Plactic signatures (insecure?)

Daniel R. L. Brown*

November 21, 2022

Abstract

Plactic signatures use the plactic monoid (semistandard tableaus with Knuth's associative multiplication) and full-domain hashing (SHAKE). Monico found an attack which likely makes plactic signatures insecure.

1 Introduction

Plactic signatures instantiate multiplicative signatures (see Table 1, §2, and [?]), with the plactic monoid[1] and full-domain hashing (see §3).

Notation	Name	Typically:
a	(Attested) Matter	A message digest
b	(Binding) Secret Key	**SECRET** to one signer
c	Checker	System-wide
d	(Digital) Signature	Attached to message
e	Endpoint	Signer-specific value
$[a, d]$	Signed Matter	Thing to be verified
$[c, e]$	Public Key	Certified as signer's
$e = bc$	Key Generation	Signer uses secret key b
$d = ab$	Signing	Signer uses secret key b
$ae = dc$	Verifying	Verifier uses public information

Table 1: Summary of plactic (and multiplicative) signatures

*danibrown@blackberry.com

[1]For more about Knuth's plactic monoid, one can start from [?] or Wikipedia.

2 Multiplicative signatures

This section describes **multiplicative** signatures, which are summarized in Table 1. Rabi and Sherman [**?**] mentioned the main idea behind multiplicative signatures in 1993.

Multiplicative signatures can be defined for any multiplicative semigroup. Security and efficiency depend on the semigroup.

Custom terminology for multiplicative signatures helps to discuss features specific to multiplicative signatures.

2.1 Multiplicative semigroups: a brief review

Recall the definition of a (**multiplicative**) **semigroup**. Firstly, a semigroup is a set together with a multiplication operation. This means, in more detail, that multiplication is a well-defined binary operation, and the set is closed under multiplication. Multiplication of variables a and b is written as ab, whenever clear from context. Secondly, multiplication must be associative, which means that it obeys the associative law: $a(bc) = (ab)c$. As an example, matrices with positive integer entries form a semigroup, under standard matrix and integer arithmetic.

2.2 The plactic monoid

This report focuses on one semigroup: Knuth's **plactic monoid**.

An introduction (for cryptographers) to the plactic monoid may be found in [**?**]. A general introduction to the plactic monoid may be found on Wikipedia. The reader is assumed henceforth to have some familiarity with the plactic monoid, to facilitate discussion of how the plactic monoid elements and multiplication are used in plactic signatures.

Elements of the plactic monoid are **semistandard tableaus**. Elements of the plactic monoid can also be represented by sequences. The canonical sequence representation is the row reading, a concatenation of the tableau's rows. More generally, every sequence represents a unique tableau, the tableau obtained by applying the Robinson–Schensted algorithm to the sequence. A tableau typically has many different sequence representations, but only one canonical representation, the row reading of the tableau.

In the plactic signatures, the tableaus d and e (from Table 1) should be communicated using the canonical representation. (Other representations risk leaking the secret key b.)

Multiplication in the plactic monoid is Knuth multiplication of semistandard tableaus, which can be implemented by: first, concatenating all the rows (row readings) of the two tableaus; second, applying the Robinson–Schensted to the resulting concatenated sequence to obtain a semistandard tableau; and third, taking the row reading as the canonical representation.

2.3 Public keys

A **public key** is a pair $[c, e]$ of elements. Element c is the **checker** and element e is the **endpoint**. The checker c can be shared between many signers, but endpoint e is usually specific to a single signer.

Alternative names for the public key in a digital signature include the following. A common term is **verification** key, because the public key is the key that the verifier needs to verify a signature. A common graphical symbol (and arguably more user-friendly indication) is a **lock**. However, the lock symbol often indicates several layers of security (for example, in web browsers, a lock symbol typically indicates successful verification of several digital signatures within a certificate chain of trusted public keys, and also an application of encryption and other symmetric-key cryptography).

For simplicity, this report presumes that a signer's public key is reliably and correctly available to the verifier.

In practice, a **public key infrastructure** (PKI) would be used to establish each signer's public key $[c, e]$, binding the cryptographic value $[c, e]$ to a more legible name of the signer. The signer's public key $[c, e]$ will be embedded into a **certificate**, which certifies that the $[c, e]$ belongs to the signer. A typical hierarchal PKI distributes some certificates manually as **root certificates**, and then transfers trust to other certificates using digital signatures (which could be plactic signatures).

2.4 Digital signatures

A **signed matter** is a pair $[a, d]$ of elements. Element a is the (**attested**) **matter** and element d is the (**digital**) **signature**. The matter is usually derived as a digest of a meaningful message (such as legible text). A matter is sometimes common to many signers (for example, when short messages like "yes" or "no" are to be signed). We often say that d is a signature **on** matter a, or that d is a signature **over** a.

A signed matter $[a, d]$ is **verifiable** for public key $[c, e]$ if

$$ae = dc, \tag{1}$$

which we call the **verification equation**. We often also say that $[a, d]$ is **valid** for $[c, e]$, or that signer $[c, e]$ has **signed** matter a, or that matter a has been signed **by** $[c, e]$, or that d is a signature **under** $[c, e]$.

If the two sides of (1) are different, then signed matter $[a, d]$ is non-verifiable for public key $[c, e]$. We also often say that d is an invalid signature.

To enable discussion of the two sides of the verification equation (1), such as in the case of invalid signatures or the during the course implementing verification, we can call ae the **endmatter** and call dc the **signcheck**. So, with this terminology, a multiplicative signature is valid if and only if the endmatter equals the signcheck.

2.5 Secret keys

A **secret key** b for public key $[c, e]$ is an element b such that

$$e = bc. \tag{2}$$

Alternative names for secret keys of digital signatures include the following. The commonly used term **private key** can be useful to distinguish from other secret keys, such as keys used in symmetric-key cryptography. Another sensible term is **signature generation** key, or **signing** key. A more mnemonic name for b is **binder**, but this is quite far from any existing traditions (which use different letter variables for the secret key).

Note an exception to this terminology in this report's implementations of plactic signatures. The value b, which is a semistandard tableau, will be derived by computing the hash of a shorter seed key. (Two reasons for this: to save on the amount of long-term secret data that needs protection, and to slightly discourage poorly chosen values for b.) In this case, the secret seed key can then be considered the secret signing key, while the tableau b is an intermediate temporary secret that is only computed during the signing and key generation procedures. But, for a security analysis, the attacker can win by finding b, entirely by-passing the seed secret key. So, even if b is derived from a seed secret key, the value b remains the **effective** secret key.

The rest of this section, which describes plactic signatures at a higher level, refers to b itself as the secret key.

A public key $[c, e]$ is **viable** if there exists at least one secret key b for $[c, e]$.

A signer can choose secret key b before choosing a public key $[c, e]$, by computing the endpoint e from the formula (2). This results in a viable public key. In the plactic monoid, it seems difficult to generate a viable public key $[c, e]$ in any way other than computing $e = bc$ for some b.

A **weak secret key** b for public key $[c, e]$ is an element b such that $abc = ae$ for all matters a (in the set of matters to be signed). In the plactic monoid, allowing matters a to range over a large set, then it seems likely that every weak secret key is a secret key. (For some other semigroups, this might not be the case.)

2.6 Signing

A signer with secret key b can sign matter a by computing signature

$$d = ab. \tag{3}$$

The resulting signed matter $[a, d]$ is valid for the signer's public key $[c, e]$, because multiplication is associative:

$$ae = a(bc) = (ab)c = dc. \tag{4}$$

A signer with public key $[c, e]$ should keep b secret, so that nobody else can generate signatures under $[c, e]$.

2.7 Key and hash spaces

For security reasons, the values a, b and c should be chosen from sufficiently secure subsets A, B, and C of the chosen semigroup (the plactic monoid). Furthermore, secure methods to choose elements a, b, c in the subsets A, B, C should be specified.

In plactic signatures with the recommended parameters, each of a, b, and c will be obtained by the same general method: as a sequence of 500 bytes, taken from the output of the extendable output function SHAKE (part of the SHA-3 hash function). The only difference will be the inputs to SHAKE: for a, the SHAKE input is the message being signed; for b, the SHAKE input is the secret seed key (a string of 100 bytes, which should be chosen uniformly at (pseudo)random); for c, the SHAKE input is a fixed, system-wide byte string (or perhaps a string customized to the signer).

3 Hashed multiplicative signatures

Hashed multiplicative signatures are multiplicative signature system in which the matter is a hash of a message. Hashed multiplication signatures are summarized in Table 2. In hashed multiplicative signatures, both the signer

Notation	Name	Typically:
a	Matter	A message digest
b	Secret key	**SECRET** to one signer
c	Checker	System-wide
d	Raw signature	Attached to the message
e	Endpoint	Signer-specific value
f	Hash function	Fixed or signer-chosen
h	Fixed hash function	System-wide, fixed or keyed, $f() = h_k()$
k	Hash function key	Fixed or signer chosen $f() = h_k()$
m	Message	Reviewed (or chosen) by signer
$[d, f]$	Signature	Extension of multiplicative signature
$[m, d, f]$	Signed message	Thing to be verified
$[c, e]$	Public key	Certified as signer's
$a = f(m)$	Digesting	Signer and verifier compute short a
$e = bc$	Key generation	Signer uses secret key b
$d = ab$	Signing	Signer uses secret key b
$ae = dc$	Verifying	Verifier uses public information

Table 2: Summary of hashed multiplicative signatures

and verifier compute the matter a from the message m by applying hash function f:

$$a = f(m). \tag{5}$$

A **hashed signature** is $[d, f]$, and the **signed message** is $[m, d, f]$.

