Classic McEliece software implementations have been checked by TIMECOP.

There can be further sources of problematic timing variations. For example, multipliers on many small CPUs take data-dependent time. Current tools to check for multiplications based on secret data are not as advanced as tools to check for branches and memory addresses based on secret data. A recent announcement from Intel [28] appears to indicate that there are timing variations in the vectorized multipliers on many Intel CPUs, although it is unclear at this point which inputs will trigger these variations.

Many cryptographic systems rely heavily on integer multiplication and would suffer heavily in performance from avoiding the CPU multipliers. Multiplications are a much smaller issue

for Classic McEliece, where the basic objects are binary vectors and binary polynomials rather than integers. There are some CPUs where integer multipliers take constant time and can be productively used to multiply binary polynomials (see [20, Section 5.1.2]), but this is not a large speedup (see [20, Table 8]), and in any case the official Classic McEliece software avoids this.

Timing variations based on *public* data are acceptable. For example, key generation in Classic McEliece involves a variable number of key-generation attempts: each attempt succeeds with probability about 29% for the non-`f` parameter sets, and close to but not exactly 100% for the `f` parameter sets. The final *successful* key generation takes constant time, and it uses separate random numbers from the unsuccessful key-generation attempts; in other words, the information about secrets that is leaked through timing is information about secrets that are not used. For this and other rejection-sampling loops, the official Classic McEliece software uses a `crypto_declassify` function provided by TIMECOP to indicate that a variable can be safely made public.

2.4 Security against other side-channel attacks and fault attacks

In addition to protection against timing attacks, protection against other side channels may be necessary depending on the intended use case. For example, smart-card implementations are often exposed to power attacks and electromagnetic attacks, and thus need to be protected against these attacks.

There are some situations where it is not clear which side channels might be available. If in doubt, implementors of a cryptosystem should generally opt for implementations that include more side-channel defenses, although sometimes this needs to be balanced against other issues such as verification of correctness.

Often attackers can trigger faults in computations, further complicating the security analysis. A one-time single-bit fault in a stored private key, something that will occur naturally for a fraction of users, can eliminate IND-CCA2 security; see [11].

For a variety of side-channel attacks and fault attacks against unprotected implementations of the McEliece cryptosystem, see [41], [37], [26], [38], [3], [33], [39], [40], [32], [19], and [25].

There is an extensive literature on general-purpose defense strategies. A general-purpose defense against fault attacks is to encode data using an error-correcting code; data stored in ECC RAM (or equivalently encoded in software; see [10]) is automatically encoded this way, but more work is required to protect data inside computations. A general-purpose defense against side-channel attacks is to *randomly* encode data. There has been progress towards automated conversion of any computation into a computation with side-channel defenses; see, e.g., [30].

	Public key	Private key	Ciphertext	Session key
mceliece348864	261120	6492	96	32
mceliece348864f	261120	6492	96	32
mceliece460896	524160	13608	156	32
mceliece460896f	524160	13608	156	32
mceliece6688128	1044992	13932	208	32
mceliece6688128f	1044992	13932	208	32
mceliece6960119	1047319	13948	194	32
mceliece6960119f	1047319	13948	194	32
mceliece8192128	1357824	14120	208	32
mceliece8192128f	1357824	14120	208	32

Table 1: Sizes of inputs and outputs to the complete cryptographic functions. All sizes are expressed in bytes.

3 Classic McEliece parameter sets

3.1 Sizes of inputs and outputs

Table 1 reports sizes of public keys, private keys, ciphertexts, and session keys for each selected parameter set. Note the large sizes of public keys, but at the same time the small sizes of ciphertexts, much smaller than structured-lattice ciphertexts that claim the same security levels. Classic McEliece is used in the PQ-WireGuard [27] VPN, which beyond security is "mainly concerned about ciphertext size".

It is possible to compress the private key down to 40 bytes (or 32 bytes for non-`f` parameter sets) with uncompression less expensive than key generation. There are intermediate compression options that further reduce the uncompression cost. Compression can be particularly useful for situations where many private keys are being stored on a small device.

3.2 Considerations in picking a parameter set

There are two basic approaches used by implementors to select cryptographic key sizes. The first approach is to prioritize performance: choose high-performance key sizes, and change only if the key sizes are demonstrated to have security problems. The second approach is to prioritize security: choose high-security key sizes, and change only if the key sizes are demonstrated to have performance problems. The second approach is safer.

