A one-time single-bit fault leaks all previous NTRU-HRSS session keys to a chosen-ciphertext attack

Daniel J. Bernstein[1,2]

[1] Department of Computer Science, University of Illinois at Chicago, USA
[2] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
djb@cr.yp.to

Abstract. This paper presents an efficient attack that, in the standard IND-CCA2 attack model plus a one-time single-bit fault, recovers the NTRU-HRSS session key. This type of fault is expected to occur for many users through natural DRAM bit flips. In a multi-target IND-CCA2 attack model plus a one-time single-bit fault, the attack recovers every NTRU-HRSS session key that was encapsulated to the targeted public key before the fault. Software carrying out the full multi-target attack, using a simulated fault, is provided for verification. This paper also explains how a change in NTRU-HRSS in 2019 enabled this attack.

Keywords: chosen-ciphertext attacks, natural faults, implicit rejection

1 Introduction

In 2016, the call for submissions for the NIST Post-Quantum Cryptography Standardization Project [61] said that NIST intends to standardize "one or more schemes that enable existentially unforgeable digital signatures with respect to an adaptive chosen message attack" and "one or more schemes that enable 'semantically secure' encryption or key encapsulation with respect to adaptive chosen ciphertext attack"—in other words, signature systems providing EUF-CMA security, and PKEs or KEMs providing IND-CCA2 security.

The EUF-CMA game allows the attacker to call an oracle that signs arbitrary messages; the only restriction is that the attacker doesn't win the game if the attacker's forged message was specifically provided as input to the oracle.

This work was funded by the Intel Crypto Frontiers Research Center; by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments—EXC 2092 CASA—390781972 "Cyber Security in the Age of Large-Scale Adversaries"; by the U.S. National Science Foundation under grant 1913167; by the Taiwan's Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP); and by the Cisco University Research Program. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation" (or other funding agencies). Permanent ID of this document: 662cf4ad8f5bff33ae4d71d56051a656d8a62e48. Date: 2022.08.29.

Similarly, the IND-CCA2 game for KEMs allows the attacker to call an oracle that decapsulates arbitrary messages, although the attacker doesn't win the game if the oracle was used specifically to decapsulate the target message.

An application providing such powerful oracles is thoroughly flawed and should not be used. But applications do sign and decapsulate *some* messages, providing *some* of the same information. Aiming merely for security in the absence of such oracles would then be a disaster, as illustrated by Bleichenbacher's million-message attack [**20**], which was demonstrated against real HTTPS servers and played an important role in ensuring attention to chosen-ciphertext attacks. See also [**21**] demonstrating continued exploitability of essentially the same attack against some servers two decades later.

Sometimes the literature suggests that it suffices to aim for security against *the oracles provided by applications*.[3] But this would be an evaluation nightmare. One would have to check all the different ways that applications handle signatures and decapsulations, consider how this can change in the future, and then evaluate whether a cryptographic system is secure in all of these contexts. So the community asks for EUF-CMA signature systems and for IND-CCA2 KEMs, rather than for something weaker.[4]

The literature often presents a simpler justification for stronger security models: namely, the blanket statement that it is always better (e.g., "more conservative") to ask for security in stronger models.[5] This blanket statement goes far beyond saying that it is better to ask for IND-CCA2 than for IND-CPA: it also implies that any proposal to replace IND-CCA2 with stronger model M_1 should be accepted, and then any proposal to replace M_1 with a stronger model M_2 should be accepted, and so on. This is its own form of evaluation nightmare.

The critical question to ask is how to manage the risk of real-world security failures so as to best protect real users from attack. The answer cannot be to devote more and more security-analysis resources to more and more obscure risks: time taken chasing a neverending series of academic targets is time taken away from ensuring more important security properties. This does not imply, however, that the right answer is to stop with EUF-CMA and IND-CCA2.

[3] See, e.g., [**60**]: "We conclude that the CNS attack is a concern for the ISO 9796-2 signature scheme with partial message recovery in environments where the attacker is capable of obtaining the signatures of a significant number (e.g., one million) of chosen messages. In environments where the attacker is not capable of obtaining these signatures, the CNS attack is not a concern."

[4] Exception: In the context of protocols that use the cryptosystem key just once, such as the SIGMA approach to secure sessions, the literature often encourages targeting merely IND-CPA. See [**47**] for a recent example. On the other hand, it is a mistake from a systems-security perspective to give users (1) a cryptosystem designed for IND-CCA2 and (2) a non-IND-CCA2 cryptosystem designed merely for IND-CPA. As [**56**] put it: "CPA vs CCA security is a subtle and dangerous distinction, and if we're going to invest in a post-quantum primitive, better it not be fragile."

[5] Occasionally exceptions are made for security notions *proven* to be unachievable.

1.1. Fragility. Beyond EUF-CMA security and IND-CCA2 security, NIST's call for submissions said that "additional security properties ... would be desirable". Let's focus on the last item in NIST's list:

> A final desirable, although ill-defined, property is resistance to misuse. Schemes should ideally not fail catastrophically due to isolated coding errors, random number generator malfunctions, nonce reuse, keypair reuse (for ephemeral-only encryption/key establishment) etc.

In 2018, a catastrophic failure was reported in Dilithium because of an isolated coding error in the official Dilithium software. Specifically, the software generated random values incorrectly, reusing randomness at a place where the specification instead generated new randomness; [**58**] announced that this "reuse of randomness can easily be exploited to recover the secret key". Evidently Dilithium fails to provide "resistance to misuse".

On the other hand, it is difficult to imagine how *any* scheme could prevent "isolated coding errors" from causing catastrophic failures,[6] never mind all the other forms of potential "misuse". Did NIST have some reason to think that "resistance to misuse" could be achieved?

Perhaps the intent was not to ask the yes-no question of whether one can construct a misuse scenario, but rather the tricky risk-assessment question of *how likely* it is for people to make mistakes that will cause a scheme to fail. It could be that other cryptographic systems are *more* failure-prone than Dilithium, and that the official Dilithium software was simply unlucky.

It is not easy to evaluate such a complicated, open-ended security "property". The lack of a clear definition violates the following Katz–Lindell [**53**] statement: "One of the key intellectual contributions of modern cryptography has been the realization that formal definitions of security are *essential* prerequisites for the design, usage, or study of any cryptographic primitive or protocol." It is also easy to see how an attacker can use this "property" as a tool to attack cryptosystem-selection processes, promoting weaker cryptosystems by selectively objecting to stronger cryptosystems.[7]

[6] A standard could insist that implementors take a majority vote of three independent implementations, but experience shows that there are correlations among errors from different implementors. Furthermore, a coding error could replace the majority vote with taking just the result of the first implementation, or an implementor could "misuse" the scheme by taking just one implementation; either way, a coding error in that implementation could cause disaster even if other implementations are perfect.

[7] In its latest report [**1**], NIST criticized Classic McEliece for a "misuse scenario" where "reusing the same error vector when encapsulating for multiple public keys" would damage security—even though (1) there have been no examples of this scenario occurring for Classic McEliece, (2) the official Classic McEliece software has always used RNGs correctly, and (3) no encapsulation mechanism is safe against external RNG failures. Meanwhile none of NIST's reports criticized Dilithium for the "misuse scenario" of reusing randomness inside a single signature—even though (1) this scenario occurred in the official Dilithium software, (2) this destroyed the security of that software, and (3) the problem was in that software, not in an external RNG.

On the other hand, the literature provides clear reasons to believe that some cryptographic systems are more failure-prone than others. For example, for ECDH systems that transmit curve points in affine coordinates (x, y), there are endless reports (e.g., [**18**]) of implementations that fail to check whether the incoming point is on the curve, and that are easily breakable as a result. This attack is structurally eliminated by ECDH systems that (as in [**6**] and [**7**]) choose twist-secure curves and transmit merely x.

Presumably there are also ways to adjust design decisions in post-quantum cryptography to reduce the chance of implementation failures. It is important to keep in mind here that there is far less evidence available today regarding post-quantum implementation failures than there is regarding pre-quantum implementation failures, and the general difficulty of evaluating implementation security means that claims of security improvements need to be investigated carefully before they are used for making decisions. This is not a reason to avoid study of the topic.

1.2. Natural DRAM faults. In 2009, Schroeder, Pinheiro, and Weber [**72**] reported the results of a 2006–2008 study of failure rates in the DRAM in "the majority of machines in Google's fleet". The observed failure rates were "25,000 to 70,000 errors per billion device hours per Mbit".

Conventional SECDED ECC DRAM encodes 64 bits of logical data in 72 bits of physical DRAM, using a distance-4 linear error-correcting code.[8] "SECDED" here means "single-error correcting, double-error detecting", and "ECC" means "error-correcting code". In particular, SECDED ECC DRAM corrects any single bit flip, while reporting the correction to the operating system. Some computer buyers make sure to buy SECDED ECC DRAM; this is also how the study from [**72**] collected data.

However, most computing devices today simply store 64 bits of logical data in 64 bits of physical DRAM. Any single physical bit flip is then a logical bit flip, directly corrupting data, with no warning to the user. For example, flipping a single bit in DRAM can silently convert the ASCII letters "NTRU" to "NTRW".

Consider a reasonably popular cryptosystem that, worldwide, has a billion active 256-bit keys stored in DRAM without SECDED. An extrapolation from the error rates reported in [**72**] suggests that between 50000 and 140000 of those keys will have a bit flipped each year.[9] This is frequent enough to mandate investigation of the security consequences.

[8] This 12.5% overhead is not the best that can be done. The overhead of a distance-4 error-correcting code, such as an extended Hamming code, drops as the dimension increases. DRAM today is normally accessed in 512-bit blocks ("lines"), larger than the 64-bit blocks conventionally used for SECDED. A 512-bit line encoded as 528 bits can be stored as 16 bits on each chip in a 33-chip module, which in principle should cost just 3.125% more than a 32-chip module; and 523 bits are enough to encode 512 bits with SECDED, as noted in, e.g., [**80**].