To **sign** message m, the signer with secret key b selects f and computed $f = ab = f(m)b$. To **verify** signed-message $[m, d, f]$ under public key $[c, e]$, the verifier checks that $f(m)e = dc$.

Generally, security of the signature schemes requires that f be chosen securely. In other words, hashed signature $[d, f]$ being valid requires that hash function f is secure. One of the simple systems described next, will help to assure verifiers that f is a secure hash function.

3.1 Choice of hash function

The hash function f will typically take the form $f(m) = h_k(m)$, where h is a keyed hash function, and k is a key for h. Because h is fixed across the whole system, the key k suffices to specify f. This allows f to have a short specification, so that the signature $[d, f] = [d, k]$ is not too long. (So, it not necessary to specify the algorithm for f in each signature.)

The hash key k can be fixed across the whole system, with the same key k in every signature. In this mode, the signed message $[a, d, f]$ reduces to $[a, d]$, because it is unnecessary for the signer to transmit k to a verifier. Plactic signatures use this mode.

Alternatively, each signer might want choose a different hash key k for each signature. For example, the signer might choose k as a deterministic, pseudorandom function of the message, like this $k = h_b(m)$. The unique k must be then be appended to the signature, which is an extra cost. A potential benefit of this approach is that it obviates the need for a collision-resistant hash function.

Plactic signatures, which use a fixed hash function, can approximate the mode of having a unique key k per signature, by using message randomizers. For example, just prepend the message with a unique key k. Assuming that the fixed hash is strong enough, the random bytes prefixing the message effectively make the fixed hash into a keyed hash. This approach would require the verifier to be aware of the randomizer, and to remove it from the signed message to recover the actual content part of the message.

Multiplicative signatures (from the previous section) can be considered to be a special case of hashed multiplication signatures where the hash function f is the identity function, meaning $f(m) = m$. Pure multiplicative signatures only allow us to sign messages that are already elements in the semigroup.

With the plactic monoid, purely multiplicative signatures would be possible, because any byte string a can represent a tableau. However, the resulting signature length would be proportional to the message length. Also, collisions in this identity function would be easy to find, since a given tableau has multiple representations (and finding them is easy). Collisions would result in certain types of forgery attacks against the signature scheme. Therefore, plactic signatures use hashed multiplicative signatures.

In this report's reference implementation, a fixed, system-wide hash function is suggested, based on the FIPS 202 hash function, SHAKE-128, which is extendable output version of SHA-3.

3.2 Full-domain (embedded) hashing

The hash function must map messages into a set A of semigroup elements. Some form of **full-domain**, **embedded** hashing is needed. Embedding refers to the the step of mapping the natural output of the hash function, usually a byte string, into the semigroup. Full-domain hashing refers to the idea that the hashed matters $a = f(m)$ should appear indistinguishable from a randomly chosen from the set A.

In the case of plactic signatures, we will assume that all entries in the semistandard tableaus have numeric values from 0 to 255. So, each entry of the tableau can then be represented as a single byte. Every byte string represents a semistandard tableau, since the Robinson–Schensted algorithm converts any byte string s into a semistandard tableau $P(s)$.

The embedding function used in plactic signatures is therefore to take the byte string output of the hash function, and consider it to be a representation of a semistandard tableau.

Towards getting a full-domain hash function, an **extendable output** hash function can be used, meaning that the hash function can output as many bytes as needed for the chosen byte size of the matter a. In this case A represents all semistandard tableaus of a given length.

The Robinson–Schensted map $s \mapsto P(s)$ is not injective, so it introduces a bias (non-uniformity) in the tableaus when the input is a unbiased (uniformly distributed) byte string. For digital signatures, this bias seems quite harmless. It slightly increases the chances of collisions, meaning messages with the same effective hash value, here the tableau a. Plactic signatures attempt to mitigate this risk by using strings of greater length.

3.3 Usability benefits of hashing

A usability benefit of hashing is that a long message m can have short hash $a = f(m)$. A short a usually means that the signature $d = ab$ is short. In other words, $f(m)b$ is shorter than mb (for some embedding of m into the semigroup).

Another usability benefit of hashing is that hashing algorithms can be faster than semigroup multiplication. In the plactic monoid, semigroup multiplication runs in time that is super-linear, sub-quadratic in the input length, whereas hash functions run in time that is linear in the input length. In other words, for long messages m, computing $f(m)b$ is faster than computing mb.

Security benefits of hashing are discussed in §6.

4 Suggested parameters

For concreteness, this report suggests some specific parameters.

4.1 Recommended parameters: ps8000

The recommended set of parameters, `ps8000`, is described below.

- Tableau entries are bytes, numbers ranging from 0 to 255.
- A tableau is represented by a byte string: string s representing tableau $P(s)$ (where P is the Robinson–Schensted algorithm).
- Values a, b, c are each 500 bytes (4000 bits).
- Values d, e are each 1000 bytes (8000 bits).
- Row readings (of semistandard tableaus) are the default representations d and e.
- Endpoint e represents public key $[c, e]$.
- Value c is fixed system-wide, or communicated out-of-band.
- The hash function is SHAKE-128.
- The hash function output length is 500 bytes.
- The embedding function is the identity function, sending byte strings (500 bytes from SHAKE-128) to byte strings (representing semistandard tableaus).
- Value c is the hash of a fixed system-wide byte string, the algorithm name, or perhaps some other string communicated out-of-band.
- Value a is the hash of a message to be signed (so is different for each message signed).
- Value b is the hash of a 100-byte string, which is to be considered the private signing key.

- A signed message consists of the concatenation of d and the message m, with d first, so a signed message is exactly 1000 bytes longer than the message signed.

The main aim for parameters `ps8000` is that any successful forgery attack (with success rate at least one half) takes computation of at least 2^{128} steps (bit operations). Furthermore, more general attack strategies should be infeasible in some other way, such as by having negligible success probability, or by having excessive number of queries to honest signers.

To clarify, parameters `ps8000` permit d and e to be other byte string representations of tableaus, not just row readings. For example, column readings are considered valid. But, the risk attached to alternative representations is the signer's responsibility. For example, a signer could just use simple concatenations to compute $d = ab$, which would be much faster, but which would be totally insecure.

4.2 Optional: strict row reading mode

As an optional strict set of requirements, we can insist on one or both the following conditions:

- Signature d is the row reading of a semistandard tableau.
- Endpoint e is the row reading of a semistandard tableau.

In other words, a signature d or endpoint e could be deemed invalid if it is not the row reading the a semistandard tableau.

The recommended default formats for d and e are row readings. In the stricter mode, a signer would be forbidden from using other byte string representations of tableaus, such as column readings. A verifier would then reject other representations, including column readings.

In the stricter mode, the verifier must do an extra check on d and e to ensure they are proper row readings. This extra check takes a small amount of computation. It arguably protects signers from using insecure representations. Signer who uses insecure representation, such as concatenation would find their signatures rejected by strict verifiers. This would be de-incentivize signers from using insecure representations, but would do so artificially, by transferring the signer's security responsibility partially to the verifier.

Strict mode is optional. It is not used by verification implementations in this report. Signing and key generation do comply with the strict mode, because they do following the recommended default of using row reading representations of tableaus for d and e.

4.3 Other parameters (to be completed)

Cryptanalysts can easily use reduced size parameters to test out various attack strategies.

Using larger parameters as a margin for error, traditionally a reasonable precaution, seems an incongruous for a new cryptographic system like plactic signatures.

Perhaps a more thorough cryptanalysis might fortuitously suggest that smaller signatures are just as safe as the `ps8000` parameters. Maybe a and b could be reduced to 300 bytes each, instead of 500 bytes each.

5 Implementations of plactic signatures

This section presents some C implementations of plactic signatures.

Implementations of cryptographic algorithms are essential to their applications, gauge their practicality, resolve ambiguities in their written descriptions, and develop their more thorough understanding.

The NIST post-quantum cryptography project has a required application programming interface (API) for C implementations[2]. This API makes it simple for application programmers to use a variety of possible digital signature algorithm in a uniform manner, without needing to interact with the algorithm internal. The application programmer mainly needs to know the byte lengths of the inputs and outputs of three C functions.

This report's implementations of plactic signatures try to use the NIST required API.

5.1 Common header files

The various C files in this report's implementations include the following header files to declare various common types, macros, and constants.

A header file `types.h` listed in Table 3 defines abbreviations for C types used often throughout the implementation, using the `typedef` mechanism.

