Cryptanalysis of the Wave Signature Scheme

Paulo S. L. M. Barreto[1] and Edoardo Persichetti[2]

[1] School of Engineering and Technology, University of Washington Tacoma
pbarreto@uw.edu

[2] Department of Mathematical Sciences, Florida Atlantic University
epersichetti@fau.edu

Abstract. In this paper, we cryptanalyze the signature scheme Wave, which has recently appeared as a preprint. First, we show that there is a severe information leakage occurring from honestly-generated signatures. Then, we illustrate how to exploit this leakage to retrieve an alternative private key, which enables efficiently forging signatures for arbitrary messages. Our attack manages to break the proposed 128-bit secure Wave parameters in just over a minute, most of which is actually spent collecting genuine signatures. Finally, we explain how our attack applies to generalized versions of the scheme which could potentially be achieved using generalized admissible $(U, U + V)$ codes and larger field characteristics.

1 Introduction

Digital signatures schemes are, without a doubt, one of the most important cryptographic protocols in use today, addressing the issue of authentication of data. Digital signatures share many similarities with confidentiality operations like encryption, key agreement, and key encapsulation; in some cases (e.g. RSA), this means that it is possible to design a signature scheme very naturally.

However, this is not always true for all families of cryptosystems, with one of the best examples being code-based cryptography. Despite code-based schemes constituting one of the leading families of proposed quantum-resistant (*aka* post-quantum) cryptosystems, designing efficient code-based signature schemes is in fact a major challenge and, to date, still an open problem. The reason for this is that the traditional hash-and-sign approach used for instance by RSA and other signatures is severely hindered by the fact that the subset of decodable syndromes of a linear code is typically only an exceedingly minute part of the total vector space, and so picking a vector at random (via hashing) most likely will not produce a decodable syndrome. This forces the signing algorithm to be repeated a prohibitively large number of times, creating an obviously impractical process. The authors of CFS [1], the first proposed signature scheme following this approach, propose particular choices of parameters to alleviate this issue, but the number of attempts required on average is still quite high (e.g. around 9! for an estimated security level of 2^{80}) and the resulting signing times are several orders of magnitude slower than schemes based on other primitives. Moreover, "CFS-friendly" codes have a very high rate, and this can lead to potential additional issues such as distinguishers [5].

Alternatively, one could think of obtaining a signature scheme using the Fiat-Shamir transform and an identification scheme: this approach works very well for lattices, for instance [6]. However, the Hamming metric that comes with linear codes (instead of the Euclidean metric used for lattices) has proved to be too weak to provide security [7], and therefore the only plausible method to obtain code-based zero-knowledge is to use multiple commitments, as in Stern's scheme [8]. The problem with this type of schemes, though, is that multiple commitments allow the attacker to cheat and be successful with non-trivial probability (2/3 in Stern's original proposal), and therefore the identification scheme needs to be repeated multiple times to guarantee acceptance. This results in a very large signature size. Despite attempts to reduce this cheating probability [9], it seems implausible to obtain a truly efficient scheme in this way.

The recent Wave scheme [3] would be a nice development in this scenario. The scheme features an arguably elegant design and comes with a formal proof of security in the sense of existential unforgeability under chosen-message attacks (EUF-CMA). If that scheme withstood cryptanalysis, it would constitute a real and most welcome breakthrough.

Our Contribution: In this work, we show that, unfortunately, the Wave scheme leaks a considerable amount of private information in genuinely generated signatures. The leaked information is enough to recover an equivalent private key from the public key and a number of collected valid signatures, enabling an attacker to forge signatures for arbitrary messages without knowing the actual private key. The number of legitimate signatures the attacker needs to gather is fairly small (around 600 for the proposed 128-bit secure Wave parameters). The equivalent key recovery runs very fast in practice (a few seconds). The most time-consuming stage by far is the generation of the collected legitimate signatures (about one minute).

This is unusual for signature schemes equipped with EUF-CMA proofs, but not unprecedented. The Courtois-Finiasz-Sendrier (CFS) scheme [1] was similarly proven existentially unforgeable by Dallot [2] under the assumption that the underlying code adopted for the scheme is indistinguishable from a uniformly random linear error-correcting code. As it turned out, the only codes known to be suitable for CFS are very high-rate Goppa codes, which have since been shown to be distinguishable from random [4]. In other words, there is nothing wrong with the proof *per se*; rather, the hardness assumptions themselves fail to hold.

The same unfortunate circumstance is found here, with a crucial difference: an attacker can actually forge Wave signatures for arbitrary messages, rather than merely a single signature for some valid but contrived, meaningless message, while CFS signatures are, to the best of our knowledge, still empirically secure (it is just that the security proof does not hold for the chosen codes). This behavior is not only predicted theoretically by our attack, it is supported by empirical evidence in the form of a Magma implementation that shows the attack to be entirely practical.

The remainder of this paper is organized as follows. We start by recapitulating theoretical notions in Section 2. We summarize the Wave digital signature scheme in Section 3. In Section 4 we show how the structure of Wave parity-check matrices unavoidably leaks structural information about the private key in each generated signature. In Section 5 we turn the leaked information into a full-fledged attack that recovers an equivalent private key and enables forging signatures. In Section 6 we present the results obtained via a simple Magma script. Section 7 discussed generalizations of our attack to other potential families of underlying codes mentioned in the Wave specification. We conclude in Section 8.
For the sake of completeness, we have included our Magma script, in Appendix A, to better illustrate our attack against the proposed 128-bit security parameters.

**** Update:* Since the first version of this report was ready, the authors of Wave have contacted us to point out that, in our description of Wave and in the script we include in the Appendix, we omit the rejection sampling they adopt to make sure the distribution of coefficients in the signatures is weakly uniform (that is, that signatures are statistically close to uniform).

In response, we point out that our attack does not depend on, and in fact is not related at all, to that distribution. As we describe in Section 4, our distinguisher is not based on biases on the overall distribution of signature coefficient. Rather, it can be viewed as a *differential* distinguisher: it keeps track of individual differences between certain components, not their distributions. This differential bias is not removed when some candidate signatures are rejected during signing: it cannot be removed, since doing so would necessarily mean violating the Hamming weight constraint on the signatures.

Indeed, the script in the Appendix can be readily adapted to simply use signatures generated elsewhere. Only the final forgery would have to be updated accordingly, namely, a few candidate signatures would have to be computed from the recovered equivalent and tested for whatever rejection sampling criterion is adopted. The same expected amount of rejections are expected for forged signatures as they are for genuine ones (between 25% and 33% according to the authors of Wave).

Summarizing: out attack still holds against a fully detailed implementation of Wave; it does not depend on the total counts of signature coefficient values being uniform, and is not thwarted by making them so.