[9] Presumably this is an underestimate of the error rate: one would not expect average user devices to be as reliable as Google's air-conditioned, systematically monitored, frequently replaced servers.

1.3. Contributions of this paper. This paper shows that, in the IND-CCA2 attack model augmented to include a one-time flip of one bit stored by the legitimate user, NTRU-HRSS is devastatingly insecure: there is an efficient attack that recovers the NTRU-HRSS session key. In a multi-target IND-CCA2 attack model similarly augmented to include a one-time single-bit fault, the same attack efficiently recovers all of the NTRU-HRSS session keys that were encapsulated to the targeted public key before the fault.

Section 4.2 presents the full multi-target attack. For verification, as a supplement to this paper, attack software is provided that carries out the multi-target attack against the official NTRU-HRSS software, using a simulation of the required fault. See Section 2 for a comparison of this paper's attack to previous fault attacks.

Section 4.3 formulates analogous fault attacks against Streamlined NTRU Prime and Classic McEliece, and explains why both of those attacks are blocked by plaintext confirmation, a CCA defense already built into the CCA conversions inside those cryptosystems. (This should not be interpreted as a claim that Streamlined NTRU Prime and Classic McEliece are immune to *all* fault attacks.) See Section 3 for a survey of chosen-ciphertext attacks and defenses.

Interestingly, NTRU-HRSS had included the same CCA defense in its original design, but then *removed the defense on the basis of papers claiming to have proven that the defense was not necessary*. See Section 4.4. Those papers were considering a more limited attack model.

1.4. Acknowledgments. This paper is inspired by a series of discussions with Tanja Lange regarding IND-CCA2 attacks and defenses. In particular, Lange pointed out plaintext confirmation as a countermeasure to fault attacks.

2 Fault attacks

This section explains how this paper's fault attack fits into the broader literature on fault attacks.

A fault is like a software bug or a hardware bug in that it complicates analyses of computer behavior: it violates the implicit assumption that each computation is being carried out correctly. As a further complication, a fault is like a physical side channel in that it comes from physical effects whose boundaries are hard to formalize and analyze. Even if a system is secure in the absence of faults, the attacker can hope that the system becomes breakable when faults occur.

2.1. A generic fault attack. If one wants to skip the complications of analyzing physical effects—or if one believes the blanket statement that it is better to ask for security in stronger models; see Section 1—then one might hypothesize that the attacker has the power to induce arbitrary faults in computations. Under this hypothesis, the following generic fault attack extracts the internal secrets from *any* computation whose output is visible to the attacker.

View the computation as an unrolled circuit consisting of NAND gates, and consider a NAND gate $a, b \mapsto 1 - ab$ producing output at the end of the computation. If the output is 0 then $a = b = 1$. If the output is 1 then the attack deduces a, b by re-running the computation with a bit-flip fault on a and then with a bit-flip fault on b. The attack now knows the inputs to the NAND gate.

The attack then similarly targets the inputs to an earlier NAND gate that produced a, while using a set-to-1 fault to force $b = 1$ so that changes in a are visible as changes in the output $1 - ab$. (Set-to-1 faults can also be used in place of the bit-flip faults in the previous paragraph.) The attack proceeds upwards in the same way through each NAND gate, extracting the entire internal state of the computation.

The number of runs of the computation is $\Theta(n)$ where n is the circuit size. Each run uses $O(d)$ faults to ensure that the targeted bit is visible in the output, where $d \le n$ is the circuit depth.

Internal checks in the computation, such as verifying signatures before releasing them, do nothing to stop this attack: checks are just like any other computation in succumbing to faults. Randomizing the computation simply requires the attacker to apply further faults to zero the randomness. Destroying the device after 1000 computations requires keeping track of the number of computations, and the attacker can apply faults to zero that number. Destroying the device after *one* computation does not require a counter but still requires triggering a self-destruct mechanism, and the attacker can apply faults to clear the trigger.

2.2. Specializing, optimizing, and demonstrating the generic fault attack. A typical fault attack in the literature can be viewed as

- specializing the generic attack from Section 2.1 to a particular target and
- optimizing the specialized attack so that the attacker does not need to induce as many faults.

The resulting attacks vary in how many faults they use and in how precisely targeted those faults are.

Sometimes fault-attack papers include real-world demonstrations that one can produce the necessary faults by, e.g., heating a circuit, firing lasers at the circuit, etc.; see, e.g., [**30**]. Sometimes faults can be induced by software; see, e.g., [**73**].

For most attacks, one cannot reasonably expect the requisite faults to occur naturally. One can try to stop these attacks by cutting off data flow that the attacker might be able to use to induce faults in the legitimate user's computation. This includes keeping the attacker physically away from the device, and constraining software behavior so as to avoid faults.

2.3. Natural-fault attacks. Occasionally a fault attack relies on such a small number of faults that one *can* expect naturally occurring physical effects to produce the requisite faults. Eliminating the attacker's ability to induce faults does nothing to stop an attack of this type. The classic example, pointed out by Boneh, Demillo, and Lipton [**23**], is as follows.[10]

[10] As a different example of using just one fault, consider the IND-CCA2 game for KEMs. The attacker is free to send a ciphertext with one bit flipped, and to inspect

The job of an RSA signer is to compute an eth root s of h modulo pq, where (pq, e) is the public key and h is a hash of the message being signed. This is the same as computing $s = h^d \bmod pq$ for a suitable decryption exponent d. “RSA-CRT”, the usual speed-oriented choice of RSA signature algorithm, computes an eth root s_p of h modulo p as $h^{d_p} \bmod p$ where $d_p = d \bmod (p-1)$, similarly computes an eth root s_q of h modulo q, and combines s_p with s_q to obtain s.

Now say the signer signs the same message again, but this time there is a fault in the computation of s_p—anything that produces a wrong result; for example, a bit flip in d_p. The resulting signature S will then be the same as the correct signature s modulo q but not modulo p, and the attacker can compute q as $\gcd\{S - s, pq\}$.

A variant by Lenstra [**57**] is to compute q as $\gcd\{S^e - h, pq\}$. This variant assumes that the attacker also sees the message being signed, but avoids the need for multiple signatures of one message, so the attack works with passive observation of objects that are normally sent in the clear, namely messages and signatures.

One of the countermeasures suggested in [**23**] is to check signatures before releasing them. In real-world RSA, the encryption exponent e is chosen to be small, so the check adds very little to the cost of signing. But typical RSA descriptions don’t include this check, and typical tests of RSA software don’t detect the check, so it is easy to imagine RSA software being deployed without the check.

Sullivan–Sippe–Heninger–Wustrow [**75**] announced in 2022 that they had exploited faults to extract “private RSA keys associated with a top-10 Alexa site” and “browser-trusted wildcard certificates for organizations that used a popular VPN product”. Some hosts were observed in [**75**, Section 5.3] to produce faulty signatures over months or even years, suggesting “that the errors they experience are persistent: disk corruption or memory corruption affecting the private key.” Other faults were transient; perhaps a secret key was copied from disk to DRAM, then a bit flipped in DRAM, and then the same DRAM was reused for other data, wiping out the flipped bit. On the other hand, [**75**] reported unsuccessfully trying some possibilities for flipped bits. A different possibility considered in [**75**] is “failing hardware”.

2.4. Algorithm dependence in natural-fault attacks. At the time of [**23**], the primary RSA specification was PKCS #1 v1.5, released in 1993. Secret keys were specified to have the following components (see [**51**, Section 7.2]):

- the public key n,
- the encryption exponent e,

the resulting session key; now simply hypothesize that a fault flips the bit back at the beginning of decapsulation. One reason that this is a less satisfactory example than [**23**] is that it requires a specific fault to occur during a narrow window of time, while a fault in a stored secret key at any moment—something more likely to occur naturally—opens up the attack of [**23**].

- the decryption exponent d,
- the secret primes p and q,
- the integers d_p and d_q, and
- the inverse of q modulo p.

There are many ways to double-check these secret keys so as to detect flipped bits: check whether n matches pq, check whether d_p matches d mod $(p-1)$, check whether ed is 1 modulo $p-1$, etc. With more work one can *correct* flipped bits (and also correct any errors that might occur inside the signing computation).

Consequently, the fault attack from [**23**] was an attack against *some* algorithms computing the specified signing function. Stopping the attack required changing commonly used algorithms (for example, to check signatures as mentioned above), but did not require a new specification of the signing function,[11] new test vectors, etc.

As another example of algorithm dependence in natural-fault attacks, consider the following three versions of the Ed25519 signature system:

- In standard Ed25519 (see [**49**]), the secret key is a 32-byte string that is hashed to obtain (1) a secret scalar and (2) another secret that is hashed together with the message to obtain a nonce. Any bit flip in the stored secret key will produce completely different hash output, leading to garbage signatures of no evident value for the attacker.
- In the most commonly used variant of Ed25519, the secret key is 64 bytes: the same 32-byte string as above, plus a copy of the 32-byte public key. With the simplest signing algorithm, a fault in these 64 bytes will leak the secret key. This is an algorithm-dependent attack; a signing algorithm that double-checks the secret scalar against the public key will detect the fault.
- A variant of Ed25519 with a more efficient fault-attack countermeasure incorporates another 32 bytes of randomness into the input to the hash producing the nonce, without the cost of checking the public key. This variant was considered in, e.g., [**9**] and (as a fault-attack countermeasure) [**66**, Section 8].

To summarize, the availability of fault attacks is sensitive to details of (1) the cryptosystem at hand and (2) the algorithms used for that cryptosystem.

2.5. Comparison. Like the attack from [**23**] against RSA-CRT, this paper's attack against NTRU-HRSS works if a single bit is flipped in a stored secret key, an event that will occur naturally for some users.