[2]The NIST PQC project takes its requirements from the SUPERCOP system of timing cryptography.

```
typedef unsigned char u; typedef unsigned long long ll;
```

Table 3: File `types.h`

File `api.h` listed in Table 4 is an application programming interface (API) header file required by the NIST PQC project. First, the file specifies the byte sizes for keys (secret and public) and signature. Next, the file specifies a string formalizing algorithm's name (including key size parameters). Finally, it declares the required C function prototypes for key generation, signing and verifying (using the abbreviated type names defined in file `types.h`).

```
#define CRYPTO_SECRETKEYBYTES   100
#define CRYPTO_PUBLICKEYBYTES  1000
#define CRYPTO_BYTES           1000
#define CRYPTO_ALGNAME  "placticsignature8000bits"
#include "types.h"
int crypto_sign_keypair(u*pk, u*sk);
int crypto_sign(u*sm, ll*smlen, const u*m, ll mlen, const u*sk);
int crypto_sign_open(u*m, ll*mlen, const u*sm, ll smlen, const u*pk);
```

Table 4: File `api.h`

File `lengths.h` listed in Table 5 specifies the byte sizes of various intermediate array variables arising during the course of signing. The main signing implementation uses these lengths. The reference implementation of plactic monoid multiplication also uses the lengths to determine the size of memory to allocate for a buffer that stores the entries of the tableau.

5.2 Implementing plactic monoid multiplication

This section provides a few implementations of plactic monoid multiplication, which is the core operation of plactic signatures.

5.2.1 A header file for plactic monoid multiplication

The C implementations share common interface, described by the header file `plactic.h` listed in Table 6.

```
#include "api.h"
enum {
  sklen = CRYPTO_SECRETKEYBYTES,
  elen  = CRYPTO_PUBLICKEYBYTES,
  dlen  = CRYPTO_BYTES,
  blen  = (dlen<elen?dlen:elen)/2,
  alen  = dlen - blen,  clen = elen - blen,
  aelen = alen + elen, dclen = dlen + clen};
```

Table 5: File `lengths.h`

```
#include "types.h"
void multiply(u*ab, const u*a, int alen, const u*b, int blen);
```

Table 6: File `plactic.h`

In this interface, the intent is that the inputs `a` and `b` to the function `multiply` are byte strings of lengths given by input `alen` and `blen`. The input byte strings represent tableaus, which could be row readings of semistandard tableaus, or perhaps just arbitrary byte strings obtained as the output of a hash function.

The output `ab` is byte string of length `alen`+`blen`. The output `ab` is a byte string representing a tableau, the product of tableaus represented by byte strings `a` and `b`. Generally, `ab` is intended to be the row reading of the tableau, which is the canonical representation of the tableau.

A different tableau representation might be acceptable too, but the representation should minimum not leak information about the input tableaus. The worst case would be a plactic monoid multiplication that just concatenates the input strings, because the concatenation is one possible representation of the output in the plactic monoid. The concatenation representation would immediately reveal the input strings, which is totally insecure, because the signer's effective secret key would be revealed by the public key. By contrast, a column reading instead of a row reading should be fine. A randomized representation might work for signatures and public keys, but a canonical representation seems necessary for signature verification. It is unclear (to me) if any alternate can safely offer much advantage over the row reading representation.

Beware that the caller of function `multiply` needs to ensure that enough memory is allocated for byte strings at locations pointed to by variables `a`, `b`, and `ab`, with the room for `alen`, `blen`, and `alen`+`blen` bytes respectively.

Beware that, some of the plactic monoid multiplication implementations might not tolerate overlapping of the strings pointed by `a`, `b`, `ab`, (although the reference implementation tries to tolerate such an overlap). If the output `ab` overlaps with inputs `a` or `b`, then the multiplication might also modify the inputs `a` or `b`. (Proper array bounds checking might avoid this problem, but this report's implementations skips some of these mitigations.)

5.2.2 A reference implementation

File `plactic-ref.c` listed in Table 7 is a reference implementation of plactic monoid multiplication. Beware that the reference implementation does not aim for side channel resistance.

```
/* Plactic multiplication. (c) Dan Brown, BlackBerry, 2021 */
// Reference
#include "lengths.h" // aelen, u
#define swap(a,b) (a^=b, b^=a, a^=b)
enum{aemax=aelen, rmax=256, smax=((aemax*49)/8)};
void insert(u**t, int*r, u v){ int i=0, j;
  for(; i<rmax &&  r[i] && v<t[i][r[i]-1]; swap(t[i][j], v), i++)
    for(j=0; j<r[i] && t[i][j]<=v; j++) ;
  t[i][r[i]++] = v;}
void multiply (u*ab, const u*a, int alen, const u*b, int blen){
  int i, j, ablen=alen+blen, r[rmax]={0}; u s[smax], *t[rmax]={s};
  for(i=1; i<rmax; i++) t[i] = t[i-1] + ablen/i;
  for(i=0; i<ablen;i++) insert (t, r, (i<alen)? a[i]: b[i-alen]);
  for(i=rmax; i--;) for(j=0; j<r[i]; j++) *ab++ = t[i][j];}
```

Table 7: File `plactic-ref.c`

The reference implementation allocates a memory buffer `s` to store the rows of the product tableau. Each row occupies a fixed place in `s` with enough room between rows such that insertion of new entries does not require any shifting of rows. The buffer's size is proportional to the signature parameters, such as the `ps8000` parameters. The included file `lengths.h` provides `aelen`, deduced from the signature parameters, which is then used to calculate the buffer size. (The defined type `u` is also provided by `lengths.h`.)

The reference implementation assumes that the tableau entries are confined to byte values, 0–255 (as represented by type `u`). This assumption implies that are at most 256 rows (because the tallest column has distinct entries), a fact that the reference implementation uses by defining a constant `rmax=256`. The maximum size `smax` of the linear buffer to store the tableau's rows is calculated by rounded-up multiplication `aemax` by the harmonic number $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{256} < 6.125 = 49/8$.

Function `multiply` allocates three array variables `r`, `s`, and `t`. Array `r` records the row lengths of the tableau, and is initialized with all row lengths are zero. Array `s` is a buffer where the tableau entries are stored: each row of the tableau will occupy a distinct segment of `s`, chosen so that each segment is long enough to store the maximum possible length for the given row. Array `t` is an array of pointers into these segments of `s`: so doubly-indexed `t[i][j]` is the value of the individual tableau entry located at row i and column j.

The first `for` loop in function `multiply` sets up the pointers `t[i]` at fixed locations within `s`. The second `for` loop iterates Robinson–Schensted insertion, running over bytes of input `a` and then bytes of input `b`. The third `for` loop is a doubly-nested `for` loop, and places the row reading of the table stored `t` into the output array `ab`.

Function `insert` implements the Robinson–Schensted insertion, by inserting `v` into the tableau represented by the pair of arrays `t` and `r`. The first `for` loop bumps entries into the rows. Nested inside the first `for` loop is a second inner `for` loop that finds the correct entry to be bumped. When there is no entry to bump, the two nested `for` loops are done. The last line of function `insert` appends a new entry to the end of the next available row, and increments the length of that row by one.

5.2.3 Binary searching in rows

File `plactic-search.c` listed in Table 8 modifies the reference implementation `plactic-ref.c`, by using a binary search (instead of a left-to-right scan) within a row to find which entry needs to be bumped to the next row and replaced with the next element to insert.

In more detail, file `plactic-search.c` modifies the `insert` function from `plactic-ref.c` by replacing a scan search by a binary search in the lowest 64 rows. The motivation for binary searching in only lowest 64 rows, is that

```
/* Plactic multiplication. (c) Dan Brown, BlackBerry, 2021 */
// Binary search (in lower rows)
#include "lengths.h" // aelen, u
#define swap(a,b) (a^=b, b^=a, a^=b)
enum{aemax=aelen, rmax=256, smax=((aemax*49)/8)};
void insert(u**t, int*r, u v){ int i=0, j=aemax, k, m;
  for(; i<rmax && r[i] && v<t[i][r[i]-1]; swap(t[i][j], v), i++)
    if(i<64)
      for(k=r[i]-1, k=(j<k)?j:k, j=0; t[i][j]<=v; )
        v<t[i][m=(++j+k)/2]? swap(m,k): swap(m,j) ; else
      for(j=0; j<r[i] && t[i][j]<=v; j++) ;
  t[i][r[i]++] = v;}
void multiply (u*ab, const u*a, int alen, const u*b, int blen){
  int i, j, ablen=alen+blen, r[rmax]={0}; u s[smax], *t[rmax]={s};
  for(i=1; i<rmax; i++) t[i] = t[i-1] + ablen/i;
  for(i=0; i<ablen;i++) insert (t, r, (i<alen)? a[i]: b[i-alen]);
  for(i=rmax; i--;) for(j=0; j<r[i]; j++) *ab++ = t[i][j];}
```

Table 8: File plactic-search.c

in short rows, left-to-right scanning is faster than binary search, probably because there is less overhead calculation. An approximate tuning by trial and error led to the choice of 64 to minimize verification time. Experiments using row length, which can only be determined at run-time to decide between binary search or left-to-right scan, suggested that run-time decisions were slower.

With the recommended parameters `ps8000`, the binary search method gives a small speed-up (over the reference implementation `plactic-ref.c`): approximately 5% faster signing, and 15% faster verifying.

Table 9 shows the differences – excluding spacing differences – between the files `plactic-search.c` and `plactic-ref.c`: the main difference being that a binary search is applied to the bottom 64 rows.

```
$--$ diff -b plactic-search.c plactic-ref.c
2c2
< // Binary search (in lower rows)
---
> // Reference
6c6
< void insert(u**t, int*r, u v){ int i=0, j=aemax, k, m;
---
> void insert(u**t, int*r, u v){ int i=0, j;
8,10d7
<       if(i<64)
<         for(k=r[i]-1, k=(j<k)?j:k, j=0; t[i][j]<=v; )
<           v<t[i][m=(++j+k)/2]? swap(m,k): swap(m,j) ; else
```

Table 9: Differences between `plactic-search.c` and `plactic-ref.c`

5.2.4 Sped-up plactic multiplication

Generally, the signer computed d as the row reading of a tableau. A verification can check a signature d for this condition, and also then speed up the can speed up the computation of dc (the signcheck), because the reason is that signer has already completed part of the Robinson–Schensted algorithm to compute dc, namely the part applied to d.