2 Preliminaries

Notation: We denote with $\mathbf{0}$ the all-zero matrix and with $\mathbf{I}$ the identity matrix.

Let p be a prime number and let n an k be positive integers with $k < n$. The *Hamming weight* of a vector $x \in \mathbb{F}_p^n$ is defined as the number $\mathsf{wt}(x)$ of its nonzero components. An (n, r)*-linear code* $\mathcal{C}$ of length n, dimension k and co-dimension $r = n - k$ is a k-dimensional vector subspace of $\mathbb{F}_p^n$. The code is called *ternary* in the case $p = 3$ (and *p-ary* in general).

Every linear code is spanned by the rows of some matrix $G \in \mathbb{F}_p^{k \times n}$, called a *generator matrix* of $\mathcal{C}$. Since bases of vector spaces are not unique, every code admits multiple generator matrices, each corresponding to a particular choice of basis. Equivalently, a linear code can be described as the kernel of a matrix $H \in \mathbb{F}_p^{r \times n}$, called *parity-check matrix*, i.e. $\mathcal{C} = \{c \mid cH^T = \mathbf{0}\}$. The *codeword* $c \in \mathcal{C}$ corresponding to a vector $m \in \mathbb{F}_q^k$ is given by $c = mG$. The *syndrome* $s \in \mathbb{F}_p^r$ of a vector $e \in \mathbb{F}_p^n$ is the vector $s := eH^T$.

A $(U, U+V)$ *code* is a linear code that admits a parity-check matrix of form

$$H := \begin{bmatrix} H_U & \mathbf{0} \\ -H_V & H_V \end{bmatrix} \in \mathbb{F}_p^{r \times n},$$

for some $H_U \in \mathbb{F}_p^{r_U \times (n/2)}$, $H_V \in \mathbb{F}_p^{r_V \times (n/2)}$, and $r_U + r_V = r$.

3 The Wave Signature Scheme

The original Wave signature scheme follows the hash-and-sign approach. This means that signatures are obtained by decoding syndromes, which are (randomized) hashes of the message to be signed, into error patterns of a certain Hamming weight. Traditionally in code-based cryptography, this Hamming weight is required to be very low (well below the Gilbert-Varshamov bound), so that the honest signer can make use of his trapdoor, i.e. the decoding algorithm associated to the code in use, while decoding is hard for an attacker. However, this is not the only instance in which decoding is hard: as the authors of Wave point out, the problem is easy if the weight grows beyond the bound, but it becomes hard again when it becomes very high.

The setting that is actually proposed in Wave is the latter (see [3, "Tweaking the Prange Decoder for Reaching Large Weights"]), where the error pattern has very high Hamming weight. The reason is that the trapdoor enables solving dense decoding instances which are seemingly much harder than sparse instances when the code characteristic p is odd. The trapdoor itself consists of the particular class of codes chosen, a generalized version of $(U, U+V)$ codes called *Generalized Admissible* $(U, U+V)$ codes. These codes can be decoded using a small variant of the $(U, U+V)$ decoder, for which the range of relative weights that are easy to decode is wider than the generic "easy" range (hence the trapdoor).

The only concrete instantiation specified by the authors of Wave is in characteristic $p = 3$ and for plain $(U, U+V)$ codes. We will simplify our exposition accordingly, although we will also discuss the general case later.

3.1 Key Pair

The Wave private key is a triple (S, H_{sk}, P), where $S \in \mathbb{F}_3^{r \times r}$ is a nonsingular matrix, $H_{sk} \in \mathbb{F}_3^{r \times n}$ is the parity-check matrix of a linear code of even length n, co-dimension r (and dimension $k = n - r$), and $P \in \mathbb{F}_3^{n \times n}$ is a permutation matrix (i.e. each row and each column have only a single component of unit value, and all other components are zero), where H_{sk} is a generator matrix of a $(U, U+V)$ code, i.e. it has the shape:

$$H_{sk} := \begin{bmatrix} H_U & \mathbf{0} \\ -H_V & H_V \end{bmatrix} \in \mathbb{F}_3^{r \times n},$$

with $H_U \in \mathbb{F}_3^{r_U \times (n/2)}$, $H_V \in \mathbb{F}_3^{r_V \times (n/2)}$, and $r_U + r_V = r$. The public key associated to the triple is $H_{pk} := SH_{sk}P$.

3.2 Signing and Verifying

The structure of the private parity-check matrix allows to decode a syndrome $s := (s_U, s_V)$ with $s_U \in \mathbb{F}_3^{r_U}$, $s_V \in \mathbb{F}_3^{r_V}$, into an error pattern $e := (e_U, e_U + e_V)$ with $e_U, e_V \in \mathbb{F}_3^{n/2}$. If we initially disregard the Hamming weight of the solution, we just need to obtain arbitrary solutions of the two linear systems $e_V H_V^T = s_V$ and $e_U H_U^T = s_U$, as one can check that $e = (e_U, e_U + e_V)$ is an arbitrary solution of $eH_{sk}^T = s$.

Solving those two linear systems is easy: choose any $n/2 - r_U$ (resp. $n/2 - r_V$) components of the desired error pattern and set them to arbitrary values, then solve the resulting square linear system for the remaining r_U (resp. r_V) components. Prange's method does essentially this, but is tailored to look for a solution of specifically low or high Hamming weight: instead of setting the chosen components to random values, set them all to zero (to obtain a low-weight solution) or else set them all to random nonzero values (to obtain a high-weight solution).

As the authors point out, one can actually do much better to obtain high-weight solutions of the target linear system $eH_{sk}^T = s$. The technique consists of choosing e_V entirely at random, and then computing e_U to satisfy certain criteria. Specifically, since we want $e_U + e_V$ to have as few zero entries as possible, and given that the code has characteristic 3, it is enough to make as many entries on e_U equal to the corresponding entries on e_V if the latter are nonzero (if $e_V[j] = \pm 1$, setting $e_U[j] \leftarrow e_V[j]$ not only ensures $e_U[j] \neq 0$, but also $e_U[j] + e_V[j] = -e_U[j] \neq 0$), or else a random nonzero value if those entries on e_V are zero (in which case $e_U[j] = \pm 1$ at random and $e_U[j] + e_V[j] = e_U[j] \neq 0$). Consequently, all but r_U entries on e_U and as many on $e_U + e_V$ can be forced to be nonzero. The remaining r_U entries on e_U are computed as solutions to the linear system $e_U H_U^T = s_U$ and hence their values are not coercible to nonzero values, nor are the values of the corresponding entries on $e_U + e_V$, for the same reason. However, since those $2r_U$ values are expected to be roughly uniformly distributed, only about 1/3 of them will turn out to be zero, and the expected weight of $(e_U, e_U + e_V)$ is thus about $n - 2e_U/3$.