The security level of the McEliece cryptosystem against known attacks is much more stable than the security level of other proposals for post-quantum encryption. Known attacks against the smallest selected parameter set, `348864`, are more expensive than brute-force AES-128 key search. However, AES-128 is breakable with foreseeable improvements in computer technology, at least with multi-target attacks (see generally [5]), so implementors are advised to take higher-security parameters.

The 6688128 and 6960119 parameter sets are recommended choices for long-term security. The 6960119 parameter set was introduced in [17] as maximizing security for keys that fit into 1MB. The 6688128 parameter set slightly simplifies implementations (for example, public keys and ciphertexts have no padding bits; see "IND-CCA2 for encodings" in the separate "design rationale" document) and has almost the same security level.

The 8192128 parameter set has even higher quantitative security against known attacks—about 15% more bits of security; in other words, billions of times more difficult to break—but 6688128 and 6960119 are already beyond feasible attacks by such a large distance that it is difficult to see what risk is being addressed by 8192128. Meanwhile 6688128 and 6960119 have the advantage of an extra defense explained in "OW-CPA security of length below field size" in the separate "guide for security reviewers" document.

The f and non-f versions of each parameter set are interoperable. The f versions have faster key generation, while the non-f versions have simpler key generation.

4 Engineering cryptographic network applications for efficiency

KEMs designed for IND-CCA2 security, such as Classic McEliece, are designed so that a public key can be safely reused for many ciphertexts from many users. This has important consequences for applications that are trying to minimize the overall costs of using the KEM.

The bottom diagram in Figure 1 depicts 30 clients at 10 Internet service providers sending ciphertexts to a server. Each ciphertext is sent to the local ISP and then from the local ISP to the server.

The top diagram in Figure 1 depicts the preliminary step of the server distributing its public key to those 30 clients. Since the same public key is being sent to all clients, it would be redundant to send the public key three times to each ISP. Instead each ISP caches the public key to distribute to any local clients that want it. This caching reduces the costs of public-key distribution by a factor of almost 3, and the effective benefits increase as the number of clients per ISP increases.

In other words, a public-key byte does not incur the same inherent costs as a ciphertext byte: each ciphertext byte must be sent end-to-end, while a public-key byte can use a lower-cost broadcast network. This is compatible with periodically refreshing public keys and erasing old private keys for forward secrecy.

Caching mechanisms are already widely deployed on the Internet today. For example, the Internet's Domain Name System (DNS) can be used for arbitrary types of data (split into packets of suitable size), and automatically caches data at ISPs. A DNS cache on a single computer is typically configured to use 90% of the RAM on the computer (see, e.g., [23]), which generally means many gigabytes, i.e., space for many thousands of McEliece keys. This is not enough to cache a key for every server on the Internet, but it easily covers the