File `plactic-prep.c` listed in Table 10 implements this trick to speed up verification by about 15% (over file `plactic-search.c`). Many new line of C code have been added to, but these new lines do not use much run-time. The new C code scans d, assuming row reading format, and check that the row reading is correct.

File `plactic-prep.c` also applies a different tuning of the decision process for choosing between scan and binary search, but that accounts for only a 3% speed-up.

5.2.5 Low-memory plactic multiplication

File `plactic-lowmem.c` listed in Table 11 aims to use less memory than the reference implementation, at the cost of longer runtime.

To do this, function `multiply` first concatenates input arrays `a` and `b` into the the output array `ab`. Then it iteratively applies Knuth's relations to the array `ab`, achieving the same effect as the Robinson–Schensted algorithm. Instead of scanning rows of a two-dimensional array, and then swapping entries with variable `v`, adjacent elements of a one-dimensional array are swapped, in-place.

Some previous versions of this report had named `plactic-lowmem.c` as the reference implementation of plactic monoid multiplication. This previous choice was partly because the low memory code is actually simpler than the two-dimensional array version and partly because the use of Knuth's relations ties the implementation more closely connected to the monoid algebraic structure, and thus to idea behind multiplicative signatures. But the current reference implementation, is now based on the Robinson–Schensted algorithm instead. This is partly because Robinson–Schensted pre-dates Knuth relations, partly because Robinson–Schensted algorithm seems to be considered more widely than the Knuth relations.

5.2.6 Towards constant-time multiplication

File `plactic-constant.c` listed in Table 12 is a step towards implementing plactic monoid multiplication in constant time, which is a step towards preventing some side channel attacks. The cost of this potential security improvement is, unfortunately, much longer run-time.

File `plactic-constant.c` is constructed by modifying file `plactic-lowmem.c`. The secret-dependent branching statements from `plactic-lowmem.c` are re-

```
/* Plactic multiplication. (c) Dan Brown, BlackBerry, 2021 */
// Binary search, sped-up if d has row reading form ...
#include "lengths.h" // aelen, u
#define swap(a,b) (a^=b, b^=a, a^=b)
enum{aemax=aelen, rmax=256, smax=((aemax*49)/8)};
void insert(u**t, int*r, u v, int l){int i=0, j=aemax, k, m;
  for(; i<rmax && r[i] && v<t[i][r[i]-1]; swap(v,t[i][j]), i++)
    if(128<l || i<32)
      for(k=r[i]-1, k=(j<k)?j:k, j=0; t[i][j]<=v /* && j<r[i]*/; )
        v<t[i][m=(++j+k)/2]? swap(m,k): swap(m,j) ; else
      for(j=0; j<r[i] && t[i][j]<=v; j++) ;
  t[i][r[i]++] = v;}
int check(u**t,int*r,int s){int i,j,k=r[0];
  for(i=1;i<rmax;k+=r[i++]) {
      if(r[i]>r[i-1]) return 0;
      for(j=0; j<r[i]; j++)
        if(t[i][j] <= t[i-1][j]) return 0;}
  if(k!=s) return 0;
  for(i=0;i<rmax;i++)
    for(j=1; j<r[i]; j++)
      if(t[i][j] < t[i][j-1]) return 0;
  return 1;}
void wipe(int*r){int i;for(i=0;i<rmax;i++)r[i]=0;}
void multiply (u*ab, const u*a, int alen, const u*b, int blen){
  int i, j, k=0, ablen=alen+blen, r[rmax]={0}; u s[smax], *t[rmax]={s};
  for(i=1; i<rmax; i++) t[i] = t[i-1] + ablen/i;
  if(alen==dlen){
    for (i=k=0; k<alen-1; k++) i+=a[k+1]<a[k];
    for (j=k=0; k<alen  ; k++){
      if(0<=i && i<rmax && j<alen/(i+1)) t[i][j]=a[k], r[i]++;
      if (k<alen-1 && a[k+1] < a[k]) i--, j=0; else j++;}
    if (!check(t,r,alen)) wipe(r),k=0;}
  for(; k<ablen; k++) insert (t, r, (k<alen)? a[k]: b[k-alen], k);
  for(i=rmax-1; 0<=i; i--) for(j=0; j<r[i]; j++) *ab++ = t[i][j];}
```

Table 10: File `plactic-prep.c`

```
/* Plactic multiplication. (c) Dan Brown, BlackBerry, 2021 */
// low memory implementation
#include "types.h"
#define swap(a,b) (a^=b, b^=a, a^=b)
#define knuth(k, xyz) \
(    (xyz[2] <  xyz[k-1]) && \
     (xyz[0] <= xyz[k])   && \
(swap(xyz[1] ,  xyz[(k+1)%3]), 1==1))
void multiply (u*ab, const u*a, int alen, const u*b, int blen){
  int i,j,k;
  for(i=0; i<alen+blen; i++)
    ab[i] = (i<alen)? a[i]: b[i-alen];
  for(i=0; i<alen+blen; i++)
    for(j=i-2; 0<=j && ab[j+2] < ab[j+1]; j--)
      for(k=1; k<=2; k++)
        for(; 0<=j && knuth(k, (ab+j) ) ; j--)  ;}
```

Table 11: File `plactic-lowmem.c`

placed by non-branching statement that use boolean-based arithmetic, instead of conditional instructions.

In more detail, file `plactic-constant.c` attempts to be constant-time by running a state machine with four states $\{-1, 0, 1, 2\}$. The states 1 and 2 correspond to Knuth's two relations defining the plactic monoid. The states 0 and -1 are used to when to manage whether there is need to apply the transformation associated with the Knuth relations. State 0 means that there is still to apply them, while state -1 indicates that no further transformations are needed.

Unfortunately, the code in file `plactic-constant.c` still uses secret-dependent array-indexing and the C remainder operator "`%`". Both of these C operations are known to lead to side channel attacks under certain conditions.

A signature implementation using file `plactic-constant.c` failed to pass the tests require by the TIMECOP. This is unsurprising, since, for example, `plactic-constant.c` uses secret-dependent array-indexing.

Because the constant-time implementation is much slower than the reference implementation, an alternative side channel mitigation might be useful.

```
/* Plactic multiplication. (c) Dan Brown, BlackBerry, 2021 */
// Towards constant-time?
#include "types.h"
#define swap(a,b,c) (c*=a-b, b+=c, a-=c)
#define xyz(i) xyz[(i)%3]
int knuth(int h, u*xyz){
  u d= (xyz[1] <= xyz[2])   & (0==h),
    e= (xyz[2] <  xyz(2+h)) &
       (xyz[0] <= xyz(0+h)) & (0<h);
  swap (xyz[1] ,  xyz(1+h), e);
  return d + (0!=e) + (0>h);}
void multiply (u*ab, const u*a, int alen, const u*b, int blen){
  int g,h,i,j,k;
  for(i=0; i<alen+blen; i++)
    ab[i] = (i<alen)? a[i]: b[i-alen];
  for(i=0; i<alen+blen; i++)
    for(h=0, j=i-2; j>=0; j-=g, h+=1-k, h%=3){
      k=knuth(h, ab+j);
      g  = k | (2==h);
      h -= k & (0==h);}}
```

Table 12: File `plactic-constant.c`

5.2.7 Jeu de Taquin (to be completed)

Schutzenberger defined the jeu de taquin, a non-deterministic algorithm, that can be used as yet another way to determine the semistandard tableau of a given sequence.

The jeu de taquin approach has not been implemented for this report. The jeu de taquin approach looks to be less efficient than the approach based on Knuth's relations, since jeu de tauqin looks to have slighlty worse than quadratic run-time.

Perhaps, an implementation of jeu de taquin would realize significant benefits over the other approaches used in this report.

5.3 Signing implementation

File `sign.c` listed in Table 13 implements key generation, signing and verifying, following the interface defined in file `api.h` listed in Table 4, and using the helper definitions from file `sign-defs.c` listed in Table 14.

The implementation `sign.c` places the digital signature $d = ab$ as prefix of the signed message, rather than the more traditional suffix to the signed message. The rationale for the prefixed signature is to somewhat protect a user who mistakenly processes a signed message file with a user application instead of using `crypto_sign_open` to verify the signature and extract the acutal message. Typical user applications begin processing at beginning of the file. For such application, the first file data encountered is the signature prefix $d = ab$, which is not in a form that the application can process. Thus, the appliation or the user will quickly see that the signed message is not the usual file expected the for application.

By contrast, a suffix signature in a long signed message might allow a user application to start processing the signed message, even if the signature suffix is invalid. In other words, a suffix singnature suffers from a greater security risk from user accident.