3.3 Parameters

Only one set of parameters is suggested in the original description of Wave, namely, ternary codes of length $n = 5172$, dimension $k = 3908$ and target

Hamming weight $w = 4980$, with $k = k_U + k_V$, $k_U = 2299$, $k_V = 1609$ (and hence $r = r_U + r_V = 1264$, $r_U = 287$, $r_V = 977$). Here we see how the dense setting outperforms the more intuitive, sparse one: in this configuration, sparse Wave could only expect to decode a syndrome to an error pattern of weight around 747, while the dense setting decoding to $w = \lfloor n - 2r_U/3 \rfloor = 4980$ leaves only 192 zero columns.

4 Structural Leakage

We now show that, as it turns out, it is possible to accumulate statistics from observed genuine signatures and recover information on the secret column permutation P.

Indeed, consider the distribution of entry values in the Wave setting with very dense error patterns before the column permutation. We know that $n/2 - r_U$ entries of e_U have their values chosen rather than computed as solutions of a linear system. About $2/3$ out of those entries $e_U[j]$, namely, those corresponding to $e_V[j] \neq 0$, are set to be equal to $e_V[j]$, in which case $e_U[j] + e_V[j] = -e_U[j]$. The remaining $1/3$ entries, corresponding to $e_V[j] = 0$, are set to $e_U[j] = \pm 1$ uniformly at random, in which case $e_U[j] + e_V[j] = e_U[j]$. The values of the remaining (computed) r_U entries of e_U are close to uniform, as are the corresponding entries of $e_U + e_V$. If neither is zero, which happens about $4/9$ of the time, they are either equal or opposite with equal probability, namely, $2/9$ of the time each.

Overall, $(2/3)(n/2 - r_U) + (2/9)r_U = n/3 - (4/9)r_U$ entries of e_U are nonzero and opposite to its corresponding entry on $e_U + e_V$, $(1/3)(n/2 - r_U) + (2/9)r_U = n/6 - (1/9)r_U$ entries of e_U are nonzero and equal to that entry on $e_U + e_V$, and $(5/9)r_U$ entries of e_U are zero or correspond to a zero entry on $e_U + e_V$.

Now let $e := (e_U, e_U + e_V)$ before the column permutation. Then, discarding the entries where either e_U or $e_U + e_V$ are zero, we see that

$$\Pr\{e[j] = -e[j + n/2]\} \approx \frac{n/3 - (4/9)r_U}{n/2 - (5/9)r_U}$$

and

$$\Pr\{e[j] = e[j + n/2]\} \approx \frac{n/6 - (1/9)r_U}{n/2 - (5/9)r_U}.$$

Therefore, after the column permutation is applied, one still expects

$$\Pr\{e[j] = -e[h]\} \approx \frac{n/3 - (4/9)r_U}{n/2 - (5/9)r_U} \approx 2/3$$

and

$$\Pr\{e[j] = e[h]\} \approx \frac{n/6 - (1/9)r_U}{n/2 - (5/9)r_U} \approx 1/3$$

whenever two columns j and h correspond to the same original pair and their values are both nonzero. If columns j and h are nonzero but do *not* correspond to one same original pair, then $\Pr\{e[j] = e[h]\} \approx 1/2$ and $\Pr\{e[j] = -e[h]\} \approx 1/2$, since their values are independent and essentially uniform.

Thus, observing a certain amount of genuine signatures, it would be possible in principle to accumulate enough statistics to infer which pairs of columns on the public parity-check matrix are more likely to correspond to matching pairs of columns on the secret parity-check matrix. To that end, among those signatures where any target columns j and h are both nonzero, count how many times their values are equal and how many times they are opposite: if the ratio is close to 2 (or $1/2$), these columns are likely a matching pair in the unpermuted code; if the ratio is close to 1, they are likely unrelated.

One might think, at first glance, that adopting a *signed* column permutation might thwart the accumulation of statistics. However, since the permutation is fixed for all signatures (it is a feature of the key pair), the actual result would merely be swapping the unbalanced probabilities without affecting the ratio of those probabilities. In particular, if two originally matching columns j and h are permuted and one of them is sign-swapped, then

$$\Pr\{e[j] = -e[h]\} \approx \frac{n/3 - (4/9)r_U}{n/2 - (5/9)r_U}$$

and

$$\Pr\{e[j] = e[h]\} \approx \frac{n/6 - (1/9)r_U}{n/2 - (5/9)r_U}$$

when they correspond to the same original pair and their values are both nonzero: the counts are inverted, but their ratio is close to 2 (or $1/2$) rather than to 1, as would be the case for unmatching columns. For simplicity, then, we henceforth only focus on the case where P is an unsigned permutation.

4.1 Gathering Statistics

There is a simple and effective way to gather the required statistics to recover the essential code structure. Let C be the matrix whose rows consist of all t collected signatures, each being a vector from $\mathbb{F}_3$. Lift the entries of C to $\mathbb{Z}$ with zero-centered entries (i.e. view C as a matrix of integers in range $[-1..1]$ rather than $[0..2]$). Then one can distinguish between matching and unmatching columns by looking for the largest entry on each row of $C' := -C^T C$.

This works because, when two given columns i and j do not match in the private code, the product of the corresponding coefficients on any given signature is either zero (in which case it does not contribute to the value of C'_{ij}), or else is uniformly distributed between -1 and 1. Either way, C'_{ij} will be simply a sum of up to t values uniformly chosen from $\{-1, 1\}$, with an expected value of zero and a standard deviation of $\sqrt{t}$.

However, when the columns do match, C'_{ij} will be the difference between the number of opposite and equal coefficients that occur at those columns among all collected signatures. Thus the expected value of C'_{ij} in this case is not zero but around $t/3$ and again a standard deviation of around $\sqrt{t}$, with discrepancies due to the noise introduced when those columns are computed rather than chosen and neither is zero.

Except for those discrepancies, this distinguisher is likely to identify a pair of matching columns as long as the observed values between matching and unmatching columns do not coincide. Coincidence happens within α standard deviations when $0 + \alpha\sqrt{t} = t/3 - \alpha\sqrt{t}$, that is, when $\alpha = \sqrt{t}/6$. For a desired (un)likeliness of failure (which corresponds to the probability of an event at or outside α standard deviations), this yields the approximate number of signatures needed to ensure success in guessing correctly one matching column pair, namely, $t = (6\alpha)^2$. Here we use the central limit theorem to approximate the distribution of C'_{ij} with a normal distribution (which is reasonable since the signature coefficients are, to a high enough precision, independent and identically distributed).