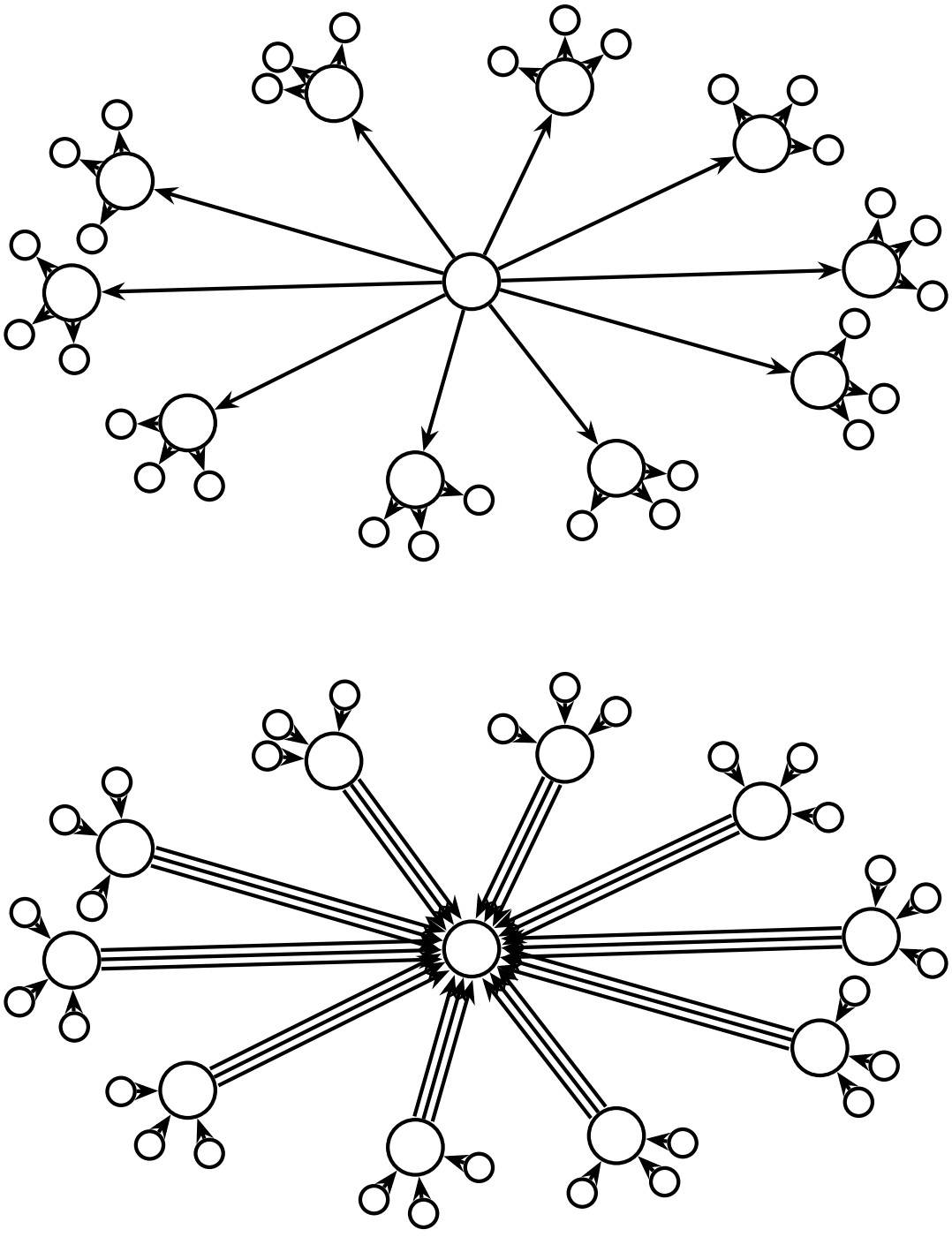

Figure 1: Top: Server sending a public key to 30 clients at 10 ISPs, with the public key sent only once to each ISP. Bottom: Server receiving ciphertexts from 30 clients at 10 ISPs.

most popular servers.

Furthermore, while applications typically need ciphertexts to be delivered immediately, they can typically retrieve most, if not all, of the necessary public keys in advance. A mobile phone, for example, often needs to deliver ciphertexts through a cellular network, but can retrieve public keys while it is on a lower-cost wireless network.

A structured-lattice KEM typically has public keys and ciphertexts around a kilobyte, so implementors can aim to gain about a factor 2 in cost from efficient public-key distribution. For Classic McEliece, keys are much larger but ciphertexts are much smaller, so implementors can aim to gain much more from efficient public-key distribution—and can aim for lower total costs than any other post-quantum KEM can achieve.

5 Existing implementations of the Classic McEliece operations

This section reviews existing official and unofficial implementations of Classic McEliece key generation, encapsulation, and decapsulation.

Some implementations listed below are for the round-3 version of Classic McEliece, which includes "plaintext confirmation". Plaintext confirmation produces a small slowdown in computations and expands ciphertexts by 32 bytes.

Another encapsulation implementation is available as part of McTiny [15], which shows how clients can stream ephemeral Classic McEliece keys through a stateless network server to set up new sessions. The server immediately handles each network packet, and does not allocate any RAM per client.

5.1 Official implementations

The official implementations are the following four software implementations for each of the ten selected parameter sets:

- `ref`, portable C software. This implementation is designed for clarity, not performance. This is the reference implementation of Classic McEliece.
- `vec`, portable C software. This implementation vectorizes across 64-bit integers.
- `sse`, C software using machine-specific intrinsics. This implementation uses the Intel/AMD 128-bit vector instructions.
- `avx`, C software using machine-specific intrinsics. This implementation uses the Intel/AMD 256-bit vector instructions.