(Yet further protection against accident non-verified application processing of signed messages is a message recovery. For plactic signatures, a message recovery could be achieved as follows. The signer takes 1000-byte signature as input to SHAKE-256 to generate a masking stream of length equal to the message. The masking stream is XORed with the message. The signed message is concatentaion of the signature (as a prefix) and the masked message. To verify and open, the process is reverse. This mode might prevent more

```
/* Plactic signatures. (c) Dan Brown, BlackBerry, 2021. */
#include "sign-defs.c"
int crypto_sign_keypair(u*pk, u*sk){
  u *b=pk; u *c=b+blen, *e=b;
  if (sk != pk) random(sk);
  digest (b, sk), digest (c, name);
  multiply (e, b, c);  return  0;}
int crypto_sign(u*sm, ll*smlen, const u*m, ll mlen, const u*sk){
  u *a=sm; u *b=a+alen, *d=a;
  digest (a, m), digest (b, sk);
  multiply (d, a, b), copy (sm + dlen, m), *smlen = dlen + mlen;
  return 0;}
int crypto_sign_open(u*m, ll*mlen, const u*sm, ll smlen, const u*pk){
  u new(ae), new(dc); u *a=ae, *c=dc+dlen; const u *d=sm, *e=pk;
  *mlen=0;
  if (smlen < dlen) return  -4;
  smlen -= dlen, sm += dlen;
  digest (a, sm), digest (c, name);
  multiply (ae, a, e), multiply (dc, d, c);
  if (0 != compare (ae, dc)) return -1;
  *mlen = smlen; copy(m, sm); return 0;}
```

Table 13: File sign.c

user accidents, such as reading from the tail end of an invalid signed message, but has an extra performance cost, doubling the amount of hashing for long messages. Perhaps, instead of SHAKE-256 to generate a mask, a much faster making algorithm would suffice. But is this cryptographic quality protection? Is this modelling the user as adversary against the user? Note that this further protection can be applied aftermarket, in the sense that it can easily be applied and removed plactic signatures as defined by `sign.c`.)

```
#include <string.h>
#include "plactic.h"
#include "keccak.h"
#include "rng.h"
#include "lengths.h"
#define namelen          (u64)strlen((char*)name)
#define new(a)           a[a##len]
#define compare(a,b)     memcmp (a, b, (size_t)b##len)
#define copy(a,b)        ((a!=0&&a!=b)? memcpy(a, b, (size_t)b##len): 0)
#define multiply(ab,a,b) multiply(ab, a, a##len, b, b##len)
#define digest(t,m)      FIPS202_SHAKE128(m, m##len, t, t##len)
#define random(a)        (void)randombytes(a, a##len)
u*name=(u*)CRYPTO_ALGNAME;
```

Table 14: File `sign-defs.c`

The implementation `sign.c` sets a default value for the checker tableau c to be system-wide, as the output of the hash function SHAKE-128, applied to the official name parameters of the plactic signatures `placticsignature8000bits`.

Optionally, a programmer using `sign.c` can change the value of this checker, as follows. The programmer can re-assign the global variable `name`, by pointing it to a string of the programmer's choice. The reference implementation `sign.c` will hash this string instead of the official algorithm name.

A programmer using `sign.c` can have `sign.c` generate a new secret key, or the programmer can supply secret key. To supply an old, pre-existing secret key, the programmer calls function `crypto_sign_keypair` with two equal pointers `pk` and `sk`. Equality of these pointers indicates to `sign.c` not to generate a random secret key, but rather to compute a public key from the given secret key. This is done by over-writing the memory location pointed to by `sk`. A programmer calling `sign.c` is presumed to be capable of maintaining a long-term secret key in a safe location, so that this over-writing will not destroy the only copy of the secret key.

5.4 Auxiliary implementations

File `rng-util.c` listed in Table 15 likely suffices for the way that a plactic signature utility would use random numbers.

```
#include <stdio.h>
#include "keccak.h"
#include "rng.h"
int randombytes(unsigned char *x, unsigned long long xlen){
  FILE *rng = fopen("/dev/urandom","r");
  if(!rng || xlen!=fread(x,1,xlen,rng)) {/*uh oh*/}
  FIPS202_SHAKE128 (x,xlen,x,xlen); return xlen;}
```

Table 15: File `rng-util.c`

The header file `rng.h` listed in Table 16 simply specifies the prototype for the function `randombytes`.

```
int randombytes(unsigned char *x, unsigned long long xlen);
```

Table 16: File `rng.h`

A header file `keccak.h` for the SHA-3 Keccak hash function is listed in Table 17.

```
typedef unsigned char u8; typedef unsigned long long u64;
void FIPS202_SHAKE128(const u8*in, u64 inLen, u8*out, u64 outLen);
```

Table 17: File `keccak.h`

File `keccak.c` listed in Table 18 is an indentation-added excerpt of one of the C implementations of SHAKE-128 from the official github source code for Keccak. This source code is highly condensed, but still considerably longer than than the source code for plactic monoid multiplication.

```
#include "keccak.h"
#define FOR(i,n) for(i=0; i<n; ++i)
typedef unsigned int ui;
void Keccak(ui r, /*ui c,*/ const u8 *in, u64 inLen, u8 sfx, u8 *out, u64 outLen);
void FIPS202_SHAKE128(const u8 *in, u64 inLen, u8 *out, u64 outLen)
{Keccak(1344, /*256,*/ in, inLen, 0x1F, out, outLen);}
static int LFSR86540(u8 *R) { (*R)=((*R)<<1)^(((*R)&0x80)?0x71:0); return ((*R)&2)>>1; }
#define ROL(a,o) ((((u64)a)<<o)^(((u64)a)>>(64-o)))
static u64 load64(const u8 *x) { ui i; u64 u=0; FOR(i,8) { u<<=8; u|=x[7-i]; } return u; }
static void store64(u8 *x, u64 u) { ui i; FOR(i,8) { x[i]=u; u>>=8; } }
static void xor64(u8 *x, u64 u) { ui i; FOR(i,8) { x[i]^=u; u>>=8; } }
#define rL(x,y) load64((u8*)s+8*(x+5*y))
#define wL(x,y,l) store64((u8*)s+8*(x+5*y),l)
#define XL(x,y,l) xor64((u8*)s+8*(x+5*y),l)
static void KeccakF1600(void *s) {
  ui r,x,y,i,j,Y; u8 R=0x01; u64 C[5],D;
  for(i=0; i<24; i++) {
    /*theta*/
    FOR(x,5) C[x]=rL(x,0)^rL(x,1)^rL(x,2)^rL(x,3)^rL(x,4);
    FOR(x,5) { D=C[(x+4)%5]^ROL(C[(x+1)%5],1); FOR(y,5) XL(x,y,D); }
    /*rho pi*/
    x=1; y=r=0; D=rL(x,y);
    FOR(j,24) { r+=j+1; Y=(2*x+3*y)%5; x=y; y=Y; C[0]=rL(x,y); wL(x,y,ROL(D,r%64)); D=C[0]; }
    /*chi*/
    FOR(y,5) { FOR(x,5) C[x]=rL(x,y); FOR(x,5) wL(x,y,C[x]^((~C[(x+1)%5])&C[(x+2)%5])); }
    /*iota*/
    FOR(j,7) if (LFSR86540(&R)) XL(0,0,(u64)1<<((1<<j)-1)); } }
void Keccak(ui r, /*ui c,*/ const u8 *in, u64 inLen, u8 sfx, u8 *out, u64 outLen) {
  /*initialize*/
  u8 s[200]; ui R=r/8; ui i,b=0; FOR(i,200) s[i]=0;
  /*absorb*/
  while(inLen>0) {
    b=(inLen<R)?inLen:R;
    FOR(i,b) s[i]^=in[i];
    in+=b; inLen-=b;
    if (b==R) { KeccakF1600(s); b=0; } }
  /*pad*/
  s[b]^=sfx;
  if((sfx&0x80)&&(b==(R-1))) KeccakF1600(s);
  s[R-1]^=0x80; KeccakF1600(s);
  /*squeeze*/
  while(outLen>0) {
    b=(outLen<R)?outLen:R;
    FOR(i,b) out[i]=s[i];
    out+=b; outLen-=b;
    if(outLen>0) KeccakF1600(s); } }
```

Table 18: File `keccak.c`

6 Plactic signature security

This section discusses forgery attack strategies against plactic signatures.

Some types of forgery attacks translate into various computational problems, such as division, cross-multiplication and parallel division.

6.1 Divide to find a secret key from public key

A secret key b for public key $[c, e]$ can be found by **division operator** (written / and called **divider** for short) with the computation

$$b = e/c. \tag{6}$$

If signatures are to be secure, then division must be difficult. More precisely, the division problem to compute e/c must be difficult for each public key $[c, e]$.

Recall (from [**?**]) that / is a divider if $((bc)/c)c = bc$ for all b, c. This means that e/c will be a secret key for public key $[c, e]$. Conversely, the ability to find a secret key from a public key, implies a divider (that works when the inputs are from a public keys $[c, e]$).

For some semigroups, but not the plactic monoid, a weaker kind of division suffices: $a((bc)/c)c = abc$ for all a, b, c. In other words, it suffices to find a weak secret key. In the plactic monoid, it seems that a weak secret key is a secret key, so that any weak divider is a divider.

6.2 Left division to find a secret key from a signature

Suppose that binary operator \ is a **left divider** (meaning $a(a\backslash(ab)) = ab$ for all a, b, as in [**?**]). Suppose that d is signature of matter a. Use left division to compute a value

$$b' = a\backslash d. \tag{7}$$

By definition of left division, we have $ab' = d$.