For instance, $\alpha = 4$ corresponds to a probability of failure of $1/15787$ per column pair, or roughly $1/6$ for all pairs when $n = 5172$ (the only proposed Wave code length). To attain this, about $t = (6\alpha)^2 = 576$ signatures must be collected. In oractice more signatures are required to compensate for the aforementioned noise (but not too many more).

4.2 Recovering the Essential Key Structure

Although the above process does not reveal the exact column permutation P, it allows to group the columns of the parity-check matrix in such a way that matching pairs are exactly at a distance of $n/2$ apart. This grouping corresponds to applying a partial column permutation Q that keeps the overall structure (though not the actual coefficients) of the lower part of the private parity-check matrix unchanged, namely:

$$H_{pk}Q = S(H_{sk}PQ)$$

where

$$H_{sk}PQ = \begin{bmatrix} A' & B' \\ -W' & W' \end{bmatrix}$$

for some $A', B' \in \mathbb{F}_3^{r_U \times (n/2)}$ and $W' \in \mathbb{F}_3^{r_V \times (n/2)}$.

Moreover, doing this allows to compute a different basis for the same code where the structure of the lower part of the parity-check matrix is publicly visible. Namely, it becomes possible to find a matrix R such that:

$$RH_{pk}Q = \begin{bmatrix} A & B \\ -W & W \end{bmatrix}.$$

The obvious technique to achieve this is to compute the echelon form of a matrix Z consisting of the sum of the left and right sides of $H_{pk}Q$, namely:

$$Z := H_{pk}Q \cdot \begin{bmatrix} \mathbf{I} \\ \mathbf{I} \end{bmatrix} = S \cdot \begin{bmatrix} A' & B' \\ -W' & W' \end{bmatrix} \cdot \begin{bmatrix} \mathbf{I} \\ \mathbf{I} \end{bmatrix} = S \cdot \begin{bmatrix} A' + B' \\ \mathbf{0} \end{bmatrix},$$

since the rank of Z is no more than r_U. Thus, the unscrambling matrix R we look for is one such that:

$$R \cdot Z = \begin{bmatrix} Z' \\ \mathbf{0} \end{bmatrix}$$

for some Z'. Such a matrix is effectively computed via plain Gaussian elimination, and one can check that $H_{bk} := RH_{pk}Q$ has the desired form.

It remains to show how to make use of H_{bk} to forge Wave signatures. We will discuss this in the next section.

5 Forging Signatures

As we have seen, the permutation Q and the unscrambling matrix R, recovered through leaks from genuine signatures, reveal part of the private parity-check structure, yielding a parity-check matrix H_{bk} with the structure:

$$H_{bk} := RH_{pk}Q = \begin{bmatrix} A & B \\ -W & W \end{bmatrix}.$$

A crucial remark is that the permutation Q likewise restores the structure $e' = (e'_U, e'_U + e'_V)$ of valid signatures, even though the actual coefficients probably differ from those of the actual signature. This means that $e'[j] = \pm e'[j + n/2] \neq 0$ for all but $\lfloor n - 2r_U/3 \rfloor$ entries of e' when $e'_V[j] \neq 0$, that the ratio between equal and opposite values in such matching pairs is about 2, and that the coefficients of e'_V are essentially random (only the coefficients of e'_U have to be chosen according to the same guidelines as before).

Now consider the task of decoding a syndrome $s' := (s_U, s_V)$ into a w-dense error pattern e' under the parity check matrix H_{bk}, that is, solving $e'H_{bk}^T = s'$. We have:

$$\begin{aligned} e'H_{bk}^T &= \begin{bmatrix} e'_U & e'_U + e'_V \end{bmatrix} \cdot \begin{bmatrix} A^T & -W^T \\ B^T & W^T \end{bmatrix} \\ &= \begin{bmatrix} e'_U A^T + (e'_U + e'_V)B^T & -e'_U W^T + (e'_U + e'_V)W^T \end{bmatrix} \\ &= \begin{bmatrix} e'_U (A+B)^T + e'_V B^T & e'_V W^T \end{bmatrix} \\ &= \begin{bmatrix} s'_U & s'_V \end{bmatrix}. \end{aligned}$$

This can be carried out by finding a random solution to $e'_V W^T = s'_V$, then solving $e'_U(A+B)^T = s'_U - e'_V B^T$. To keep the Hamming weight high, simply follow the same strategy as for the original Wave private structure: set $n/2 - r_U$ coefficients $e'_U[j] = e'_V[j]$ when $e'_V[j] \neq 0$ or to a random nonzero value when $e'_V[j] = 0$, then solve for the remaining r_U coefficients.

To forge signatures for any message (properly hashed to a syndrome s), notice that any valid signature satisfies $eH_{pk}^T = s \Leftrightarrow (eQ)(Q^T H_{pk}^T R^T) = sR^T \Leftrightarrow e'H_{bk}^T = s'$, with $e' := eQ$ and $s' := sR^T$. Thus we can forge a Wave signature e under key H_{pk} for a given syndrome s by recovering H_{bk} as shown in the previous section, then solving $e'H_{bk}^T = s'$ for e' of the correct weight with $s' := sR^T$, and finally returning $e = e'Q^T$.

6 Measurements and Results

We have implemented our attack via a simple Magma script, which we have included in the Appendix. Below, we report the results obtained when running the script on Wave parameters. The machine used is an iMac, with processor Intel Core i5 at 3.2 GHz and 16Gb of RAM. In the script we actually allow for the Hamming weight of genuine signatures to be within 5% of the target weight. This is by no means a restriction of the attack: it is merely a way to speedup the generation of legitimate Wave signatures, a process which could otherwise be unnecessarily slow: while it is not hard to get signatures whose weight is ever so slightly off the target $w = 4980$, a legitimate user would have to spend quite some time to obtain a signature of weight exactly w, perhaps tenfold or even longer. Yet, once these genuine signatures are available, the attack proceeds without any difference in running time, namely, just a few seconds.

Table 1. Timings (in seconds) to perform a full attack on 128-bit Wave parameters.

Collect signatures	Recover structure	Forge signature	Total time
59.83	6.15	0.12	66.10

The first column indicates the time spent to collect the required number of signatures (set to 600). This is the most expensive part of the attack. The second column reports the time necessary to recover the matrix structure corresponding to the alternative private key. Finally, the time to produce a forgery (which is negligible) is reported in the last column. Overall, the attack takes just over a minute.