These four implementations are interoperable and produce identical test vectors. All of these implementations are designed to avoid all data flow from secrets to timing.

	operation	quartile	median	average	quartile
mceliece348864	keypair	35039714	56705880	60333686	67615011
mceliece348864f	keypair	35970884	35976620	35978769	35981416
mceliece460896	keypair	116209838	153266214	213425513	264539700
mceliece460896f	keypair	117267744	117297677	117301747	117331130
mceliece6688128	keypair	265554240	443746986	479441242	532990499
mceliece6688128f	keypair	274329761	274384229	274484625	274430338
mceliece6960119	keypair	241288202	316995472	432622560	468394597
mceliece6960119f	keypair	240198020	240226771	240328382	240254131
mceliece8192128	keypair	308008713	486195290	548932457	664466919
mceliece8192128f	keypair	306203040	306238935	306349035	306280509
mceliece348864	enc	34951	36457	37585	38980
mceliece460896	enc	69674	76086	81312	88956
mceliece6688128	enc	165296	171442	175788	185077
mceliece6960119	enc	139980	144678	147192	149592
mceliece8192128	enc	155174	156945	158068	159040
mceliece348864	dec	127036	127140	127668	127256
mceliece460896	dec	262919	263046	263634	263225
mceliece6688128	dec	305910	306212	306946	306925
mceliece6960119	dec	286353	286596	287218	287038
mceliece8192128	dec	309938	310097	310773	310475

Table 2: Time for complete cryptographic functions on an Intel Haswell CPU core. All times are expressed in CPU cycles. Statistics are computed across SUPERCOP's default 93 experiments. The `f` variants have different `keypair` algorithms but identical `enc` algorithms and identical `dec` algorithms.

For `6960119` and `6960119f`, there is a distinction between Simply Decoded Classic McEliece and Narrowly Decoded Classic McEliece. The official software implements Narrowly Decoded Classic McEliece.

Table 2 reports speeds of the `avx` implementations on an Intel Haswell CPU core. These software measurements were collected using `supercop-20220506` running on a computer named `hiphop`. The CPU on `hiphop` is an Intel Xeon E3-1220 v3 running at 3.10GHz. This CPU does not support hyperthreading. It does support Turbo Boost but `/sys/devices/system/cpu/intel_pstate/no_turbo` was set to 1, disabling Turbo Boost. `hiphop` has 32GB of RAM and runs Ubuntu 18.04. Benchmarks used `./do-part`, which ran on one core of the CPU. The compiler list was reduced to just `gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -fPIC -fPIE`.

For comparison, the `mceliece8192128` software originally submitted for round 1 took about 2 billion cycles for each key-generation attempt (and on average about 6 billion cycles for total key generation), slightly under 300000 cycles for encapsulation, and slightly over 450000 cycles for decapsulation.

5.2 Microcontroller implementations

The paper [29] reported benchmarks of various post-quantum systems on an ARM Cortex-M4, specifically an STM32F4DISCOVERY board. That paper excluded Classic McEliece, saying that the Classic McEliece public keys "are too large to fit into the memory of our platform" and concluding that Classic McEliece is "arguably unsuited for a microcontroller environment of this size".

However, a newer paper [20] reports a full constant-time `mceliece348864` implementation on an STM32F4DISCOVERY board taking 2146932033 cycles for key generation (or 1430811294 cycles with `348864f`), 582199 cycles for encapsulation, and 2706681 cycles for decapsulation. This implementation is available from https://github.com/pqcryptotw/mceliece-arm-m4.

The board actually has enough memory to store public keys, namely flash memory. Flash memory is efficient for large batch operations. Classic McEliece's public-key access, even for key generation, can be organized to involve large batch operations to flash memory and relatively little SRAM.

For decapsulation, the same paper [20] reports 6353186 cycles for `mceliece460896`, 7412111 cycles for `mceliece6688128`, and 7481747 cycles for `mceliece8192128`. Decapsulation does not need the public key. The paper also reports 1081335 cycles for encapsulation for `mceliece460896`.