Unlike the attack from [**23**], this paper's attack has the further feature of being algorithm-independent: it works against *any* algorithm that computes the specified function of the secret key. The NTRU-HRSS secret key does not contain any data that a decapsulation algorithm can use to detect the fault exploited in this paper's attack. This paper's attack against NTRU-HRSS is thus a decapsulation-algorithm-independent natural-fault attack.

[11] Perhaps the signing function could have been changed to reduce the chance of problems—see Section 1.1—but this is a separate issue.

A disadvantage of this paper's attack (compared to the attack from [**23**] with the improvement of [**57**]) is that it is active. The attack takes full advantage of the flexibility of the attack model: for each target ciphertext, the attack sends some modified versions of the ciphertext before and after the fault occurs, and sees some information about the resulting session keys. Hopefully the application does not actually provide so much flexibility to the attacker. On the other hand, the rationale for asking for IND-CCA2 security (see Section 1), rather than investigating whether IND-CCA2 security is overkill for applications, applies with equal force when one extends the IND-CCA2 attack model to include a natural fault. It is interesting that the IND-CCA2 security of NTRU-HRSS is so fragile in the presence of natural faults.

Another disadvantage of this paper's attack is that it is recovering only session keys, not Alice's secret key. On the other hand, the reason an attacker wants to recover Alice's secret key is to be able to recover all session keys; this attack recovers all session keys that were communicated before the fault.

2.6. The cold-boot argument against error correction. The literature on cold-boot DRAM attacks often uses redundancy in stored data to correct flipped bits; see, e.g., [**37**, Section 5]. This is occasionally used as an argument that secret data should be stored in maximally compressed format; see, e.g., [**37**, Section 8, "suggested countermeasures", including "avoiding precomputation"]. The same argument implies that users should *not* include redundancy in data to detect and correct errors, and in particular should *not* use SECDED ECC DRAM; [**37**, Section 3.4] says "ECC memory could turn out to *help* the attacker".

However, users who avoid SECDED ECC DRAM are exposed to a large class of hard-to-analyze correctness risks and security risks that they would otherwise have avoided. Meanwhile it is clear that well-executed cold-boot DRAM attacks rarely encounter errors in the first place (see, e.g., [**37**, Table 2, "no errors" entries]) and are thus *not* stopped by attempts to avoid redundancy.

Encrypting DRAM, using a key stored in better-protected hardware, is a simpler and much more convincing defense to cold-boot DRAM attacks. Encrypting DRAM is also compatible with SECDED ECC DRAM and other protections against faults.

3 Chosen-ciphertext attacks and defenses

This section surveys the general structure of chosen-ciphertext attacks against code-based and lattice-based systems, and of various cryptosystem features that *seem to* interfere with these attacks. Beware that the literature often overstates the extent to which (some of) these features are *known* to interfere with these attacks; see Section 4.4.

NTRU-HRSS, Streamlined NTRU Prime, and Classic McEliece are used as running examples, abbreviated `ntruhrss`, `sntrup`, and `mceliece` respectively. Table 3.1 summarizes the features of these cryptosystems.

feature	`mceliece`	`sntrup`	`ntruhrss`
hashing the plaintext	yes	yes	yes
rigidity	yes	yes	yes
no decryption failures	yes	yes	yes
plaintext confirmation	yes	yes	no
implicit rejection	yes	yes	yes
hashing the ciphertext	yes	yes	no
limited ciphertext space beyond small plaintext	no	yes	no
limited plaintext space beyond small plaintext	no	no	no
no derandomization	yes	yes	yes

Table 3.1. Cryptosystem features that *seem to* (but do not necessarily) interfere with chosen-ciphertext attacks. The `mceliece`, `sntrup`, and `ntruhrss` columns indicate whether the features appear in Classic McEliece, Streamlined NTRU Prime, and NTRU-HRSS respectively. All entries are for the current versions of `mceliece`, `sntrup`, and `ntruhrss`; implicit rejection appeared in `sntrup` and `ntruhrss` in 2019, while plaintext confirmation was removed from `ntruhrss` in 2019. See text for definitions of the features.

3.2. Ciphertext structure. Throughout this section, Bob's ciphertext has the form $B = bG + d$, where G is Alice's public key and b, d are secrets, in particular with d chosen to be small. The choice of letters here is as in [**10**, Section 8], unifying notation between ECDH, "noisy DH" lattice-based and code-based systems, and further lattice-based and code-based systems.

The `mceliece` description uses an optimized ciphertext structure due to Niederreiter: simply He, where H is the public key and e is small. However, H internally consists of two parts, an identity matrix and another matrix Q, so He can be written as $e_1 + Qe_2$. This is, modulo transposition and relabeling, again a ciphertext of the form $bG + d$.

3.3. Decryption. Alice uses her private key to recover b and d. Let's assume at the outset that this recovery process is labeled as a PKE returning plaintext (b, d).

The original McEliece system [**59**] instead viewed b as the plaintext—not required to be small—and d as something chosen randomly in encryption. The original NTRU system [**40**] instead viewed d as the plaintext and b as something chosen randomly in encryption. A 1996 NTRU handout [**41**, Section 4.2] had also considered a deterministic PKE with (b, d) as the plaintext—although this handout was not put online until 2016, after deterministic NTRU PKEs had already been recommended in, e.g., [**8**].

Linear algebra easily recovers b from bG (assuming G is public and injective), but recovering b from a noisy multiple $bG + d$ is conjectured to be hard (for appropriate choices of parameters). This conjecture is often described as conjectured hardness of the "LPN", "LWE", "Ring-LPN", "Ring-LWE", "Module-LPN", or "Module-LWE" problems (again for appropriate choices of parameters), where the choice of name depends on various details of the algebraic structure containing G. These problems, in turn, are typically claimed to have been introduced in various 21st-century papers. However, the original McEliece [59] and NTRU [40, Section 3] papers had already analyzed the cost of various algorithms for the cases of LPN and Ring-LWE that matter for those cryptosystems, so it is wrong to credit those problems to subsequent papers. There is some value in generalizing the problems (for example, to study other cryptosystems), but credit for the general problems has to include credit to the cases considered earlier.

The rest of this paper ignores the possibility of recovering (b, d) purely from $(G, bG+d)$, and instead focuses on the extra power of chosen-ciphertext attacks.

3.4. Exploiting linearity for chosen-ciphertext attacks. Given the linear structure of a ciphertext $B = bG+d$ and the definition of IND-CCA2 security,[12] the obvious attack sends a modified $B' = B + \delta = bG + d + \delta$ for some small nonzero δ. The attacker hopes that the decryption process successfully returns $(b, d + \delta)$, at which point the attacker simply subtracts δ and wins.

Decryption does not work for arbitrarily large d. This is why the attacker chooses δ to be small.

For example, a `mceliece` decoder requires (b, d) to have a specific Hamming weight. The attacker chooses a random weight-2 vector δ, a vector of the form $(0, \dots, 0, 1, 0, \dots, 0, 1, 0, \dots, 0)$. There is then a good chance that $(b, d + \delta)$ has the right weight, meaning that decryption returns $(b, d + \delta)$. Various features described below are included in `mceliece` to stop this attack.

In the same example, the attacker can, more generally, choose $B' = B+\beta G+\delta$ where β, δ have total weight 2. To simplify notation, the comments below focus mainly on $B' = B + \delta$, but similar comments apply to $B' = B + \beta G + \delta$.

3.5. Feature 0: hashing the plaintext. As a preliminary step in limiting the information provided to chosen-ciphertext attacks, let's switch from a PKE to a KEM that hashes the plaintext.

Specifically, let's define encapsulation to choose the input (b, d) randomly (not necessarily uniformly; other distributions can be more convenient), and let's define decapsulation to return a hash $H(b, d)$. The attacker sending $B + \delta$ and receiving $H(b, d + \delta)$ has no obvious way to reconstruct or otherwise recognize $H(b, d)$, unless the hash function H is remarkably weak.

Let's assume from now on that the goal is to build a KEM that resists chosen-ciphertext attacks. This was the target for most encryption submissions to the NIST Post-Quantum Cryptography Standardization Project, and in particular is the target for `ntruhrss`, `sntrup`, and `mceliece`. Internally, each of these KEMs is built from a PKE that produces ciphertext $bG + d$ and recovers (b, d), or something equivalent to (b, d), during decryption.

Generic transformations convert any KEM into various other cryptographic objects. For example, in the paper [74] that introduced the KEM abstraction (and specifically KEMs that hash the plaintext), Shoup built a PKE handling

[12] Beware that there are several slightly different definitions of IND-CCA2 security for PKEs. See generally [5].

variable-length user messages by using a KEM to encapsulate a session key and then using symmetric cryptography to encrypt user data under that key.

There are arguments against using KEMs. For example, the literature explains how to build a variable-length PKE with smaller ciphertexts by encoding some of the user data inside the input to a fixed-length PKE: in particular, encoding some user data inside (b, d). The usual approach is to take some randomness and some user data, apply an "all-or-nothing transform" (see generally [70]), and encode the result as (b, d); decryption reverses these steps. However, the space savings seems less important than the simplification of independently analyzing a KEM layer. All-or-nothing transforms might still be useful inside KEM designs; see Section 3.14 below.

3.6. Probing the boundaries of successful decryption. Hashing by itself does not stop chosen-ciphertext attacks. The main issue is that the attacker sending $B+\delta$ does not always receive a hash of $(b, d+\delta)$. Sometimes $d+\delta$ is too large to be decoded successfully, and then decapsulation returns a failure report instead of a hash.