7 Generalizing the Attack

As we have mentioned, the Wave scheme is formally defined on top of generalized admissible $(U, U+V)$ codes over some finite field $\mathbb{F}_p$. The authors choose to restrict the scheme description to plain $(U, U+V)$ codes of characteristic 3, justifying their choice for the sake of simplicity, but pointing out that their construction and analysis can be generalized. Although they refrain from providing explicit details on the use of these codes, we briefly show that they would still be vulnerable to simple variants of our attack, while also pointing out the limitations of our technique. We first consider the adoption of ternary generalized admissible $(U, U+V)$ codes, then we discuss larger characteristics.

7.1 Attacking Generalized Admissible $(U, U+V)$ Codes

The detailed structure of the private parity-check matrix for a generalized admissible $(U, U+V)$ code is the following:

$$\begin{bmatrix} H_U D_4 M & -H_U D_2 M \\ H_V D_3 M & -H_V D_1 M \end{bmatrix},$$

where D_1 through D_4 and M are diagonal matrices, with D_1, D_3 and M invertible, and $M := (D_1 D_4 - D_3 D_2)^{-1}$. In this description, plain $(U, U+V)$ codes correspond to the particular choice of matrices $D_1 = D_3 = D_4 = \mathbf{I}$ and $D_2 = \mathbf{0}$.

Since M and D_1 through D_4 are diagonal, they commute. Moreover, given the condition that D_1 and D_3 are both invertible, we can define $E := D_3 \cdot D_1^{-1}$ and rewrite the above matrix in the following form as:

$$\begin{bmatrix} (H_U D_4 M) \cdot (D_3 D_1^{-1})^{-1} \cdot (D_3 D_1^{-1}) & (-H_U D_2 M) \\ -(-H_V D_1 M) \cdot (D_3 D_1^{-1}) & (-H_V D_1 M) \end{bmatrix}$$

or, equivalently,

$$\begin{bmatrix} AE & B \\ -WE & W \end{bmatrix} = \begin{bmatrix} A & B \\ -W & W \end{bmatrix} \begin{bmatrix} E & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$$

where $A := (H_U D_4 M) \cdot (D_3 D_1^{-1})^{-1}, B := -H_U D_2 M$ and $W := -H_V D_1 M$.

This form is very closely related to the recovered matrix in the attack against plain $(U, U+V)$ codes, the difference consisting only of the diagonal matrix:

$$F := \begin{bmatrix} E & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$$

on the right. The effect of multiplying an error pattern (without the private column permutation) by F is just to consistently flip the sign of certain paired columns: while in the plain case $e[j] = -e[j + n/2] \neq 0$ about $2/3 - \varepsilon/2$ of the time and $e[j] = e[j + n/2] \neq 0$ about $1/3 - \varepsilon/2$ of the time (the ε being due to $2r_U/n$ computed entries whose values cannot be freely chosen), on columns where the diagonal element on F is -1 rather than 1 these probabilities are swapped.

This, however, does not prevent an attacker from finding out which columns are paired in the public permuted code, and also determining whether the corresponding entry on E is 1 or -1: the lack of balance between equal and opposite values for those entries is still roughly 2/3 to 1/3, while it is 1/2 to 1/2 for unrelated entries. Moreover, the elements on the diagonal of E are inferred to be 1 if opposite signs predominate in a certain matched pair, and to be -1 if equal signs predominate instead.

In conclusion, an attacker can still recover an equivalent private key for generalized admissible $(U, U+V)$ codes with only a small modification to the basic attack strategy for plain $(U, U+V)$ codes. The recovered equivalent trapdoor, in a sense, is even more general than the structure of generalized admissible $(U, U+V)$ codes, since it does not require that complicated decomposition to work.

7.2 Larger Characteristics

Assume now that the codes are defined over $\mathbb{F}_p$ for $p \geq 3$. The distinguisher we employ for ternary codes to pair up matching columns on legitimate signatures works by matching pairs (h, j) of columns that are not selected to be the solutions of linear systems. It is based on the observation that $e[j] = e[h]$ only when $e_V[j] = 0$ (which happens with probability around $1/p$), while if the columns do not match equality happens by chance among all nonzero values (which happens with probability around $1/(p-1)$). Thus, we count the number of times the values of two columns match, and infer they match when the count is close to $1/p$ rather than $1/(p-1)$.

For $p = 3$ the distinguisher works with a comfortable margin (1/3 of equalities for matching columns against 1/2 for unmatching ones). For larger characteristics all we need to make is a small modification to the way we count equalities/inequalities.

The generalized distinguisher for any $p \geq 3$ is designed to lead again to an expected zero-sum when the columns do not match, but to an expected sum value of roughly t/p for matched columns, where t is the number of collected signatures. To attain this, we simply compute a *weighted* sum instead, assigning weight $-\frac{p-1}{2}$ to count occurrences of $e[j] = e[h]$ and weight $\frac{p-1}{2(p-2)}$ to count occurrences of $e[j] \neq e[h]$.

Thus, when the columns are unmatched, the fraction of equal coefficients between columns h and j is mapped from $\frac{1}{p-1}$ to $-\frac{1}{p-1} \cdot \frac{p-1}{2} = -\frac{1}{2}$, and the fraction of different coefficients on those columns is mapped from $\frac{p-2}{p-1}$ to $\frac{p-2}{p-1} \cdot \frac{p-1}{2(p-2)} = \frac{1}{2}$, yielding the expected zero-sum. In contrast, for matched columns the fraction of equal coefficients is mapped from $\frac{1}{p}$ to $-\frac{1}{p} \cdot \frac{p-1}{2} = -\frac{p-1}{2p(p-2)} \cdot (p-2)$, and the fraction of different coefficients is mapped from $\frac{p-1}{p}$ to $\frac{p-1}{p} \cdot \frac{p-1}{2(p-2)} = \frac{p-1}{2p(p-2)} \cdot (p-1)$, therefore with a bias of $\frac{p-1}{2p(p-2)} \approx \frac{1}{2p}$ above the zero-sum in favor of matching column pairs.

Adapting the statistical analysis from Section 4.1 accordingly, we conclude that the number of required signatures for the generalized distinguisher is $t \approx (2p\alpha)^2 = 4\alpha^2 p^2$ where α is again the number of standard deviations that an event with the same probability as an attack failure would be away from the mean. Thus, the odds of attack success become negligible when $p^2 \in O(2^\lambda)$ for λ-bit security. More precisely, since the signature size is proportional to $\lg p$, the total number of collected bits from t signatures becomes $4\alpha^2 p^2 \lg p$, and setting this value to 2^λ yields a refined lower bound for p, namely, $p \approx 2^{\lambda/2-1}/(\alpha\sqrt{\lambda})$. For instance, for $\lambda = 128$ and $\alpha = 6$ (corresponding to a probability of roughly $n/506797346 \approx 2^{-17}$ that the attack fails for the proposed $n = 5172$), the minimum characteristic to thwart the attack would be $p \approx 2^{57}$.