An earlier paper [34] implemented some of the Classic McEliece operations for the M4, with higher cycle counts. That paper streams public keys through a separate device rather than storing the keys in flash memory.

5.3 FPGA implementations

The computations in McEliece's cryptosystem are particularly well suited for hardware implementations. FPGA implementations for the core mathematical functions (not including hashing etc.) were introduced in [42] for key generation and [43] for all operations. The implementations from [43] are available from https://caslab.csl.yale.edu/code/niederreiter/.

A complete FPGA implementation (including hashing etc.), also improving efficiency compared to [43], is described in [21]. For example, for `mceliece348864`, one of the implementations from [21] runs at 113MHz on a Xilinx Artix 7 (xc7a200t) FPGA, uses 40018 LUT, 4 DSP, 61881 FF, and 177.5 BRAM, and takes 0.97, 0.03, and 0.10 million cycles for KeyGen, Encap, and Decap respectively. Another implementation takes more cycles but fits in less area. See [21] for detailed benchmarks. The implementations from [21] are planned to be available from https://caslab.csl.yale.edu/code/pqc-classic-mceliece/.

6 Building new implementations of the Classic McEliece operations

This section describes various options and resources available for implementors building new implementations of Classic McEliece key generation, encapsulation, and/or decapsulation. The primary resources are existing implementations. These implementations are often accompanied by papers describing the necessary computations (e.g., [14] and [22]), although the papers often skip describing routine steps.

For security, implementors must make sure to implement exactly the mathematical functions described in the separate "cryptosystem specification" document, including the handling defined there of random bits and invalid inputs. All algorithmic options are subject to the rule of computing the specified mathematical functions.

Implementors must also keep in mind the security goals from Section 2, such as security against timing attacks. Some techniques for eliminating specific timing leaks are described in this section, but this is not a substitute for applying tools that comprehensively check for timing leaks.

The following subsections give implementation and testing guidance for the functions described in the separate "cryptosystem specification" document. Steps mentioned below refer to steps in algorithms stated in that document.

6.1 Key generation

Irreducible-polynomial generation. Step 3 of IRREDUCIBLE computes the minimal polynomial of β. There are various minimal-polynomial algorithms in the literature. These algorithms become simpler in this context since the minimal polynomial is kept only when it has degree t.

One simple option is to use linear algebra to find solutions $(g_0, g_1, \ldots, g_{t-1}) \in \mathbb{F}_q^t$ to the linear equation $g_0\beta^0 + g_1\beta^1 + \cdots + g_{t-1}\beta^{t-1} = \beta^t$. There are always solutions. A unique solution corresponds to a minimal polynomial of degree t, while a non-unique solution means that β must be rejected.

In principle, implementations should be tested not just on random field elements β but also on field elements β that are roots of irreducible polynomials of degrees strictly dividing t.

Field ordering. Step 2 of FIELDORDERING returns $\perp$ if $a_0, a_1, \ldots, a_{q-1}$ are not distinct. A convenient way to test distinctness is to merge this step into Step 3, which sorts the pairs (a_i, i) in lexicographic order.

Constant-time sorting is easily carried out via "sorting networks", which are sequences of min-max operations at constant array positions, as long as the underlying min-max operations are carried out in constant time. Knuth's "merge-exchange" [31, Algorithm 5.2.2M]

is a simple sorting network using $q^{1+o(1)}$ min-max operations for array size q. Tools to automatically verify sorting-network software are available from [8].

Control-bit generation. Field orderings are encoded in private keys as control bits for a Beneš network that computes the necessary permutation π. See [9] for Python software and computer-verified correctness proofs for control-bit computation; the Python software is also included in the Classic McEliece specification.

As low-cost protection against faults in the control-bit computation, implementors are advised to check after the computation that applying the Beneš network produces π, and to restart key generation if this test fails; applying the Beneš network is very fast.

Reducing to systematic form. Reducing a matrix X to systematic form means computing the unique systematic-form matrix having the same row space as X, if such a matrix exists. One way to do this is as follows:

- Use Gaussian elimination to compute R in reduced row-echelon form.
- Return R if R is in systematic form, else $\perp$.