The pattern of successes and failures is valuable information for the attacker. For example, consider again the `mceliece` decoder, which works exactly when (b, d) has a specific weight. If adding $\delta = (0, \ldots, 0, 1, 0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{F}_2^n$ to d preserves weight then exactly one of the two 1 positions must match a position set in d. Seeing enough such δ quickly reveals all of the positions in d. One can try to accelerate this by using each failing δ as a statistical indication that both 1 positions are likely to be unset in d, but the attack works quickly in any case.

This attack against the original McEliece system was introduced by Hall, Goldberg, and Schneier in [38] and by Verheul, Doumen, and van Tilborg in [78] (which says it was submitted in 1998, before [38] appeared). To be more precise, this is essentially the attack in [78, Section 4]; the attacks in [38, Section 2] and [78, Section 3] are variants that assume that the decoder works when d has *at most* a specific weight.

As another example, `ntruhrss` chooses $d \in \mathbb{Z}[x]/(x^n - 1)$ as $x - 1$ times a polynomial T with coefficients in $\{-1, 0, 1\}$, and checks the same condition during decapsulation. Adding $\delta = 2(x-1)$ changes T to $T+2$, which works when the constant coefficient of T is -1 and seems very unlikely to work otherwise; adding $\delta = -2(x-1)$ works when the constant coefficient of T is 1; adding $\delta = 2x(x-1)$ works when the next coefficient of T is -1; etc.

3.7. Probing as an attack against the secret key. Failure patterns have further consequences for PKEs that are not *rigid*, meaning that the specified decryption function can successfully decrypt multiple ciphertexts to the same plaintext.

For example, recall that the original NTRU system has just d as a plaintext, with b chosen randomly in encryption. A closer look at the system reveals that decrypting $B + \beta G$ for small β has a good chance of producing d—there are multiple ciphertexts that produce the same plaintext—and then the resulting session key is exactly the legitimate user's session key, breaking IND-CCA2.

Even worse, the pattern of successes and failures for small β reveals the secret key. Here the attacker does not need to see any information about the session keys except for knowing which $B + \beta G$ succeeded and which failed. This paper suppresses details of this attack, aside from mentioning that it is easiest here for the attacker to begin with a known (b, d). Attacks of this type against NTRU were published by Hoffstein–Silverman [42] and Jaulmes–Joux [48]; many subsequent variants have appeared.

An analogous problem occurs for PKEs that have *decryption failures*, meaning that the specified decryption function will sometimes fail to decrypt a legitimate ciphertext to the original plaintext. For example, the original NTRU system had a noticeable frequency of decryption failures, and this was exploited by Howgrave-Graham, Nguyen, Pointcheval, Proos, Silverman, Singer, and Whyte [44] to recover the secret key.

3.8. Feature 1: rigidity. The first step in limiting the power of probing is to choose a rigid PKE, so that multiple ciphertexts cannot produce the same plaintext. It is easy to convert any deterministic PKE into a rigid PKE by modifying decryption to reencrypt the plaintext and to check the result against the ciphertext.

All of `ntruhrss`, `sntrup`, and `mceliece` are designed as rigid PKEs starting from deterministic PKEs, although not always with an obvious step of reencrypting via the encryption procedure:

- Simple facts about error-correcting codes are used inside `mceliece` to accelerate the reencryption procedure. The resulting algorithm uses, asymptotically, an essentially linear number of operations, and avoids storage of the public key inside the private key.[13]
- For `ntruhrss`, the reencryption procedure is optimized to share a multiplication with the original decryption algorithm.
- For `sntrup`, the original decryption algorithm automatically avoids the analogous multiplication (since d is chosen by rounding), and reencryption simply calls the same procedure as encryption.

What matters for this feature is not whether there is a visible reencryption step, but whether the resulting PKE is rigid; this is why Table 3.1 lists “rigidity” rather than “reencryption”.

3.9. Feature 2: no decryption failures. The second step in limiting the power of probing is to choose a PKE where the specified decryption function always recovers the original plaintext from the corresponding ciphertext. There are no decryption failures in `ntruhrss`, `sntrup`, and `mceliece`.

Note that “no decryption failures” refers to decryption failures for ciphertexts obtained from the encryption algorithm. Decryption can still fail for other ciphertexts created by the attacker.

[13] See generally [12, Section 8]. Even better, the usual decoding algorithm inside `mceliece` is shown in [12, Section 7] to be rigid even without reencryption. However, [12, Section 8.4] recommends reencryption for robustness.

If a rigid PKE has no decryption failures then it decrypts exactly the ciphertexts $bG+d$ for a key-independent set of valid plaintexts (b, d). An attacker replacing $B = bG + d$ with $B' = B + \beta G + \delta$ will obtain a valid ciphertext if $(b+\beta, d+\delta)$ is in the same key-independent set, and presumably will not obtain a valid ciphertext if $(b+\beta, d+\delta)$ is not in this key-independent set. Otherwise some valid $(b', d') \neq (b+\beta, d+\delta)$ has $B' = b'G+d'$, i.e., $(b+\beta-b')G+(d+\delta-d') = 0$; but it is supposed to be hard for the attacker to find small nonzero s, t such that $sG + t = 0$. Taking large β or large δ seems even less useful.

In short, there is no obvious way for the attacker to find (β, δ) where failures will provide any information about the secret key. In the absence of such information, the secret key is protected against the attack of Section 3.7.

However, the attacker can still target the legitimate user's plaintext (b, d) via the attack from Section 3.6. This is addressed in Section 3.10.

3.10. Feature 3: plaintext confirmation. The third step in limiting the power of probing is to replace the ciphertext B with $(B, H'(b, d))$, where H' is another hash function, and to check $H'(b, d)$ on decryption.

The point of plaintext confirmation is to prevent the attacker from modifying a ciphertext for the legitimate user's secret (b, d) into a ciphertext for $(b, d + \delta)$. The attacker can replace B with $B+\delta$, but has no obvious way to replace $H'(b, d)$ with $H'(b, d + \delta)$ without first finding (b, d). If the attacker knew (b, d) then the attacker could compute the session key $H(b, d)$ without bothering to carry out a chosen-ciphertext attack. An attacker can still choose (b, d) and modify the resulting ciphertext to try to attack the secret key, but this is addressed by a rigid PKE without decryption failures; see Section 3.9.

Typically H and H' are both chosen as a cryptographic hash function applied to separate input spaces: $H(b, d) = F(1, b, d)$ and $H'(b, d) = F(2, b, d)$. An alternative is to choose H and H' as the left and right halves of the output of a cryptographic hash function: $F(b, d) = (H(b, d), H'(b, d))$. Obviously one must not select H' as H, or as any other function whose outputs reveal the H outputs on the same inputs; see [4] for examples of attacks against real proposals where H and H' were not adequately separated.

3.11. Feature 4: implicit rejection. An alternative to plaintext confirmation is "implicit rejection". This means replacing any failure output for a ciphertext B with a string $H(r, B)$, where r is a random string, part of Alice's secret key.

The idea is that replacing the failures with random garbage hides the pattern of successfully modified ciphertexts. The attacker sees $H(b, d+\delta)$ in success cases and $H(r, B + \delta)$ in failure cases, and—without knowing (b, d) in advance—has no way to distinguish these situations.

For comparison, plaintext confirmation stops the attacker's $B + \delta$ from being a valid ciphertext. These features are compatible: one can use implicit rejection to hide the pattern of successes, and use plaintext confirmation to limit the attacker's ability to create a pattern of successes in the first place.

With implicit rejection, care is required to avoid leaking the pattern of failures through timing. A typical approach starts with B, computes (b, d) in constant time along with a bit indicating failure, computes $H(r, B)$, computes $H(b, d)$, and uses the bit to select either $H(r, B)$ or $H(b, d)$ in constant time.

More generally, one can replace any failure output with $R(B)$, where R is a secretly keyed function producing output of the same length as the normal hash outputs $H(b, d)$. Well-studied message-authentication codes are faster than general-purpose hash functions.

Implicit rejection was introduced by Persichetti [**67**] in the McEliece context, and generalized by Hofheinz–Hövelmanns–Kiltz [**43**].

3.12. Feature 5: hashing the ciphertext. Instead of choosing the session key as $H(b, d)$, one can choose it as $H(b, d, B)$ where B is the ciphertext. If an attacker-chosen $B + \delta$ decrypts to the same (b, d) then the resulting session key $H(b, d, B + \delta)$ will be different from $H(b, d, B)$.

This extra hash input hides any collisions produced by decryption. For comparison, reencryption creates rigidity, preventing any collisions from appearing in the first place. These features are compatible. Note the analogy to implicit rejection hiding the pattern of successfully modified ciphertexts while plaintext confirmation eliminates those ciphertexts.

For implementors, a convenient feature of using $H(b, d, B)$ for a valid session key and $H(r, B)$ for implicit rejection is that one can easily merge the hash calls if r has the same length as (b, d). Security analysis is slightly easier if a valid session key uses $H(1, b, d, B)$ and implicit rejection uses $H(0, r, B)$; this still allows the same merging.

Hofheinz–Hövelmanns–Kiltz [**43**] observed that ciphertext hashing changed what they could prove regarding security. See [**16**, Appendix A.5] for an example of a broken cryptosystem that seems to be rescued by ciphertext hashing.

3.13. Feature 6: limited ciphertext space. Another way to reduce the attacker's ability to modify ciphertexts is to force legitimate ciphertexts $bG + d$ to be in a constrained set checked by Alice.

For example, `sntrup` chooses b randomly, and then rounds each entry of bG to the nearest multiple of 3 to obtain $B = bG + d$; each entry of d is -1 or 0 or 1. The ciphertext format enforces the multiple-of-3 rule, so an attacker's modified ciphertexts also have to follow this rule.