Consider a second matter a'. We could try to generate a signature $d' = a'b'$. This is valid if $a'e = d'c$, meaning $a'bc = a'(a\backslash d)c$. The latter equation is not guaranteed by the given definition of left division. In fact, in the plactic monoid, there are many different possible values for $a\backslash d$, because multiplication is not cancellative. It seems unlikely that $a'bc = a'b'c$ for $b \neq b'$, without somehow using a' and c to compute b'.

The plactic monoid is anti-isomorphic, so left and right division are equally difficult.

In cancellative semigroups, which does not include the plactic monoid, there is a **post-divider** such that $(ab)/b = a$ for all a, b. Similarly, a **left post-divider** has $a\backslash(ab) = b$ for all a, b. In that case, $b' = b$, so the secret key could be recovered from a signature using left division.

Although the plactic monoid is not cancellative, there might be a similar attack, via a **parallel left post-division** algorithm. Suppose that $d_i = a_i b$ for $i \in \{1, \ldots, n\}$, and that b is uniquely determined by the a_i and the d_i. A **parallel left post-division operator** finds b from the a_i and d_i, which we write as the formula $b = [a_1, \ldots, a_n]\backslash[d_1, \ldots, d_n]$. No good ideas for parallel division in the plactic monoid are known (to me).

Rather than finding a (parallel) post-divider, one may try to implement a **division-set operator**, written as $//$, and defined as:

$$d//b = \{a : ab = d\}. \tag{8}$$

In the context of multiplicative signatures, we use the left version of the division-set operator, $\backslash\backslash$, which is equivalent to the operator $//$ via the anti-automorphism of the plactic monoid. (In other semigroups, those non not anti-isomorphic, different algorithms may be needed for the left division). The attacker can compute the set $a\backslash\backslash d$. For a valid signature the actual secret key used belongs to this set: $b \in a\backslash\backslash d$. In this case, one can search for any $b \in a\backslash\backslash d$ such that $e = bc$, so that b is an effective secret key.

The erosion algorithm for division in the plactic monoid can easily be adapted to a division-set operator. A little empirical evidence suggest the following speculation: for random a and b (where $d = ab$), if division takes s steps on average, it seems that set $d//b$ has size approximately s on average, which can be made large. The time to compute the division-set might not be s times as much as a single division, because the computation between individual divisions overlaps significantly. Nonetheless, under this speculation one might be able to safely set the length of a to be as low as half the length of c.

6.3 Cross-multiply to forge unhashed signatures

A **cross-multiplier** is an operator written $*/$ such that

$$(y */ x)x = (x */ y)y, \tag{9}$$

whenever there exists u and v such that $ux = vy$. (So, if x and y are such that no such u and v, exist, then (9) is not required to hold.)

The notion of cross multiplication is common and familiar, being used to cancel terms between linear equations, for example. The notation $*/$ is not familiar, but convenient for the following discussions.

Some semigroups have fast cross-multipliers.

In a commutative semigroup, $x */ y = x$ defines a cross-multiplier. In a semigroup with a zero element 0 (such that $0z = 0$ for all z), $x */ y = 0$ defines a cross-multiplier. In a group with efficient inversion, $x */ y = y^{-1}$ defines a cross-multiplier. In the last example, division would be also be fast with $x/y = xy^{-1}$, but in the other two examples, division could potentially be much slower than cross-multiplication.

The plactic monoid is non-commutative, has no zero element, and has no inverses, so the three cross-multiplication methods above fail in the plactic monoid.

A cross-multiplier can be used for forgery of unhashed multiplicative signatures, by putting

$$[a, d] = [c */ e, e */ c]. \tag{10}$$

Because this forger uses the cross-multiplier $*/$ as an oracle, the forger has no control over the matter a (it is whatever the $*/$ algorithm outputs). This is therefore an **existential** forger (which could also be called **junk** message forger).

For hashed multiplicative signatures, the attacker would also need to find m (and f) such that $f(m) = c */ e$. For a secure hash function f such as SHAKE-128, finding such a message m should be difficult. In other words, forgery by cross-multiplication is not effective against hashed multiplicative signatures.

6.4 Dividing a signcheck by the endpoint

An attack can try to compute $(dc)/e$, where d is a genuine signature for some matter a. Because division is not cancellative, the division is likely to result in $(dc)/e = a' \neq a$.

For unhashed multiplicative signature, this would result in an existential forgery (with help from the signer, of one signed message, not necessarily chosen by the attacker). For hashed multiplicative signatures, the forger would need to invert the hash at a', which should be infeasible.

For plactic signatures, dividing by e should be slower than dividing by c, because e is longer than c, being $e = bc$.

The forger might try to use this method to generate a forgery without the help of the signer, by choosing d instead of getting d from the signer. But then, the forger faces the problem of finding d such that $dc = ue$ for some u. This is essentially the problem of cross-multiplication, already discussed.

6.5 Factor to forge unhashed signatures

To forge a matter a in an unhashed multiplicative, try to factor a as

$$a = a_2 a_1 \tag{11}$$

Then ask the signer to sign matter a_1. The signer returns signature d_1. Then compute $d = a_2 d_1$, which will a valid signature on matter a.

This would be a **chosen message** forgery (which could also be called a **signer-aided** forgery), because the forger chooses what message the signer honestly signs before getting to the forgery.

Factoring is easy in the plactic monoid. Therefore, unhashed multiplicative signatures would be vulnerable to this type of attack. For hashed multiplicative signatures, the factorization does not seem to be enough for forgery. Plactic signatures are hashed multiplication, so this attack seems to fail.

In particular, in plactic signatures, the matter length is fixed, so that any actual matter that a signer or a verifier uses cannot be factored into other matters.

If the verifier can be tricked into using longer matters, but the matters are still hashed, then factoring tableaus is not enough, because the attacker would also need to invert the hash on the factor a_2. If the signer can also be tricked into signing a matter without using a hash, then the factoring attack could work.

6.6 Re-divide to re-sign a message

Let **re-signing** refer to producing a second signature on a matter, especially when aided by observing a first signature of the matter. Traditionally, re-signing is not considered to be an attack on signatures because the matter in question has already been signed once.

Similarly, re-dividing refers to a computational problem in semigroups, related in the sense that re-dividing can be used for re-signing.

6.6.1 Re-signing details

Suppose that d is a valid plactic signature of matter a, meaning that $ae = dc$ for public key $[c, e]$. A **second signature** is a distinct signature $d' \neq d$ that is also valid, meaning that $ae = d'c$. A **re-signing** algorithm is an algorithm that can find a second signature from a first signature d, matter a and public key $[c, e]$.

The standard definition of security for signatures does not count re-signing as an attack. But more recent, more advanced signature security definitions have been introduced, under the name of **strong unforgeability**, that re-signing should be infeasible (without access to the secret signing key). In other words, these stricter security definitions regard re-signing as an attack.

Occasionaly, systems are designed that rely on signatures resistant to re-signing. Perhaps this reliance arises intuitively by modeling handwritten ink signatures or implicitly assuming an ideal oracle model for the signature interoperability, meaning anything not achievable by the oracle interface should be infeasible.

(For another example of such idealized model, consider the fact that the NIST API has the verifier open a signed message with the verification key to reveal the message. If the verification key were held as a secret, then the message should not be revealed by the signed message, in an idealized oracle operation for the verification. In other words, the idealized model, keeping the verification key secret could convert a signature scheme into some kind of encryption scheme. But many signature schemes form the signed message by concatenation of the message with a signature, in which case the signed message immediately reveals the message, without requiring the verification key. The point here is the idealizing the interfaces is perhaps not the ideal path to defining the security goals.)

In the case of plactic signatures, the condition $d' \neq d$ for a second signature can be interpreted at the concrete level of byte strings, or at the more abstract level of semistandard tableaus. Generally, it is easy to find multiple byte strings of a tableau, so the first type of second signature can be consider as a **representational** second signature, which is arguably quite bening. It is arguable more concerning to have an **monoidal** second signature, where d' and d represent distinct elemeents of the plactic monoid, meaning distinct semistandard tableaus.

Re-signing can also be done by the signer, using the secret signing key.

Regarding this is an attack is quite questionable, since the signer might be the only entity with a secret key, and yet this signer is also the attacker. Perhaps the signer would attempt to repudiate the first signature on the basis of the second. In any case, this is once again not considered as an attack under the standard definitions.

6.6.2 Re-dividing details

Re-dividing refers to the following problem in a semigroup (such as the plactic monoid). The inputs to the problem are semigroup elements x and y. An output is a semigroup element z such that $xy = zy$ and $x \neq z$.

In a general semgiroup, if y has right cancellation, then no such solution $z \neq x$ exists, meaning re-division would be impossible (not just infeasible!). But the plactic monoid does not have cancellation in most cases, so solutions to re-division can exist.

Division can be used for re-division, by choosing $z = (xy)/y$, and just ignoring the element x, and hoping that $z \neq x$. But perhaps re-division is much faster, finding a distinct z easily from x.

6.7 Attacking the hash function

An attacker can try to attack the hash function. The attacker can try to find collisions, for example. Plactic signatures use SHAKE-128, which has an internal state of 256 bits, and with an output length much higher, 4096 bits. Finding a collision by known methods, of byte string outputs of the hash, should therefore take at least 2^{128} steps.