Yet, it is far less clear how to choose secure parameters that prevent other types of vulnerability for p as large as this: it might well turn out that other vulnerabilities are introduced by this setting, which we therefore neither recommend nor claim to be secure at all. Investigating the possibilities left, if any, transcends the scope of this paper.

8 Conclusion

We have described distinguishers for (plain or generalized admissible) $(U, U+V)$ codes in small-to-moderate characteristic, exploiting leaks in genuinely generated signatures that enable recovering an equivalent signing key. The recovered key can in turn be used to sign arbitrary messages, not merely to produce an existential forgery. Thus the attack constitutes a total break of Wave, and shows that the hardness assumption required for that scheme to be secure does not hold.

The attack is practical as corroborated by empirical assessment (see the Appendix for a simple but complete implementation). Proposed parameters for the 128-bit classical security level can be broken in about a minute, with the actual key recovery taking only a few seconds after a modest number of genuine signatures are collected. Since Wave is tightly attached to codes whose structure is susceptible to attacks, we do not see how the scheme can be repaired, except maybe for considerably large characteristics (assuming such a setting does not introduce further vulnerabilities).

Acknowledgments

We are indebted and grateful to the authors of Wave for their comments on preliminary versions of this document.

References

1. Courtois, N.T., Finiasz, M., Sendrier, N.: How to achieve a McEliece-based digital signature scheme. In: Boyd, C. (ed.) Advances in Cryptology — ASIACRYPT 2001. pp. 157–174. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
2. Dallot, L.: Towards a concrete security proof of Courtois, Finiasz and Sendrier signature scheme. In: Lucks, S., Sadeghi, A.R., Wolf, C. (eds.) Research in Cryptology. pp. 65–77. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
3. Debris-Alazard, T., Sendrier, N., Tillich, J.P.: Wave: A new code-based signature scheme. arXiv preprint arXiv:1810.07554 (2018)
4. Faugere, J., Gauthier-Umana, V., Otmani, A., Perret, L., Tilllich, J.P.: A distinguisher for high-rate McEliece cryptosystems. IEEE Transactions on Information Theory **59**(10), 6830–6844 (Oct 2013). https://doi.org/10.1109/TIT.2013.2272036
5. Faugère, J.C., Gauthier-Umaña, V., Otmani, A., Perret, L., Tillich, J.P.: A distinguisher for high rate McEliece cryptosystems. In: Information Theory Workshop (ITW), 2011 IEEE. pp. 282 –286 (oct 2011). https://doi.org/10.1109/ITW.2011.6089437
6. Lyubashevsky, V.: Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In: Matsui, M. (ed.) Advances in Cryptology – ASIACRYPT 2009. pp. 598–616. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
7. Persichetti, E.: Improving the efficiency of code-based cryptography. Ph.D. thesis, ResearchSpace@ Auckland (2012)

8. Stern, J.: A new identification scheme based on syndrome decoding. In: Stinson, D.R. (ed.) Advances in Cryptology — CRYPTO' 93. pp. 13–21. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)
9. Véron, P.: Improved identification schemes based on error-correcting codes. Appl. Algebra Eng. Commun. Comput. **8**(1), 57–69 (1996)