One can streamline Gaussian elimination in this context by using early aborts. First try to reduce the initial columns to triangular form; if the answer is $\perp$ then one can skip reducing these columns to an identity matrix, and one can skip the operations on the remaining columns. There must always be a nonzero entry in column 0 (or else the answer is $\perp$), then after elimination there must always be a nonzero entry in column 1 (or else the answer is $\perp$), etc.

Reducing to semi-systematic form. As in the special case of systematic form, one way to compute the (μ, ν)-semi-systematic form is to compute the reduced row-echelon form R, and then output R if R is in (μ, ν)-semi-systematic form.

A more streamlined computation requires a nonzero entry in column 0, then after elimination requires a nonzero entry in column 1, and so on for the first $r - \mu$ columns; then computes the reduced row-echelon form of the next ν columns of the bottom μ rows, and requires this submatrix to have rank μ; and then completes the computation of reduced row-echelon form of the entire matrix.

6.2 Encapsulation

Generating fixed-weight vectors. Step 5 of FixedWeight converts $a_0, \ldots, a_{t-1}$ into the unique weight-t vector e such that $e_{a_i} = 1$ for each i. The obvious way to implement this conversion is to start with $e = 0$, then use a RAM operation to set e_{a_0} to 1, then use a RAM operation to set e_{a_1} to 1, etc. However, on most platforms, this leaks information about $a_0, \ldots, a_{t-1}$ through timing. Implementations that are not guaranteed to be using

platforms with constant-time RAM should instead simulate RAM operations using constant-time arithmetic.

Matrix-vector multiplication. Bits of e that are 0 obviously do not affect the output, but skipping them again leaks information through timing.

6.3 Decapsulation

Decoding algorithms. The Decode definition refers to H, which one can compute via $\text{MatGen}(\Gamma') = (T, \ldots)$. However, this recomputation is not necessary.

Courses on coding theory include algorithms (not using H) for Step 2 of Decode. See [12] for a more direct introduction to the simplest algorithm. For speedups to this algorithm, see generally [14] and [22].

This algorithm either returns $\perp$ or guarantees that $Hc = 0$, which mathematically implies $C = He$ in Step 4. However, implementors are advised to separately check $C = He$ in Step 4 as protection against faults in Step 2. This again does not require recomputing H. There are other parity-check matrices H' (related to $\hat{H}$ in key generation) for the same code that are recovered from Γ' much more efficiently than MatGen, and that can be applied to vectors without using quadratic space; see generally [14]. Comparing $H'(v + e)$ to 0, where $v = (C, 0, \ldots, 0)$, has the same effect as comparing C to He.

Avoiding early aborts. The logic for the correctness of Decode relies on Step 2 always finding a codeword at distance t if one exists. It does not rely on Step 2 failing in the cases that a codeword does not exist: Decode remains correct if, instead of returning $\perp$, Step 2 chooses some vector $c \in \mathbb{F}_2^n$ and continues on to Step 3.

The distinction between success and failure of Decode is secret in the context of the Classic McEliece KEM. In particular, immediately stopping the computation when Step 2 returns $\perp$ would reveal this distinction through timing, so it is recommended for implementors to have Step 2 always choose some $c \in \mathbb{F}_2^n$.

Similarly, the distinction between failures and successes inside Decap is secret information. It is recommended for implementors to always go through the same sequence of computations, using arithmetic to simulate tests and conditional assignments.

Double-checks on private keys. Beyond using generic error-correcting codes to catch occasional DRAM errors, implementors may wish to carry out more stringent checks on private keys. Options include recomputing s as a hash of δ, checking the weight of c, checking irreducibility of g, recomputing β and checking whether $g(\beta) = 0$ (which is faster than recomputing g), and similarly checking the α control bits (which is faster than recomputing them).

References

[1] *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24–27 May 2021*. IEEE, 2021. https://doi.org/10.1109/SP40001.2021.

[2] Martin R. Albrecht and Kenneth G. Paterson. Lucky microseconds: A timing attack on Amazon's s2n implementation of TLS. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology—EUROCRYPT 2016—35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8–12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 622–643. Springer, 2016. https://eprint.iacr.org/2015/1129.

[3] Roberto Avanzi, Simon Hoerder, Dan Page, and Michael Tunstall. Side-channel attacks on the McEliece and Niederreiter public-key cryptosystems. *J. Cryptogr. Eng.*, 1(4):271–281, 2011. https://doi.org/10.1007/s13389-011-0024-9.