But, the effective hash is the semistandard tableau represented by the byte string hash. Each tableau is represented by many different byte strings. Nevertheless, the space of tableaus is still large, so the output range of the hash still seems larger than 2^{256}, meaning generic collision attacks should still take at least 2^{128} steps.

6.8 Attacking message secrecy and identity secrecy

This section addresses some non-standard signature security goals that (like the case with resistance to re-signing discussed in §6.6), users sometimes intuitively expect, or even consciously want, what are considered from signatures. Occasionally, protocols are designed that rely on these non-standard features.

The non-standard goals are to resist non-standard signature attacks, in which, the message or the signer's identity is kept secret, and the attacker attempt to deduce these secrets from the signature, or perhaps even the public key. Normally, under standard definitions, signatures do not attempt hide the message or the signer's message. The purpose of signatures is to prove to while that the signer signed a message, so keeping this secret is counter to the overall purpose of signatures.

Users that want secret authentication instead of public authentication, should use a different technology. They can use key exchange to agree a secret key, and then apply symmetric message authentication. But even symmetric message authentication is not guaranteed to hide the message or the identity of the participants. To hide the message, it should be encrypted not just authenticated. To hide the identities of the parties, further mitigations are needed, and achieving full anonymity together with full authentication might be require sophisticated measures.

Nontheless, users might assume that plactic signature can hide messages or the signer's identity. To a small extent, the hash function does help.

However the sorted content of the bytes in a signature, which is mathematically the commutative image of the plactic monoid elements, would usually leak information about the message or the public key, since the content of the signature is simply a sum of the content of the hash of the message with the content of the private key.

A partial fix to this problem would be to re-define the hash function as the shuffle of the order of bytes of a fixed byte string. Under this hash function, the content of the hashes is always the same, so the content would seem not to reveal the message or the public key (or even the seed used to generate the chekcer c).

Another attack strategy would be use divisibility. The attack can compute a as the hash of the message. Upon obtaining a signature d, the attack tries to detect if a divides on the d on the left, meaning to detect if there exists b such that $d = ab$. This is a computational problem, the **divisibility problem**.

If the divisibility problem is difficult, then perhaps a shuffle-based hash would be effective in keeping a secret message hidden from an attacker with a signature.

In the plactic monoid, the divisibility problem might be easier than the division problem. For example, the erosion algorithm for division could be used to solve the divisibility problem, and will generally be produce an error as output when divisibility does not hold. Given d and a generated independently, so a does not divide d, it might, on average, be easy to determine that a does not divide d.

I have not tested the ability of erosion to detect non-divisibility. I expect it would somewhat depend on how the non-divisible a and d are generated. But I would also expect that would usually be much faster then division itself.

Therefore, I expect that message secrecy will not be viable for plactic signatures, as currently specified, or with a shuffling base hash. Message randomziation stamps, thought, might help provide message secrecy, but these have the cost of extra bandwidth.

For signer secrecy, the computational problem of detecting a **common factor** b between signature $d = ab$ and public key endpoint $e = bc$, could to detect if d is a signature under e for some secret message (whose hash a is completely unknown). Common factor detection in the plactic monoid might be much easier than division.

6.9 Linking signatures through determinism

Plactic signatures are deterministic in the sense that the same message and same private key always produces the same signature.

Such determinism introduces a linkability, which is another non-standard attack on signatures. If Bob signs the same message twice with the same private key, his signatures alone will reveal this fact, because the two signatures will be identical.

Some partial fixes to this non-standard attack are to stamp or randomize the message before signing, in a way that ensures the same message is never signed twice. This could work fine unless the verifier does not does not understand stamp and interpets the stamp as part of the message.

A Code size

More complicated cryptography is potentially more difficult to study. In particular, a security analysis might need more time for complicated algorithms, informally due to what some call a larger "attack surface". Quantifying how complicated an algorithm is may thus be worth trying, despite objective quantification being a well-known difficulty (and absolute quantification might be undecidable).

This section listed some simplisitic measurements for how complicated plactic signature algorithm is. The measures use various relatively standard software programming methods, such as the C language, the reference implementation, the word-count program `wc`, the C indentation program `indent`, the sizes of compiled object files, and also a bit of code golfing.

Similar measurements other post-quantum cryptography (PQC) algorithms might not yet be available. Other PQC algorithms might have reference implementations written with very different objectives. For example, perhaps other PQC implementations devote more code to efficiency than to interoperability, in order to demonstrate the practicality of the PQC algorithm. Code size might then measure the implementability of the other PQC algorithms, but might not be a fair measure of the intrinsic minimally interoperable description of the other PQC algoirthm. With such a lack of such measurement, it would not be inapproprate to compare the results here about plactic signatures to other PQC algorithms, so no detailed comparison is provided. Tentatively, at a high level, some of other the NIST PQC project Round 3 signature algorithms reference implementations seems to use about ten times as much code as plactic signatures. Code compression of these reference impelementations might yet reduce the code size down to the levels of plactic signatures.

Table 19 lists tsome C source files of reference implemntation (including header files). The list includes files specific to plactic signatures, but excludes files common to other signatures, such as random number generation and hashing). There are under 70 lines and under 2800 bytes.

The source files in Table 19 use an unconventional highly packed coding style. A packed style underrates how complicated that implemented algorithm is. The program `indent` can somewhat revert C code to a more standardized, conventional spacing style, with the results shown in Table 20. This expands the code size, but with fewer than 110 lines and 3000 bytes.

Compiled object file size, instead of source file size, is another way measure how complicated the algorithm is. Table 21 list object sizes when the compiler optimizes for small size. For comparison, similar object files for auxiliary files, hashing and random number generatin, used by the reference implementation are listed too.

File `sign-golf-mono.c`, listed in Table 22, is a preliminary effort of ap-

```
$--$  files="api.h lengths.h types.h plactic.h plactic-ref.c \
      sign-defs.c sign.c"
$--$  wc -lcL $files
   8  352   69 api.h
   8  238   44 lengths.h
   1   56   55 types.h
   2   82   62 plactic.h
  14  673   66 plactic-ref.c
  14  533   72 sign-defs.c
  21  825   69 sign.c
  68 2759   72 total
```

Table 19: Source code size (packed)

```
$--$  fmt='%4d%5d%5d%5d %s\n' ; \
      for file in $files; do
      printf "$fmt" $(indent < $file | wc -lwcL ) $file
      done ; \
      printf "$fmt" $(cat $files | indent | wc -lwcL) total
   8   58  373   78 api.h
  10   45  242   42 lengths.h
   2    9   56   30 types.h
   2   19   89   69 plactic.h
  28  177  780   72 plactic-ref.c
  13   52  522   72 sign-defs.c
  42  194  908   73 sign.c
 107  554 2972   78 total
```

Table 20: Source code size, after automated indentation

```
$--$  gcc -c -Os plactic-ref.c keccak.c rng-util.c sign.c \
      wc -c *.o
2784 keccak.o
2016 plactic-ref.o
1824 rng-util.o
3152 sign.o
9776 total
```

Table 21: Smaller object file sizes

plying **code golf**, meaning to make code small as possible. To be clear, code golfing forgoes many of the usual programming goals, including efficiency, source code clarity, and implementation security. The file `sign-golf-mono.c` implements the plactic monoid multiplication and multiplicative signatures, but does not implement the hash function or random number generation.

File `sign-golf-mono.c` was obtained by applying code compression to files `plactic-lowmem.c` and `sign.c`. Most spacing was removed. Various tricks with C operators and counters were used as part of an effort to decrease the file size (byte count). The content of files `lengths.h` and `sign-defs.c` were directly incorporated into `sign-golf-mono.c`, in order to make the file self-contained, a single monolithic file that has all the code specific to plactic signatures.

Beware that the code golf version drops the ability of the user choosing an arbitrary name as the seed value for hashing to obtain the checker value c, in order to avoid re-implementing `strlen`.