A Magma script to illustrate the attack

```
K := GF(3);

/**
 * Compute the Hamming weight of a vector.
 */
function Hweight(e)
    w := 0;
    for k in [1..Ncols(e)] do
        if e[k] ne 0 then
            w +:= 1;
        end if;
    end for;
    return w;
end function;

/**
 * Find any solution e of e*H^T = s, observing that
 * e*H^T = s <=> ek*Hk^T + er*Hr^T = s <=> er*Hr^T = s - ek*Hk^T
 * <=> er = (s - ek*Hk^T)*Hr^-T:
 */
function LinSolve(s, HT, hint)
    n := Nrows(HT); r := Ncols(HT); k := n - r; assert Ncols(s) eq r;
    A := {u : u in K} diff {0};
    // guess k components:
    repeat
        // choose the columns of the k components to guess:
        J := {1..n};
        e := Vector(K, n, [0 : j in [1..n]]); ss := s;
        for c in [1..k] do
            j := Random(J); J diff:= {j};
            // guess one component and update the syndrome accordingly:
            if #hint eq 0 then
                e[j] := Random(K); // entirely random
            elif hint[j] eq 0 then
                e[j] := Random(A); // random nonzero
            else
                e[j] := hint[j];
            end if;
            ss -:= e[j]*HT[j];
        end for;
        // compute the remaining coefficients:
```

```
        L := [];
        for j in [1..n] do
            if j in J then
                L cat:= [j];
            end if;
        end for;
        HH := VerticalJoin([HT[L[i]] : i in [1..r]]);
    until Determinant(HH) ne 0;
    ee := ss*HH^-1;
    c := 0;
    for j in [1..n] do
        if j in J then
            c +:= 1; e[j] := ee[c];
        end if;
    end for;
    assert c eq r;
    //assert e*HT eq s;
    return e;
end function;

/**
 * Compute the matrix R such that R*M is in echelon form, and also the rank of M.
 */
function Echelon(M)
    r := Nrows(M); m := Ncols(M);
    A := HorizontalJoin(M, IdentityMatrix(K, r)); // [ M | I ]
    n := Ncols(A);
    // echelonize the i-th row:
    p := 0; // pivot column
    for i in [1..r] do
        repeat
            p +:= 1;
            // force unit pivot on the p-th column:
            for k in [i..r] do
                if A[k, p] ne 0 then
                    scale := A[k, p]; // A[k, p]^-1; // no inversion if char 3
                    // swap k-th and i-th rows and
                    // normalize i-th row from the p-th column onward:
                    swap := Submatrix(A, k, p, 1, n - p + 1);
                    InsertBlock(~A, Submatrix(A, i, p, 1, n - p + 1), k, p);
                    InsertBlock(~A, scale*swap, i, p);
                    break; // done pivoting
                end if;
            end for;
        until A[i, p] eq 1 or p eq m;
        // clear the p-th column below the i-th line:
        for k in [i+1..r] do
            if A[k, p] ne 0 then
                scale := -A[k, p];
                InsertBlock(~A, Submatrix(A, k, p, 1, n - p + 1)
                    + scale*Submatrix(A, i, p, 1, n - p + 1), k, p);
            end if;
        end for;
    end for;
    // compute rank(M) from its echelon form:
    rank := 0;
    for i in [1..r] do
        if &and[A[i, j] eq 0 : j in [1..m]] then
            break;
        end if;
        rank +:= 1;
    end for;
    //assert Rank(M) eq rank;
    return Submatrix(A, 1, m+1, r, r), rank;
end function;

/**
 * Create a random (U, U+V) Wave key pair.
 */
function KeyGen(rU, rV, n)
    n2 := n div 2; assert rU lt n2; assert rV lt n2;
    r := rU + rV;
    // create a matrix Hsk of form [  U  O ]:
    //                             [ -V  V ]
    U := Matrix(K, rU, n2, [Random(K) : i in [1..rU], j in [1..n2]]);
    UT := Transpose(U);
    V := Matrix(K, rV, n2, [Random(K) : i in [1..rV], j in [1..n2]]);
    VT := Transpose(V);
    O := ZeroMatrix(K, rU, n2);
    Hsk := VerticalJoin(HorizontalJoin(U, O), HorizontalJoin(-V, V));
    // create an invertible scrambling matrix S:
    repeat
        S := Matrix(K, r, r, [Random(K) : i, j in [1..r]]);
    until Determinant(S) ne 0;
    SinvT := Transpose(S^-1);
    // create a permutation matrix P:
    P := ZeroMatrix(K, n, n);
    J := {1..n};
    for i in [1..n] do
        j := Random(J); J diff:= {j};
        P[i, j] := 1;
    end for;
    // compute the public key:
    Hpk := S*Hsk*P;
    return UT, VT, SinvT, Hsk, P, Hpk;
end function;

/**
 * Sign a given syndrome s given a Wave private key (S^-T, U^T, V^T, P)
 * to an error pattern of weight w +- wThreshold.
 *
 * NB: The weight threshold is used only to speed up the generation of valid signatures.
 * It is irrelevant for the attack, and could be set to zero (for signing and verification).
 */
function Sign(SinvT, UT, VT, P, s, w, wThreshold)
    r := Ncols(s); assert Nrows(SinvT) eq r; assert Ncols(SinvT) eq r;
    n := Nrows(P); k := n - r; assert Ncols(P) eq n; assert n mod 2 eq 0; assert r lt n;
    n2 := n div 2;
    rU := Ncols(UT); kU := n2 - rU; assert Nrows(UT) eq n2; assert rU lt n2;
    rV := Ncols(VT); kV := n2 - rV; assert Nrows(VT) eq n2; assert rV lt n2;
    assert r eq rU + rV;
    // compute the privately decodable syndrome:
    ss := Eltseq(s*SinvT);
    sU := Vector(K, rU, ss[1..rU]);
    sV := Vector(K, rV, ss[rU+1..r]);
    // find any solution of eV*V^T = sV:
    eV := LinSolve(sV, VT, []);
    // find a dense solution of eU*U^T = sU s.t. |wt(eU, eU + eV) - w| <= wThreshold:
    satt := 0;
    repeat
        satt +:= 1;
        eU := LinSolve(sU, UT, Eltseq(eV));
        e := Vector(K, n, Eltseq(eU) cat Eltseq(eU + eV));
        wt := Hweight(e);
        assert wt ge n - 2*rU and wt le n;
    until Abs(wt - w) le wThreshold or satt gt 100;
    assert satt lt 100; // random sigs have weight very close to, but often not exactly, w.
    // permute the error pattern:
    return e*P;
end function;

/**
 * Verify a purported signature e for syndrome s
 * under public key Hpk within a given weight threshold of w.
 *
 * NB: The weight threshold is used only to speed up the generation of valid signatures.
 * It is irrelevant for the attack, and could be set to zero (for signing and verification).
 */
function Verify(Hpk, e, s, w, wThreshold)
    return Abs(Hweight(e) - w) le wThreshold and e*Transpose(Hpk) eq s;
end function;

/**
 * Basic Wave key generation, signing & verification tests.
 */
procedure TestKeySigVer(rU, rV, n, w, wThreshold, keys, sigs)
    if keys gt 0 and sigs gt 0 then
        "**** Testing signing/verification:", keys, "key(s), ", sigs, "sig(s) per key...";
        r := rU + rV; assert n gt rU + rV;
        fail := 0;
        for key in [1..keys] do
            UT, VT, SinvT, Hsk, P, Hpk := KeyGen(rU, rV, n);
            for sig in [1..sigs] do
                s := Vector(K, r, [Random(K) : j in [1..r]]);
                e := Sign(SinvT, UT, VT, P, s, w, wThreshold);
                fail +:= (Verify(Hpk, e, s, w, wThreshold)) select 0 else 1;
            end for;
        end for;
        if fail eq 0 then
            "**** Testing complete. No failures detected.";
        else
            "**** Testing complete. Failures:", fail;
        end if;