[4] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24–27 May 2021* [1], pages 777–795. https://eprint.iacr.org/2019/1393.

[5] Daniel J. Bernstein. Break a dozen secret keys, get a million more for free, 2015. https://blog.cr.yp.to/20151120-batchattacks.html.

[6] Daniel J. Bernstein. The post-quantum Internet, 2016. https://cr.yp.to/talks.html#2016.02.24.

[7] Daniel J. Bernstein. D2.5: Internet: Integration, 2018. https://pqcrypto.eu.org/deliverables/d2.5.pdf.

[8] Daniel J. Bernstein. djbsort, 2019. https://sorting.cr.yp.to.

[9] Daniel J. Bernstein. Verified fast formulas for control bits for permutation networks, 2020. https://cr.yp.to/papers.html#controlbits.

[10] Daniel J. Bernstein. `libsecded`, 2022. https://pqsrc.cr.yp.to/downloads.html.

[11] Daniel J. Bernstein. A one-time single-bit fault leaks all previous NTRU-HRSS session keys to a chosen-ciphertext attack, 2022. https://cr.yp.to/papers.html#ntrw, to appear at INDOCRYPT 2022.

[12] Daniel J. Bernstein. Understanding binary-Goppa decoding, 2022. https://cr.yp.to/papers.html#goppadecoding.

[13] Daniel J. Bernstein and Billy Bob Brumley. Timing attacks, 2022. https://timing.attacks.cr.yp.to/.

[14] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems—CHES 2013—15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 250–272. Springer, 2013. https://tungchou.github.io/papers/mcbits.pdf.

[15] Daniel J. Bernstein and Tanja Lange. McTiny: Fast high-confidence post-quantum key erasure for tiny network servers. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12–14, 2020*, pages 1731–1748. USENIX Association, 2020. https://mctiny.org.

[16] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems, 2022. https://bench.cr.yp.to.

[17] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the McEliece cryptosystem. In Buchmann and Ding [18], pages 31–46. https://eprint.iacr.org/2008/318.

[18] Johannes A. Buchmann and Jintai Ding, editors. *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17–19, 2008, Proceedings*, volume 5299 of *Lecture Notes in Computer Science*. Springer, 2008. https://doi.org/10.1007/978-3-540-88403-3.

[19] Pierre-Louis Cayrel, Brice Colombier, Vlad-Florin Dragoi, Alexandre Menu, and Lilian Bossuet. Message-recovery laser fault injection attack on the Classic McEliece cryptosystem. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology—EUROCRYPT 2021—40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part II*, volume 12697 of *Lecture Notes in Computer Science*, pages 438–467. Springer, 2021. https://eprint.iacr.org/2020/900.

[20] Ming-Shing Chen and Tung Chou. Classic McEliece on the ARM Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):125–148, 2021. https://doi.org/10.46586/tches.v2021.i3.125-148.

[21] Po-Jen Chen, Tung Chou, Sanjay Deshpande, Norman Lahr, Ruben Niederhagen, Jakub Szefer, and Wen Wang. Complete and improved FPGA implementation of Classic McEliece. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(3):71–113, 2022. https://doi.org/10.46586/tches.v2022.i3.71-113.

[22] Tung Chou. McBits revisited. In Fischer and Homma [24], pages 213–231. https://tungchou.github.io/papers/mcbits_revisited.pdf.

[23] Internet Systems Consortium. Configuration reference, 2022. https://bind9.readthedocs.io/en/latest/reference.html.

[24] Wieland Fischer and Naofumi Homma, editors. *Cryptographic Hardware and Embedded Systems—CHES 2017—19th International Conference, Taipei, Taiwan, September 25–28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*. Springer, 2017.

[25] Qian Guo, Andreas Johansson, and Thomas Johansson. A key-recovery side-channel attack on Classic McEliece implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):800–827, 2022. https://doi.org/10.46586/tches.v2022.i4.800-827.

[26] Stefan Heyse, Amir Moradi, and Christof Paar. Practical power analysis attacks on software implementations of McEliece. In Sendrier [36], pages 108–125. https://doi.org/10.1007/978-3-642-12929-2_9.