An advantage of constraining ciphertexts via the ciphertext format is that this constraint does not rely on Alice's decapsulation algorithm. This does not mean that the constraint is as effective as other defenses. Typically such ciphertext constraints are presented as a way to reduce the use of randomness and reduce ciphertext sizes; there is very little cryptanalytic literature considering the extent to which these constraints interfere with chosen-ciphertext attacks.[14]

[14] Given recent misinformation regarding rounding, it seems necessary to emphasize that the cryptanalytic question here is whether rounding is *stronger* than adding random errors: this attack avenue *obviously* works against random errors, whereas analysis is required of the extent to which the attack avenue is blocked by rounding. See also [**69**], which finds that rounding complicates side-channel-assisted chosen-ciphertext attacks.

Here is another example of constraining the set of ciphertexts. Recall that McEliece's original cryptosystem has ciphertexts $bG+d$ where b is arbitrary and d is small. To constrain $bG+d$ to a linear subspace V, first choose a small d and then find, by linear algebra, b for which $bG+d \in V$. It is easy to select V so that b always exists and is unique, and it is easy to show that these constrained ciphertexts $bG+d$ are equivalent to Niederreiter's ciphertexts.

As noted above, Niederreiter's ciphertexts can also be viewed as having the form $bG+d$, where different variables are now labeled as b, d, G, and where (b, d) is required to be small. One could further constrain $bG+d$ to a limited subspace by choosing b randomly and then finding a small d for which $bG+d$ is in that subspace; this means solving a decoding problem for that subspace.

In Table 3.1, "limited ciphertext space beyond small plaintext" means that $bG+d$ is constrained *beyond* requiring small (b, d), so `mceliece`'s use of Niederreiter's ciphertexts does not qualify, whereas further constraining $bG+d$ as in the previous paragraph would qualify.

3.14. Feature 7: limited plaintext space. One last way to reduce the attacker's ability to modify ciphertexts is to limit the space of plaintexts (b, d).

In the standard attacks, the attacker is choosing (β, δ) so that the target plaintext (b, d) has a noticeable chance of $(b+\beta, d+\delta)$ also being a plaintext. Constraining the plaintext space can reduce this chance to something negligible.

Typically there is a reasonably efficient way to compress (b, d) into an s-bit string where the number N of choices of (b, d) is not far below 2^s. Normally N, and therefore 2^s, is much larger than 2^{256}. A standard way to sample from a "structureless" set of s-bit strings is as follows: start with a 256-bit string, zero-pad to s bits, and then apply an all-or-nothing transform. One can then try decompressing the resulting s-bit string to (b, d); if this fails then one can try again with a new 256-bit string. Unless there is some surprising interaction between the all-or-nothing transform and the compression mechanism, each try will succeed with probability approximately $N/2^s$, and one can statistically check this with experiments.

Alice, upon decrypting a ciphertext to obtain (b, d), compresses (b, d) to s bits, inverts the all-or-nothing transform, and checks for the zero-padding. Defining hashes in terms of the 256-bit string instead of (b, d) forces implementations to invert the all-or-nothing transform, although one still has to worry that implementations will skip the zero-padding check.

Presumably an all-or-nothing transform is overkill here, since most of the structure in the plaintext (b, d) is not easy to see in ciphertexts $bG+d$. It would be interesting to identify the relevant security properties of plaintext sets, and to optimize construction algorithms and recognition algorithms for secure sets.

An alternative way to limit the plaintext space is as follows. Take any algorithm to randomly generate (b, d), and compose it with any cryptographic random-number generator producing the necessary bits of randomness from a 256-bit seed. This is generally hard to invert, but one can transmit, as part of the ciphertext, the seed encrypted under a hash of (b, d).

Because of various patent issues that remain unresolved at the time of this writing,[15] I'm currently limiting time spent investigating Kyber [2] and other cryptosystems in the GAM/LPR family. However, it is interesting to note that this family relies on the seed approach for another reason, namely "derandomization". Care is required here regarding security: my paper [11] gives examples of cryptosystems where derandomization loses about 100 bits of security, and the impact of derandomization on GAM/LPR systems requires cryptanalysis. None of `ntruhrss`, `sntrup`, and `mceliece` have this issue. This is reported in the "no derandomization" line in Table 3.1.

4 The NTRU-HRSS attack

This section presents this paper's attack against `ntruhrss`, and describes an accompanying software package `attackntrw` [14] that successfully carries out the attack against existing `ntruhrss` software with a simulated fault.

This section also presents analogous attacks against `sntrup` and `mceliece`, and explains why these attacks are blocked by the plaintext confirmation built into `sntrup` and `mceliece`. This section continues by reviewing how "provable security" led `ntruhrss` to remove plaintext confirmation, and concludes by evaluating possible countermeasures to protect `ntruhrss`.

4.1. Attack model. The attack model considered here is the standard IND-CCA2 attack model for KEMs, plus a one-time bit flip at a uniform random position inside Alice's stored secret key. "One time" means that there is a time at which a bit flips—and then the bit *stays* flipped; the bit does not magically return to its previous value. The attacker can carry out any number of chosen-ciphertext queries to the original secret key before the bit flip and to the new secret key after the bit flip.

The attack below requires the bit flip to occur within 256 specific bits inside Alice's secret key. This does not occur with probability 1, but it does occur with noticeable probability, namely $256/z$, where the secret key has z bits. In the real world, one expects a fault in these 256 bits to naturally occur for the fraction of users described in Section 1.2. Note that padding the secret key, increasing z, would not reduce the number of users affected, although it would reduce $256/z$.

It is easy to see that one can achieve security in this model (unlike the more general fault-attack models reviewed in Section 2) if and only if one can achieve standard IND-CCA2 security without faults: simply change the secret-key format to include error correction, for example with a distance-3 Hamming code or a distance-4 extended Hamming code, and apply an error-correcting decoder inside the decapsulation algorithm. However, a KEM that lacks this feature in its secret-key format might be breakable in this model whether or not it is IND-CCA2. The attack below shows that `ntruhrss` is breakable in this model.

[15] See, e.g., [1, page 18]: "If the agreements are not executed by the end of 2022, NIST may consider selecting NTRU instead of Kyber." There are also various relevant patents that do not seem to be considered in [1], such as CN107566121A.

The attack is actually stated for multi-target IND-CCA2 (plus a one-time bit flip), but readers not familiar with multi-target IND-CCA2 can freely focus on the case of a single target ciphertext.

A weaker starting attack model than IND-CCA2 would suffice for this attack. What the attack needs to see is simply whether specified pairs of session keys are identical within the attacker-chosen ciphertexts.

4.2. Attack details. The available `ntruhrss` software supports one parameter set, namely `ntruhrss701`. The following description focuses on `ntruhrss701` for concreteness.

Eve sees Alice's `ntruhrss701` public key G and any number of legitimate ciphertexts $B_1, B_2, \ldots$. These are elements of the ring $(\mathbb{Z}/8192)[x]/(x^{701} - 1)$, encoded as strings. The `attackntrw` software uses the official `nturhrss701` software[16] to generate a public key and 10 target ciphertexts.

For each j, Eve sends Alice various modified versions (described below) of the legitimate ciphertext B_j, and observes the resulting session keys, as allowed by the (multi-target) IND-CCA2 attack model. To ensure that there is no cheating, the `attackntrw` software carries out decapsulation via an `alice_oracle` function that (1) aborts if the input matches any of $B_1, B_2, \ldots$ and otherwise (2) calls the official `ntruhrss701` software.

A fault then occurs, flipping a bit anywhere inside the implicit-rejection key from Section 3.11, the random string r stored inside the secret key.[17] The `attackntrw` software simulates such a fault by flipping the next-to-bottom bit of the last byte of Alice's secret key; this bit happens to be inside r, and flipping any other bit inside r would also work.

Eve then sends the same modified ciphertexts to Alice, observes the resulting session keys, and performs a simple calculation (described below) to extract the secrets b_j, d_j inside each ciphertext $B_j = b_jG + d_j$. The `attackntrw` software performs this calculation and verifies that the session keys computed by the attack match the session keys obtained from the official `ntruhrss701` software.

Eve's modified versions of B_j have the form $B_j + 2(x-1)x^e$ and $B_j - 2(x-1)x^e$ for $0 \le e < 701$, so overall there are 1402 modifications of each ciphertext. One could try to improve this—for example, just 701 modifications would identify about 1/3 of the coefficients of the relevant secret and limit the other 2/3

[16] Officially, NTRU-HRSS has three software releases and a development repository. One software release, via PQClean, was eliminated by PQClean in July 2022 [52] since NTRU is "no longer under consideration by NIST", even though, as noted above, [1] says "NIST may consider selecting NTRU instead of Kyber". The second software release, via BoringSSL, is of Google's `ntruhrss` variant, which is "not compatible" with the NTRU-HRSS specification. The third software release, via the SUPERCOP [15] benchmarking framework, is what `attackntrw` uses.

[17] Faults could also flip other bits of the secret key, or—in a broader model—bits of code, intermediate bits in computations, etc. This paper is analyzing the impact of faults in r; again, this should not be interpreted as making security claims regarding arbitrary fault attacks.

to just two values, presumably enough information to make a lattice attack feasible—but attackntrw is already very fast with 1402 modifications.

The point of these modifications is that, as noted in Section 3.6, d_j has the form $(x-1)T_j$ where T_j has coefficients in $\{-1, 0, 1\}$, and the modified ciphertext $B_j \pm 2(x-1)x^e = b_jG + d_j \pm 2(x-1)x^e$ will decrypt successfully[18] when $T_j \pm 2x^e$ has coefficients in $\{-1, 0, 1\}$, i.e., when the coefficient of x^e in T_j is ∓ 1, whereas it cannot be expected to decrypt successfully otherwise.

Without the fault, the pattern of decryption failures would be hidden by implicit rejection. However, with the fault, a decryption failure is immediately visible as a ciphertext producing a different session key before and after the fault: it would be astonishing if changing a bit in r produced a hash collision! Eve sees these mismatches, reconstructs T_j and thus d_j, and follows the relevant steps in the decapsulation algorithm to reconstruct b_j and the corresponding session key, completely breaking ntruhrss in this attack model.