    end if;
end procedure;

/**
 * Collect a number of legitimate signatures created under a given Wave private key.
 */
function CollectSigs(SinvT, UT, VT, P, w, wThreshold, numSig)
    ZZ := Integers();
    r := Nrows(SinvT); assert Ncols(SinvT) eq r;
    assert Nrows(UT) eq Nrows(VT); assert Ncols(UT) + Ncols(VT) eq r;
    n := Nrows(P); assert Ncols(P) eq n;
    // collect signatures:
    sigTab := ZeroMatrix(ZZ, numSig, n);
    for sig in [1..numSig] do
        s := Vector(K, r, [Random(K) : j in [1..r]]);
        e := Sign(SinvT, UT, VT, P, s, w, wThreshold);
        sigTab[sig] := Vector(ZZ, n, [(e[j] eq 2) select ZZ!(-1) else ZZ!e[j] : j in [1..n]]);
    end for;
    return sigTab;
end function;

/**
 * Recover the matching of column pairs from both sides of the private (U, U+V)
 * parity-check matrix, given the corresponding public key
 * and statistics inferred from the collected signatures.
 */
function RecoverQ(Hpk, sigTab)
    r := Nrows(Hpk); n := Ncols(Hpk); n2 := n div 2; assert n mod 2 eq 0;
    // pair up matching columns:
    count := -Transpose(sigTab)*sigTab;
    n := Nrows(count);
    pair := [0 : j in [1..n]];
    for j in [1..n] do
        // guess the pair for the target column:
        maxval, paired := Max(Eltseq(count[j]));
        pair[j] := paired;
    end for;
    assert &and[pair[pair[j]] eq j : j in [1..n]];
    // compute a corresponding column permutation:
    Q := ZeroMatrix(K, n, n);
    done := {};
    col := 0;
    for j in [1..n] do
        if j in done then
            continue;
        end if;
        col +:= 1;
        Q[j][col] := 1;
        Q[pair[j]][col + n2] := 1;
        done join:= {j, pair[j]};
    end for;
    //assert Determinant(Q) ne 0;
    return Q;
end function;

/**
 * Find a different but equally consistent permutation of pairs of matching columns.
 */
function RandomizeQ(Q)
    n := Nrows(Q); n2 := n div 2; assert Ncols(Q) eq n; assert n mod 2 eq 0;
    QQ := ZeroMatrix(K, n2, n2);
    A := {1..n2};
    for i in [1..n2] do
        j := Random(A); A diff:= {j};
        QQ[i, j] := 1;
    end for;
    //assert Determinant(QQ) ne 0;
    Q *:= DiagonalJoin(QQ, QQ);
    return Q;
end function;

/**
 * Recover a basis of the public code that enables forging signatures,
 * given the public parity-check matrix Hpk
 * and the column permutation Q that pairs up matching columns.
 *
 * Return the basis change matrix R
 * (and possibly a modified but equally valid permutation Q of matching columns).
 */
function RecoverR(Q, Hpk, rU)
    r := Nrows(Hpk); n := Ncols(Hpk); n2 := n div 2;
    assert Nrows(Q) eq n; assert Ncols(Q) eq n; assert n mod 2 eq 0;
    rV := r - rU; assert rU lt r;
    zatt := 0;
    repeat zatt +:= 1;
        // prepare linear system:
        Hvk := Hpk*Q;
        Z := Submatrix(Hvk, 1, 1, r, n2) + Submatrix(Hvk, 1, n2+1, r, n2);
        R, rZ := Echelon(Z);
        assert rZ eq rU;
        if rZ lt rU then
            // try a different permutation of matching column pairs:
            Q := RandomizeQ(Q);
        end if;
    until rZ ge rU or zatt gt 100;
    //"zatt =", zatt;
    assert zatt eq 1;
    assert rZ ge rU;
    //assert IsZero(Submatrix(R*Z, rU + 1, 1, rV, n2));
    //assert Determinant(R) ne 0;
    return Q, R; // NB: Q has potentially been updated
end function;

/**
 * Recover a full equivalent Wave private key for a given public key
 * given a suitable collection of legitimate signatures.
 */
function RecoverStructure(Hpk, rU, sigTab)
    r := Nrows(Hpk); n := Ncols(Hpk); n2 := n div 2; assert n mod 2 eq 0;
    rV := r - rU; assert rU lt r;
    // exploit statistics to pair up columns:
    Q := RecoverQ(Hpk, sigTab);
    qatt := 0;
    repeat
        repeat
            qatt +:= 1;
            Q, R := RecoverR(Q, Hpk, rU);
            Hbk := R*Hpk*Q;
            //assert IsZero(Submatrix(Hbk, rU+1, 1, rV, n2) + Submatrix(Hbk, rU+1, n2+1, rV, n2));
            BT := Transpose(Submatrix(Hbk, 1, n2+1, rU, n2)); // B^T
            DT := Transpose(Submatrix(Hbk, 1, 1, rU, n2)) + BT; // (A + B)^T
            rD := Rank(DT);
            assert rD eq rU;
        until rD eq rU or qatt gt 100;
        assert qatt le 100;
        WT := Transpose(Submatrix(Hbk, rU+1, n2+1, rV, n2)); // W^T
        rW := Rank(WT);
    until rW eq rV or qatt gt 100;
    //"qatt =", qatt;
    assert qatt le 100;
    return Q, R, BT, DT, WT;
end function;

/**
 * Forge a Wave signature for an arbitrarily given syndrome,
 * given an equivalent trapdoor corresponding to the legitimate public key.
 */
```

```
function ForgeSignature(R, BT, DT, WT, Q, Hpk, s, w, wThreshold)
    r := Nrows(Hpk); n := Ncols(Hpk); n2 := n div 2; rU := Ncols(BT);
    assert Nrows(Q) eq n; assert Ncols(Q) eq n; assert n mod 2 eq 0;
    rV := r - rU; assert rU lt r;
    // map it to a trapdoor-decodable syndrome:
    sp := Eltseq(s*Transpose(R));
    sU := Vector(K, rU, sp[1..rU]);
    sV := Vector(K, rV, sp[rU+1..r]);
    // find any solution of eV*W^T = sV:
    eV := LinSolve(sV, WT, []);
    // find a dense solution of e_U D^T = s_U - e_V B^T.
    att := 0;
    repeat
        att +:= 1;
        eU := LinSolve(sU - eV*BT, DT, Eltseq(eV));
        e := Vector(K, n, Eltseq(eU) cat Eltseq(eU + eV));
        wt := Hweight(e);
        assert wt ge n - 2*rU and wt le n;
    until Abs(wt - w) le wThreshold or att gt 100;
    assert att le 100;
    //"**** att =", att;
    // permute the error pattern:
    e := e*Transpose(Q);
    return e;
end function;

// some toy parameters for testing:
//kU := 23; kV := 16; k := kU + kV; n := 52; r := n - k;
//kU := 46; kV := 32; k := kU + kV; n := 104; r := n - k;
//kU := 92; kV := 64; k := kU + kV; n := 208; r := n - k;
//kU := 230; kV := 161; k := kU + kV; n := 518; r := n - k;
//kU := 460; kV := 322; k := kU + kV; n := 1034; r := n - k;
//kU := 920; kV := 644; k := kU + kV; n := 2068; r := n - k;
//kU := 1150; kV := 805; k := kU + kV; n := 2586; r := n - k;
//kU := 1840; kV := 1288; k := kU + kV; n := 4132; r := n - k;
// actually proposed 128-bit level parameters:
kU := 2299; kV := 1609; k := kU + kV; n := 5172; r := n - k;

n2 := n div 2;
rU := n2 - kU;
rV := n2 - kV;
w := n - Ceiling(2*rU/3);
wThreshold := Ceiling(w/20); // 5% tolerance, just for practicality of legitimate signatures
"n  =", n,  ": w  =", w;
"kU =", kU, ": rU =", rU;
"kV =", kV, ": rV =", rV;
"k  =", k,  ": r  =", r;

//TestKeySigVer(rU, rV, n, w, wThreshold, 0, 0);
```

```
// create a sample key pair:
UT, VT, SinvT, Hsk, P, Hpk := KeyGen(rU, rV, n);

// collect signatures:
numSig := 600; // in case of failure, slightly increase this number
"collecting", numSig, "signatures..";
time sigTab := CollectSigs(SinvT, UT, VT, P, w, wThreshold, numSig);

//"breaking Wave...";

"recovering structure...";
time Q, R, BT, DT, WT := RecoverStructure(Hpk, rU, sigTab);

"forging signature...";
// get target syndrome:
s := Vector(K, r, [Random(K) : j in [1..r]]);
time e := ForgeSignature(R, BT, DT, WT, Q, Hpk, s, w, wThreshold);

"**** success?", Verify(Hpk, e, s, w, wThreshold);
```