[27] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Florian Weber, and Philip R. Zimmermann. Post-quantum WireGuard. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24–27 May 2021* [1], pages 304–321. https://eprint.iacr.org/2020/379.

[28] Intel. Data operand independent timing instruction set architecture (ISA) guidance, 2022. https://web.archive.org/web/20220821152732/https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html.

[29] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. 2019. https://eprint.iacr.org/2019/844.

[30] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. Automated generation of masked hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):589–629, 2022. https://doi.org/10.46586/tches.v2022.i1.589-629.

[31] Donald E. Knuth. The art of computer programming, volume III: sorting and searching, 1973.

[32] Norman Lahr, Ruben Niederhagen, Richard Petri, and Simona Samardjiska. Side channel information set decoding using iterative chunking—plaintext recovery from the "Classic McEliece" hardware reference implementation. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology—ASIACRYPT 2020—26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 881–910. Springer, 2020. https://eprint.iacr.org/2019/1459.

[33] H. Gregor Molter, Marc Stöttinger, Abdulhadi Shoufan, and Falko Strenzke. A simple power analysis attack on a McEliece cryptoprocessor. *J. Cryptogr. Eng.*, 1(1):29–36, 2011. https://doi.org/10.1007/s13389-011-0001-3.

[34] Johannes Roth, Evangelos G. Karatsiolis, and Juliane Krämer. Classic McEliece implementation with low memory footprint. In Pierre-Yvan Liardet and Nele Mentens, editors, *Smart Card Research and Advanced Applications—19th International Conference, CARDIS 2020, Virtual Event, November 18–19, 2020, Revised Selected Papers*, volume 12609 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2020. https://eprint.iacr.org/2021/138.

[35] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9–13, 2020*, pages 1461–1480. ACM, 2020. https://eprint.iacr.org/2020/534.

[36] Nicolas Sendrier, editor. *Post-Quantum Cryptography, Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25–28, 2010. Proceedings*, volume 6061 of *Lecture Notes in Computer Science*. Springer, 2010. https://doi.org/10.1007/978-3-642-12929-2.

[37] Abdulhadi Shoufan, Falko Strenzke, H. Gregor Molter, and Marc Stöttinger. A timing attack against Patterson algorithm in the McEliece PKC. In Dong Hoon Lee and Seokhie Hong, editors, *Information, Security and Cryptology—ICISC 2009, 12th International Conference, Seoul, Korea, December 2–4, 2009, Revised Selected Papers*, volume 5984 of *Lecture Notes in Computer Science*, pages 161–175. Springer, 2009. https://doi.org/10.1007/978-3-642-14423-3_12.

[38] Falko Strenzke. A timing attack against the secret permutation in the McEliece PKC. In Sendrier [36], pages 95–107. https://doi.org/10.1007/978-3-642-12929-2_8.

[39] Falko Strenzke. Message-aimed side channel and fault attacks against public key cryptosystems with homomorphic properties. *J. Cryptogr. Eng.*, 1(4):283–292, 2011. https://doi.org/10.1007/s13389-011-0020-0.

[40] Falko Strenzke. Timing attacks against the syndrome inversion in code-based cryptosystems. In Philippe Gaborit, editor, *Post-Quantum Cryptography—5th International Workshop, PQCrypto 2013, Limoges, France, June 4–7, 2013. Proceedings*, volume 7932 of *Lecture Notes in Computer Science*, pages 217–230. Springer, 2013. https://eprint.iacr.org/2011/683.

[41] Falko Strenzke, Erik Tews, H. Gregor Molter, Raphael Overbeck, and Abdulhadi Shoufan. Side channels in the McEliece PKC. In Buchmann and Ding [18], pages 216–229. https://doi.org/10.1007/978-3-540-88403-3_15.

[42] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based key generator for the Niederreiter cryptosystem using binary Goppa codes. In Fischer and Homma [24], pages 253–274. https://eprint.iacr.org/2017/595.

[43] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based Niederreiter cryptosystem using binary Goppa codes. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography—9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9–11, 2018, Proceedings*, volume 10786 of *Lecture Notes in Computer Science*, pages 77–98. Springer, 2018. https://eprint.iacr.org/2017/1180.

[44] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. Cryptogr. Eng.*, 7(2):99–112, 2017. https://eprint.iacr.org/2016/224.