To recap: ntruhrss relies critically on implicit rejection for (conjecturally) achieving IND-CCA2, but implicit rejection is fragile, losing security when a natural fault occurs.

4.3. How plaintext confirmation stops analogous mceliece and sntrup attacks. A valid mceliece ciphertext has the form $B = bG + d$ where (b, d) has a specific Hamming weight. An analogous chosen-ciphertext attack replaces B with $B + \beta G + \delta$, where (β, δ) is chosen by the attacker to have a good chance of having the right Hamming weight of $(b + \beta, d + \delta)$, as in Section 3.4. The attacker again detects whether implicit rejection has occurred by checking whether a session key is the same before and after a fault.

Similarly, a valid sntrup ciphertext has the form $bG + d$ where (b, d) has coefficients in $\{-1, 0, 1\}$ and b has a specific Hamming weight. An analogous chosen-ciphertext attack replaces B with $B + \beta G + \delta$ where (β, δ) are chosen by the attacker to have a good chance of still having coefficients in $\{-1, 0, 1\}$ in $(b + \beta, d + \delta)$ and the right Hamming weight for $b + \beta$; e.g., take $\beta = 0$ and set exactly one coefficient in δ to 1 to detect whether that coefficient of d is 1.

However, for both mceliece and sntrup, the ciphertext also includes plaintext confirmation, another hash of (b, d). As in Section 3.10, the attacker has no way to replace this with a hash of $(b + \beta, d + \delta)$. So *all* of the modified ciphertexts are (implicitly) rejected, eliminating the information that the attack needs.

For sntrup, there is an independent reason that the attack does not work as stated: see Section 3.13. However, there could be workarounds for the attacker. Plaintext confirmation makes much more obvious that the attack fails.

4.4. How proofs led ntruhrss to remove plaintext confirmation. The original version of ntruhrss in 2017 included plaintext confirmation, as did the ntruhrss submission to round 1 of the NIST competition: see [**45**, Algorithm 6, "e_2"] and [**46**, Section 1.10.4, "*qrom_hash*"]. However, the ntruhrss submission

[18] Exception: The multi-target IND-CCA2 attack model will also prevent successful decryption if a modified ciphertext happens to collide with another legitimate ciphertext. However, such collisions are so rare that they can safely be ignored.

to round 2 of the NIST competition in 2019 removed plaintext confirmation. It is interesting to look at why.

The reason for original ntruhrss and round-1 ntruhrss to include plaintext confirmation was not that *plaintext confirmation interferes with attacks*, but rather that *plaintext confirmation seemed necessary for certain types of proofs.* This distinction became important later.

Saito, Xagawa, and Yamakawa [71] proposed a modification of round-1 ntruhrss, writing in [71, Section 1.2] that "the obtained KEM is CCA secure in the QROM" under a specific assumption. The modification was designed to be as simple as possible to support the underlying QROM proof; the proof relied on implicit rejection but not on plaintext confirmation; consequently, the modification did not include plaintext confirmation.

The round-2 ntruhrss submission document [31, page 24] said that the KEM from [71] "has a tight security reduction in the ROM and avoids the plaintext-confirmation hash", along with having "a tight reduction in the QROM". The round-2 ntruhrss KEM is the KEM from [71] plus some further changes that are not relevant here. For comparison, previous versions of ntruhrss had appealed to the QROM proofs from [76], which assumed plaintext confirmation.

To summarize: Why did ntruhrss end up deciding that it was not useful to spend ciphertext space on plaintext confirmation? Answer: because plaintext confirmation turned out to be unnecessary for various types of proofs. But this paper shows that even a one-time single-bit fault is enough to break the proofs!

The practice of eliminating any cryptosystem features not needed for proofs is common in cryptography—but not universal. The possibility of plaintext confirmation stopping attacks not stopped by implicit rejection was already noted in [16, Section 17]: implicit rejection and plaintext confirmation "target different aspects of attacks", so it is "difficult to justify a recommendation against the dual-defense construction". More broadly, Koblitz wrote the following in [54, page 977]:

> Anyone working in cryptography should think very carefully before dropping a validation step that had been put in to prevent security problems. Certainly someone with Krawczyk's experience and expertise would never have made such a blunder if he hadn't been over-confident because of his "proof" of security.

The "proof" mentioned here in [54] was erroneous, but the same danger appears when a correct proof is in a model too narrow to capture real-world attacks.

4.5. Countermeasures for NTRU-HRSS. Any algorithm computing the specified ntruhrss decapsulation function will be vulnerable to the same attack. There is nothing in the secret-key format that the algorithm can use to detect that r has had a fault: r is simply 256 bits of randomness generated independently of the rest of the secret key.[19] The fault converts a valid secret key into another valid secret key.

Consequently, to stop this attack, implementors have to use a cryptosystem that isn't the currently specified `ntruhrss` cryptosystem. Perhaps the simplest approach is to switch to another secret-key format that makes bit flips detectable or even correctable, such as the generic use of Hamming codes mentioned in Section 4.1.

Implementors can also replace the specified decapsulation function with a more complicated *stateful* function that tries to detect attack patterns and to limit the exposure of each ciphertext. One approach is to maintain a database of previously seen values of d and reject nearby values, and similarly for b; but this could be a serious performance problem if "nearby" is too generous, and could allow attacks if "nearby" is too strict. An alternative is to maintain a database of ciphertexts and reject any repeated ciphertexts (modulo any "benign malleability" allowed by the cryptosystem), if this is suitable for the application. See [**42**, Section 2] for further stateful approaches. All of these approaches complicate the data flow and raise denial-of-service questions.

More options are available for implementors willing to break ciphertext interoperability with `ntruhrss`. Plaintext confirmation is an obvious choice; see Section 4.3. Limiting the ciphertext space or the plaintext space could help, but this needs analysis. Hashing the ciphertext does not help: the attack detects failing ciphertexts by seeing that a fault leads to different results for the same ciphertext.

4.6. Whose responsibility is error correction? Let's assume that there's an objective of changing the secret-key format, specifically encoding the secret key using a distance-4 extended Hamming code. This fixes natural bit flips anywhere in the secret key, not just in r, so it is attractive whether or not there is plaintext confirmation.

There's still a question of *who* should encode the secret key. Should the `ntruhrss` specification be updated to specify an encoded secret-key format? Or should applications encode secret keys, and much more data, to protect all of that data against bit flips? Or should the operating system build error correction into paging mechanisms, and continually sweep through pages to check for errors? Or should the hardware apply error correction to all data stored in DRAM?

The attack relies on all of these layers failing to act. Note that the fact that there are multiple layers that can act gives each layer an excuse not to act, especially when nobody is responsible for the security of the system as a whole.

One could respond that any layer that *can* take action should do so: the `ntruhrss` designers can specify error correction, so they should; applications can correct errors, so they should; the operating system can correct errors, so it should; and the hardware can correct errors, so it should. These layers can

[19] For comparison, the specified `mceliece` secret-key format already includes a 256-bit seed that can be double-checked against the rest of the secret key. This seed was specified to allow compression, but implementors can reuse it for double-checks of whether various faults have occurred.

share specifications, and to some extent implementations, of the error-correction mechanisms. But this nevertheless means added complications at each layer. Surely a simpler, more easily reviewed system can address the problem at hand, the same way that twist-security and x-coordinates address the ECDH security problem mentioned in Section 1.1 without the complications of implementations having to check point validity.

SECDED ECC DRAM handles DRAM bit flips in a way that's measurable and seems robust. Unfortunately, for many years it seemed that computer manufacturers were using the minor costs of SECDED ECC DRAM for market segmentation, in much the same way that 19th-century railroad companies installed a roof on *some* train cars for market segmentation; see generally [**63**, Section 3]. Perhaps DDR5 "on-die ECC"—which at least tries to catch DRAM errors, although it does not protect data in transit to the CPU—will eventually put an end to the non-ECC era, but non-ECC equipment will continue to be in use for many years.

It is clear that many options will require software for error correction. As another supplement to this paper, I've released a `libsecded` software library [**13**] that encodes arrays in RAM using a distance-4 Hamming code. However, this paper does not draw conclusions regarding the optimal way forward.

References

[1] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, Yi-Kai Liu, *Status report on the third round of the NIST Post-Quantum Cryptography Standardization Process* (2022). NISTIR 8413. URL: `https://csrc.nist.gov/publications/detail/nistir/8413/final`. Citations in this document: §1.1, §3.14, §3.14, §4.2.

[2] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, Damien Stehlé, *CRYSTALS-Kyber: Algorithm specifications and supporting documentation* (2020). URL: `https://web.archive.org/web/20211007045636/https://pq-crystals.org/kyber/data/kyber-specification-round3.pdf`. Citations in this document: §3.14.

[3] Mihir Bellare (editor), *Advances in cryptology—CRYPTO 2000, 20th annual international cryptology conference, Santa Barbara, California, USA, August 20–24, 2000, proceedings*, Lecture Notes in Computer Science, 1880, Springer, 2000. ISBN 3-540-67907-3. See [48].

[4] Mihir Bellare, Hannah Davis, Felix Günther, *Separate your domains: NIST PQC KEMs, oracle cloning and read-only indifferentiability*, in Eurocrypt 2020 [**28**] (2020), 3–32. URL: `https://eprint.iacr.org/2020/241`. Citations in this document: §3.10.

[5] Mihir Bellare, Dennis Hofheinz, Eike Kiltz, *Subtleties in the definition of IND-CCA: when and how should challenge decryption be disallowed?*, Journal of Cryptology **28** (2015), 29–48. URL: `https://eprint.iacr.org/2009/418`. Citations in this document: §3.4.

[6] Daniel J. Bernstein, *Re: Current consensus on ECC* (2001). URL: https://groups.google.com/g/sci.crypt/c/mu_paShEU3w/m/m491pYxHbtAJ. Citations in this document: §1.1.

[7] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in PKC 2006 [**79**] (2006), 207–228. URL: https://cr.yp.to/papers.html#curve25519. Citations in this document: §1.1.

[8] Daniel J. Bernstein, *A subfield-logarithm attack against ideal lattices* (2014). URL: https://blog.cr.yp.to/20140213-ideal.html. Citations in this document: §3.3.

[9] Daniel J. Bernstein, *How to design an elliptic-curve signature system* (2014). URL: https://blog.cr.yp.to/20140323-ecdsa.html. Citations in this document: §2.4.

[10] Daniel J. Bernstein, *Comparing proofs of security for lattice-based encryption* (2019). Second PQC Standardization Conference. URL: https://cr.yp.to/papers.html#latticeproofs. Citations in this document: §3.2.

[11] Daniel J. Bernstein, *On the looseness of FO derandomization* (2021). URL: https://cr.yp.to/papers.html#footloose. Citations in this document: §3.14.

[12] Daniel J. Bernstein, *Understanding binary-Goppa decoding* (2022). URL: https://cr.yp.to/papers.html#goppadecoding. Citations in this document: §3.8, §3.8, §3.8.

[13] Daniel J. Bernstein, `libsecded` *(software package)* (2022). URL: https://pqsrc.cr.yp.to/downloads.html. Citations in this document: §4.6.

[14] Daniel J. Bernstein, `attackntrw` *(software package)* (2022). URL: https://pqsrc.cr.yp.to/downloads.html. Citations in this document: §4.

[15] Daniel J. Bernstein, Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems* (2022). Accessed 25 August 2022. URL: https://bench.cr.yp.to. Citations in this document: §4.2.

[16] Daniel J. Bernstein, Edoardo Persichetti, *Towards KEM unification* (2018). URL: https://cr.yp.to/papers.html#tightkem. Citations in this document: §3.12, §4.4.

[17] Eli Biham (editor), *Fast software encryption, 4th international workshop, FSE '97, Haifa, Israel, January 20–22, 1997, proceedings*, Lecture Notes in Computer Science, 1267, Springer, 1997. ISBN 3-540-63247-6. See [70].

[18] Eli Biham, Lior Neumann, *Breaking the Bluetooth pairing—the fixed coordinate invalid curve attack*, in SAC 2019 [**65**] (2019), 250–273. URL: https://www.cs.technion.ac.il/~biham/BT/. Citations in this document: §1.1.

[19] Mario Blaum, Patrick G. Farrell, Henk C. A. van Tilborg (editors), *Information, coding and mathematics*, Kluwer International Series in Engineering and Computer Science, 687, Kluwer, 2002. MR 2005a:94003. See [78].

[20] Daniel Bleichenbacher, *Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1*, in Crypto 1998 [**55**] (1998), 1–12. URL: https://archiv.infsec.ethz.ch/education/fs08/secsem/Bleichenbacher98.pdf. Citations in this document: §1.

[21] Hanno Böck, Juraj Somorovsky, Craig Young, *Return of Bleichenbacher's oracle threat (ROBOT)*, in [**34**] (2018), 817–849. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bock. Citations in this document: §1.

[22] Dan Boneh (editor), *Advances in cryptology—CRYPTO 2003, 23rd annual international cryptology conference, Santa Barbara, California, USA, August 17–21, 2003, proceedings*, Lecture Notes in Computer Science, 2729, Springer, 2003. ISBN 3-540-40674-3. See [44].

[23] Dan Boneh, Richard A. DeMillo, Richard J. Lipton, *On the importance of checking cryptographic protocols for faults (extended abstract)*, in Eurocrypt 1997 [**36**] (1997), 37–51; see also newer version [**24**]. URL: `https://link.springer.com/content/pdf/10.1007/3-540-69053-0_4.pdf`. Citations in this document: §2.3, §2.3, §2.3, §2.3, §2.4, §2.4, §2.5, §2.5, §2.5.

[24] Dan Boneh, Richard A. DeMillo, Richard J. Lipton, *On the importance of eliminating errors in cryptographic computations*, Journal of Cryptology **14** (2001), 101–119; see also older version [**23**]. URL: `https://crypto.stanford.edu/~dabo/abstracts/faults.html`.

[25] Joe P. Buhler (editor), *Algorithmic number theory, third international symposium, ANTS-III, Portland, Oregon, USA, June 21–25, 1998, proceedings*, Lecture Notes in Computer Science, 1423, Springer, 1998. ISBN 3-540-64657-4. See [40].

[26] Kevin Butler, Kurt Thomas (editors), *31st USENIX Security Symposium*, USENIX Association, 2022. ISBN 978-1-939133-31-1. See [75].

[27] L. Jean Camp, Stephen Lewis (editors), *Economics of information security*, Advances in Information Security, 12, Springer, 2004. ISBN 1-4020-8089-1. See [63].

[28] Anne Canteaut, Yuval Ishai (editors), *Advances in cryptology—EUROCRYPT 2020—39th annual international conference on the theory and applications of cryptographic techniques, Zagreb, Croatia, May 10–14, 2020, proceedings, part II*, Lecture Notes in Computer Science, 12106, Springer, 2020. ISBN 978-3-030-45723-5. See [4].

[29] Anne Canteaut, 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*, Lecture Notes in Computer Science, 12697, Springer, 2021. ISBN 978-3-030-77885-9. See [30].

[30] Pierre-Louis Cayrel, Brice Colombier, Vlad-Florin Dragoi, Alexandre Menu, Lilian Bossuet, *Message-recovery laser fault injection attack on the Classic McEliece cryptosystem*, in [**29**] (2021), 438–467. URL: `https://eprint.iacr.org/2020/900`. Citations in this document: §2.2.

[31] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, *NTRU: algorithm specifications and supporting documentation* (2019). URL: `https://ntru.org/f/ntru-20190330.pdf`. Citations in this document: §4.4.

[32] John R. Douceur, Albert G. Greenberg, Thomas Bonald, Jason Nieh (editors), *Proceedings of the eleventh international joint conference on measurement and modeling of computer systems, SIGMETRICS/Performance 2009, Seattle, WA, USA, June 15–19, 2009*, ACM, 2009. ISBN 978-1-60558-511-6. See [72].

[33] Orr Dunkelman, Stefan Dziembowski (editors), *Advances in cryptology—EUROCRYPT 2022—41st annual international conference on the theory and applications of cryptographic techniques, Trondheim, Norway, May 30–June 3, 2022, proceedings, part III*, Lecture Notes in Computer Science, 13277, Springer, 2022. ISBN 978-3-031-07081-5. See [47].

[34] William Enck, Adrienne Porter Felt (editors), *27th USENIX security symposium, USENIX Security 2018, Baltimore, MD, USA, August 15–17, 2018*, USENIX Association, 2018. URL: `https://www.usenix.org/conference/usenixsecurity18`. See [21].

[35] Wieland Fischer, Naofumi Homma (editors), *Cryptographic hardware and embedded systems—CHES 2017—19th international conference, Taipei, Taiwan,*

September 25–28, 2017, proceedings, Lecture Notes in Computer Science, 10529, Springer, 2017. ISBN 978-3-319-66786-7. See [45].

[36] Walter Fumy (editor), *Advances in cryptology—EUROCRYPT '97, international conference on the theory and application of cryptographic techniques, Konstanz, Germany, May 11–15, 1997, proceeding*, Lecture Notes in Computer Science, 1233, Springer, 1997. ISBN 3-540-62975-0. See [23].

[37] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, Edward W. Felten, *Lest we remember: cold boot attacks on encryption keys*, in USENIX Security 2008 [**64**] (2008), 45–60. URL: https://www.usenix.org/events/sec08/tech/full_papers/halderman/halderman.pdf. Citations in this document: §2.6, §2.6, §2.6, §2.6.

[38] Chris Hall, Ian Goldberg, Bruce Schneier, *Reaction attacks against several public-key cryptosystems*, in ICICS 1999 [**77**] (1999), 2–12. URL: https://cypherpunks.ca/~iang/pubs/paper-reaction-attacks.pdf. Citations in this document: §3.6, §3.6, §3.6.

[39] Martin Hirt, Adam D. Smith (editors), *Theory of cryptography—14th international conference, TCC 2016-B, Beijing, China, October 31–November 3, 2016, proceedings, part II*, Lecture Notes in Computer Science, 9986, 2016. ISBN 978-3-662-53643-8. See [76].

[40] Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman, *NTRU: a ring-based public key cryptosystem*, in ANTS III [**25**] (1998), 267–288. URL: https://ntru.org/f/hps98.pdf. Citations in this document: §3.3, §3.3.

[41] Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman, *NTRU: a new high speed public key cryptosystem* (2016). Circulated at Crypto 1996, put online in 2016. URL: https://ntru.org/f/hps96.pdf. Citations in this document: §3.3.

[42] Jeffrey Hoffstein, Joseph H. Silverman, *Reaction attacks against the NTRU public key cryptosystem* (2000). URL: https://ntru.org/f/tr/tr015v2.pdf. Citations in this document: §3.7, §4.5.

[43] Dennis Hofheinz, Kathrin Hövelmanns, Eike Kiltz, *A modular analysis of the Fujisaki-Okamoto transformation*, in TCC 2017-1 [**50**] (2017), 341–371. URL: https://eprint.iacr.org/2017/604. Citations in this document: §3.11, §3.12.

[44] Nick Howgrave-Graham, Phong Q. Nguyen, David Pointcheval, John Proos, Joseph H. Silverman, Ari Singer, William Whyte, *The impact of decryption failures on the security of NTRU encryption*, in Crypto 2003 [**22**] (2003), 226–246. URL: https://iacr.org/archive/crypto2003/27290225/27290225.pdf. Citations in this document: §3.7.

[45] Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, *High-speed key encapsulation from NTRU*, in [**35**] (2017), 232–252. URL: https://eprint.iacr.org/2017/667. Citations in this document: §4.4.

[46] Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, *NTRU-HRSS-KEM: algorithm specifications and supporting documentation* (2017). URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/Round-1-Submissions. Citations in this document: §4.4.

[47] Loïs Huguenin-Dumittan, Serge Vaudenay, *On IND-qCCA security in the ROM and its applications: CPA security is sufficient for TLS 1.3*, in Eurocrypt 2022 [**33**] (2022), 613–642. URL: https://eprint.iacr.org/2021/844. Citations in this document: §1.

[48] Éliane Jaulmes, Antoine Joux, *A chosen-ciphertext attack against NTRU*, in Crypto 2000 [**3**] (2000), 20–35. URL: https://www.iacr.org/archive/crypto2000/18800021/18800021.pdf. Citations in this document: §3.7.

[49] Simon Josefsson, Ilari Liusvaara, *Edwards-curve digital signature algorithm (EdDSA)* (2017). URL: https://datatracker.ietf.org/doc/rfc8032/. Citations in this document: §2.4.

[50] Yael Kalai, Leonid Reyzin (editors), *Theory of cryptography—15th international conference, TCC 2017, Baltimore, MD, USA, November 12–15, 2017, proceedings, part I*, Lecture Notes in Computer Science, 10677, Springer, 2017. ISBN 978-3-319-70499-9. See [43].

[51] Burt Kaliski, *PKCS #1: RSA encryption version 1.5* (1998). URL: https://datatracker.ietf.org/doc/html/rfc2313. Citations in this document: §2.4.

[52] Matthias Kannwischer, *Remove schemes that are no longer under consideration by NIST* (2022). URL: https://github.com/PQClean/PQClean/commit/b0fc0c098c7930e04931071a46cc7d520a1f8d1b. Citations in this document: §4.2.

[53] Jonathan Katz, Yehuda Lindell, *Introduction to modern cryptography: principles and protocols*, Chapman & Hall/CRC, 2007. ISBN 978-1-58488-551-1. URL: https://www.cs.umd.edu/~jkatz/imc.html. Citations in this document: §1.1.

[54] Neal Koblitz, *The uneasy relationship between mathematics and cryptography*, Notices of the American Mathematical Society **54** (2007), 972–979. URL: https://www.ams.org/notices/200708/tx070800972p.pdf. Citations in this document: §4.4, §4.4.

[55] Hugo Krawczyk (editor), *Advances in cryptology—CRYPTO '98, 18th annual international cryptology conference, Santa Barbara, California, USA, August 23–27, 1998, proceedings*, Lecture Notes in Computer Science, 1462, Springer, 1998. ISBN 3-540-64892-5. See [20].

[56] Adam Langley, *CECPQ2* (2018). URL: https://www.imperialviolet.org/2018/12/12/cecpq2.html. Citations in this document: §1.

[57] Arjen K. Lenstra, *Memo on RSA signature generation in the presence of faults* (1996). URL: https://infoscience.epfl.ch/record/164524. Citations in this document: §2.3, §2.5.

[58] Vadim Lyubashevsky, *OFFICIAL COMMENT: CRYSTALS-DILITHIUM* (2018). URL: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/aWxC2ynJDLE/m/YOsMJ2ewAAAJ. Citations in this document: §1.1.

[59] Robert J. McEliece, *A public-key cryptosystem based on algebraic coding theory* (1978), 114–116. JPL DSN Progress Report. URL: https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF. Citations in this document: §3.3, §3.3.

[60] Alfred Menezes, *Evaluation of security level of cryptography: RSA signature schemes (PKCS#1 v1.5, ANSI X9.31, ISO 9796)* (2002). URL: https://www.cryptrec.go.jp/exreport/cryptrec-ex-1014-2002.pdf. Citations in this document: §1.

[61] National Institute of Standards and Technology, *Submission requirements and evaluation criteria for the post-quantum cryptography standardization process* (2016). URL: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf. Citations in this document: §1.

[62] Jesper Buus Nielsen, Vincent Rijmen (editors), *Advances in cryptology—EUROCRYPT 2018—37th annual international conference on the theory and applications of cryptographic techniques, Tel Aviv, Israel, April*

29–May 3, 2018 proceedings, part III, Lecture Notes in Computer Science, 10822, Springer, 2018. ISBN 978-3-319-78371-0. See [71].

[63] Andrew M. Odlyzko, *Privacy, economics, and price discrimination on the internet*, in [**27**] (2004), 187–211. URL: `http://www.dtc.umn.edu/~odlyzko/doc/privacy.economics.pdf`. Citations in this document: §4.6.

[64] Paul C. van Oorschot (editor), *Proceedings of the 17th USENIX security symposium, July 28–August 1, 2008, San Jose, CA, USA*, USENIX Association, 2008. ISBN 978-1-931971-60-7. See [37].

[65] Kenneth G. Paterson, Douglas Stebila (editors), *Selected areas in cryptography—SAC 2019—26th international conference, Waterloo, ON, Canada, August 12–16, 2019, revised selected papers*, Lecture Notes in Computer Science, 11959, Springer, 2020. ISBN 978-3-030-38470-8. See [18].

[66] Trevor Perrin, *The XEdDSA and VXEdDSA signature schemes* (2016). URL: `https://www.signal.org/docs/specifications/xeddsa/xeddsa.pdf`. Citations in this document: §2.4.

[67] Edoardo Persichetti, *Improving the efficiency of code-based cryptography*, Ph.D. thesis, 2012. URL: `https://persichetti.webs.com/Thesis%20Final.pdf`. Citations in this document: §3.11.

[68] Bart Preneel (editor), *Advances in cryptology—EUROCRYPT 2000, international conference on the theory and application of cryptographic techniques, Bruges, Belgium, May 14–18, 2000, proceeding*, Lecture Notes in Computer Science, 1807, Springer, 2000. ISBN 3-540-67517-5. See [74].

[69] Prasanna Ravi, Martianus Frederic Ezerman, Shivam Bhasin, Anupam Chattopadhyay, Sujoy Sinha Roy, *Will you cross the threshold for me? Generic side-channel assisted chosen-ciphertext attacks on NTRU-based KEMs*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2022.1** (2022), 722–761. URL: `https://eprint.iacr.org/2021/718`. Citations in this document: §3.13.

[70] Ronald L. Rivest, *All-or-nothing encryption and the package transform*, in FSE 1997 [**17**] (1997), 210–218. URL: `https://people.csail.mit.edu/rivest/pubs/Riv97d.pdf`. Citations in this document: §3.5.

[71] Tsunekazu Saito, Keita Xagawa, Takashi Yamakawa, *Tightly-secure key-encapsulation mechanism in the quantum random oracle model*, in Eurocrypt 2018 [**62**] (2018), 520–551. URL: `https://eprint.iacr.org/2017/1005`. Citations in this document: §4.4, §4.4, §4.4, §4.4.

[72] Bianca Schroeder, Eduardo Pinheiro, Wolf-Dietrich Weber, *DRAM errors in the wild: a large-scale field study*, in [**32**] (2009), 193–204. URL: `https://www.cs.toronto.edu/~bianca/papers/sigmetrics09.pdf`. Citations in this document: §1.2, §1.2, §1.2.

[73] Mark Seaborn, Thomas Dullien, *Exploiting the DRAM rowhammer bug to gain kernel privileges* (2015). URL: `https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`. Citations in this document: §2.2.

[74] Victor Shoup, *Using hash functions as a hedge against chosen ciphertext attack*, in Eurocrypt 2000 [**68**] (2000), 275–288. URL: `https://www.iacr.org/archive/eurocrypt2000/1807/18070279-new.pdf`. Citations in this document: §3.5.

[75] George Arnold Sullivan, Jackson Sippe, Nadia Heninger, Eric Wustrow, *Open to a fault: On the passive compromise of TLS keys via transient errors*, in USENIX Security 2022 [**26**] (2022), 233–250. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/sullivan`. Citations in this document: §2.3, §2.3, §2.3, §2.3.

[76] Ehsan Ebrahimi Targhi, Dominique Unruh, *Post-quantum security of the Fujisaki-Okamoto and OAEP transforms*, in [**39**] (2016), 192–216. URL: `https://eprint.iacr.org/2015/1210`. Citations in this document: §4.4.

[77] Vijay Varadharajan, Yi Mu (editors), *Information and communication security, second international conference, ICICS'99, Sydney, Australia, November 9–11, 1999, proceedings*, Springer, 1999. ISBN 3-540-66682-6. See [38].

[78] Eric R. Verheul, Jeroen M. Doumen, Henk C. A. van Tilborg, *Sloppy Alice attacks! Adaptive chosen ciphertext attacks on the McEliece public-key cryptosystem*, in [**19**] (2002), 99–119. MR 2005b:94041. URL: `https://research.utwente.nl/en/publications/sloppy-alice-attacks-adaptive-chosen-ciphertext-attacks-on-the-mc`. Citations in this document: §3.6, §3.6, §3.6.

[79] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, Tal Malkin (editors), *Public key cryptography—9th international conference on theory and practice in public-key cryptography, New York, NY, USA, April 24–26, 2006, proceedings*, Lecture Notes in Computer Science, 3958, Springer, 2006. ISBN 978-3-540-33851-2. See [7].

[80] Meilin Zhang, Vladimir M. Stojanovic, Paul Ampadu, *Reliable ultra-low-voltage cache design for many-core systems*, IEEE Transactions on Circuits and Systems II: Express Briefs **59** (2012), 858–862. URL: `https://www.rle.mit.edu/isg/documents/Zhang_TCASII12.pdf`. Citations in this document: §1.2.