Preprocessing macros can be removed, and more conventional indentation of the code can be restored, by applying a command line like this:

```
gcc -E sign-golf-mono.c | grep -v ^# | indent | cat -s
```

Such formatting arguably provides fairer versions to compare under code golfing. Piping that output into a word count command line, such as `wc -w`, might also be a fairer way of code golf scoring than a simpler byte count or line count.

```
#include "keccak.h"
#include "rng.h"
#include "api.h"
#define S(a,b)(a^=b,b^=a,a^=b,1)
#define R(k,x)x[2]<x[k-1]&&x[0]<=x[k]&&S(x[1],x[(k+1)%3])
#define X(a,b)x(a##b,a,a##L,b,b##L)
#define H(t,m)FIPS202_SHAKE128(m,m##L,t,t##L)
u n[]=CRYPTO_ALGNAME;enum{kL=CRYPTO_SECRETKEYBYTES,
 eL=CRYPTO_PUBLICKEYBYTES,dL=CRYPTO_BYTES,bL=(dL<eL?dL:eL)/2,
 aL=dL-bL,cL=eL-bL,aeL=aL+eL,dcL=dL+cL,abL=dL,bcL=eL,nL=(sizeof n)-1};
void x(u*d,const u*a,int A,const u*b,int B){int i=A+B,j,k;for(;i--;)
 d[i]=(i<A)?a[i]:b[i-A];for(i=1;i++<A+B;)for(j=i-2;d[j+1]<d[j]&&j--;)
 for(k=2;k--;)for(;0<=j&&R(2-k,(d+j));j--);}
int crypto_sign_keypair(u*p,u*k){u*bc=p,*b=bc,*c=b+bL;
 if(k!=p)randombytes(k,kL);H(b,k),H(c,n);X(b,c);return 0;}
int crypto_sign(u*s,ll*sL,const u*m,ll mL,const u*k){
 u*ab=s,*a=ab,*b=a+aL;int i=0;H(a,m),H(b,k);X(a,b);
 for(;i<mL;i++)s[dL+i]=m[i];*sL=dL+mL;return 0;}
int crypto_sign_open(u*m,ll*mL,const u*s,ll sL,const u*p){
 u ae[aeL],*a=ae,dc[dcL],*c=dc+dL;const u*d=s,*e=p;int i=aeL;
 if(0>(sL-=dL))return-4;s+=dL;H(a,s),H(c,n);X(a,e),X(d,c);
 for(;i--;)if(ae[i]^dc[i])return-1;
 for(;++i<sL;)m[i]=s[i];*mL=i;return 0;}
```

Table 22: File `sign-golf-mono.c`

B Some timing results

Example runs of a timing program for plactic signatures are listed in Tables 23 and 24. The examples were run on a day-to-day personal computer.

```
$--$ gcc plactic-ref.c keccak.c rng-util.c sign.c timer.c \
     -o Timers/time-ref-O3 -O3

$--$ grep name /proc/cpuinfo | uniq
model name      : Intel(R) Core(TM) i5-8350U CPU @ 1.70GHz

$--$ Timers/time-ref-O3
Key pair generation
Average 5.218701e+05 cycles, relative deviation 0.038044
Signing, under same key, same message: 'Time me!'
Average 4.879321e+05 cycles, relative deviation 0.031624
Signing, under new keys, new random 200-byte messages
Average 5.066483e+05 cycles, relative deviation 0.040254
Verifying same signature, under same key, of same message: 'Time me!'
Average 1.471657e+06 cycles, relative deviation 0.061509
Verifying signatures, under new keys, of new random 200-byte messages
Average 1.461324e+06 cycles, relative deviation 0.056194
Sucessfully verified all 3457 out of 3457 signatures.
```

Table 23: A timing run of the reference implementation

Beware that no extra effort towards accurate benchmarking were applied in the results of Table 23 and Table 24.

Discrepancies from Table 23 and Table 24 have subsequently been found, by running the tests under more stringent conditions.

Battery power operation approximately triples the average cycle counts. Worse, the relative deviations are higher. Perhaps the system somehow reduces the cycle count when connected to external power, as a power management feature.

TurboBoost (over-clocking feature) being turned off approximately doubles the cycle count. This seems to contradicts the naive understanding

```
$--$ Timers/time-search-O3
Key pair generation
Average 4.985541e+05 cycles, relative deviation 0.089669, (excluding max)
Signing, under same key, same message: 'Time me!'
Average 4.588053e+05 cycles, relative deviation 0.021693, (excluding max)
Signing, under new keys, new random 200-byte messages
Average 4.729537e+05 cycles, relative deviation 0.026023, (excluding max)
Verifying same signature, under same key, of same message: 'Time me!'
Average 1.223879e+06 cycles, relative deviation 0.014959, (excluding max)
Verifying signatures, under new keys, of new random 200-byte messages
Average 1.228747e+06 cycles, relative deviation 0.018206, (excluding max)
Sucessfully verified all 3457 out of 3457 signatures.
$--$ Timers/time-prep-O3
Key pair generation
Average 5.028088e+05 cycles, relative deviation 0.123387, (excluding max)
Signing, under same key, same message: 'Time me!'
Average 4.421923e+05 cycles, relative deviation 0.057066, (excluding max)
Signing, under new keys, new random 200-byte messages
Average 4.695113e+05 cycles, relative deviation 0.052737, (excluding max)
Verifying same signature, under same key, of same message: 'Time me!'
Average 9.760128e+05 cycles, relative deviation 0.039104, (excluding max)
Verifying signatures, under new keys, of new random 200-byte messages
Average 1.003544e+06 cycles, relative deviation 0.046278, (excluding max)
Sucessfully verified all 3457 out of 3457 signatures.
$--$ Timers/time-lowmem-O3
Key pair generation
Average 1.939491e+06 cycles, relative deviation 0.017501, (excluding max)
Signing, under same key, same message: 'Time me!'
Average 1.929069e+06 cycles, relative deviation 0.016375, (excluding max)
Signing, under new keys, new random 200-byte messages
Average 1.942828e+06 cycles, relative deviation 0.019002, (excluding max)
Verifying same signature, under same key, of same message: 'Time me!'
Average 7.465280e+06 cycles, relative deviation 0.047148, (excluding max)
Verifying signatures, under new keys, of new random 200-byte messages
Average 7.427583e+06 cycles, relative deviation 0.038609, (excluding max)
Sucessfully verified all 3457 out of 3457 signatures.
$--$ Timers/time-con-O3
Key pair generation
Average 1.132362e+07 cycles, relative deviation 0.006755, (excluding max)
Signing, under same key, same message: 'Time me!'
Average 1.128513e+07 cycles, relative deviation 0.005047, (excluding max)
Signing, under new keys, new random 200-byte messages
Average 1.133913e+07 cycles, relative deviation 0.017428, (excluding max)
Verifying same signature, under same key, of same message: 'Time me!'
Average 4.999261e+07 cycles, relative deviation 0.007808, (excluding max)
Verifying signatures, under new keys, of new random 200-byte messages
Average 5.003160e+07 cycles, relative deviation 0.008686, (excluding max)
Sucessfully verified all 3457 out of 3457 signatures.
```

Table 24: Other timing results

that over-clocking only changes the time per cycle, not the number of cycles. Perhaps, TurboBoost does more than over-clocking?

Hyper-threading disabling seemed to have negligible effect on the cycle counts.

File `timer.c`, listed in Table 25, is the latest version of the timing program for plactic signatures that was used to generate these test results.

```c
/* Time key generation, signing and verifying */
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "rng.h"
#include "api.h"
int reps= 3456, cycles=1, all=0, test_no=5;
static ll ns (void) {
  struct timespec t;
  clock_gettime(CLOCK_REALTIME, &t);
  return t.tv_sec * (ll)1e9 + t.tv_nsec ;}
static ll cy(void) {
  unsigned int lo, hi;
  __asm__  __volatile__ ("xorl %%eax,%%eax \n        cpuid"
               ::: "%rax", "%rbx", "%rcx", "%rdx");
  __asm__ __volatile__ ( "rdtsc" : "=a" (lo), "=d" (hi));
  return (unsigned long long)hi << 32 | lo;}
static ll tm (void){return cycles? cy(): ns();}
static double sqrt (double x){double y=x; int i=200;
  while(i--) y = (y+x/y)/2;
  return y;}
static void report_stats (double sum, double sum2, double top){
  double avg=(sum-top)/(reps-1),
    dev=sqrt((sum2-top*top)/(reps-1)-avg*avg)/avg;
  printf ("Average %e %s, relative deviation %f, (excl. max)\n",
         avg, (cycles? "cycles": "nanoseconds"), dev);}
#define TIME(CODE,...) for(test_no,t=s2=s=0, i=reps; i--; ){\
    __VA_ARGS__; \
    if (!all) fprintf(stderr,"%d,""%d \r",test_no,i+1); \
    o=tm(); CODE;   n=tm(); \
    d = n-o; s += d; s2 += d*d; t=(d>t)?d:t; \
    if (all) fprintf(stderr,"%g %s",d, i?"":"\n");} \
  test_no--; if (!all) fprintf(stderr,"   \r"); \
  report_stats(s,s2,t);
static void time_sign_keypair (void){
  u pk[CRYPTO_PUBLICKEYBYTES], sk[CRYPTO_SECRETKEYBYTES];
  double o,n,d,s,s2,t; int i;
  printf ("Key pair generation\n");
  TIME(crypto_sign_keypair(pk,sk),);}
static void time_sign (void) {
  u pk[CRYPTO_PUBLICKEYBYTES], sk[CRYPTO_SECRETKEYBYTES];
  u m[200]="Time me!"; ll mlen=strlen((char *)m), smlen;
  u sm[CRYPTO_BYTES+sizeof(m)];
  double o,n,d,s,s2,t;  int i;
  printf("Signing, under same key, same message: '%s'\n",m);
  crypto_sign_keypair(pk,sk);
  TIME(crypto_sign(sm, &smlen, m, mlen, sk),);
  mlen=sizeof(m);
  printf("Signing, under new keys, new random %llu-byte messages\n",mlen);
  TIME(crypto_sign(sm, &smlen, m, mlen, sk),
      (crypto_sign_keypair(pk,sk),randombytes(m,mlen)));}
static void time_sign_open (void){
  u pk[CRYPTO_PUBLICKEYBYTES], sk[CRYPTO_SECRETKEYBYTES];
  u m[200]="Time me!"; ll mlen=strlen((char *)m), smlen;
  u sm[CRYPTO_BYTES+sizeof(m)];
  double o,n,d,s,s2,t;   int i; int fail=0;
  crypto_sign_keypair(pk,sk);
  crypto_sign(sm, &smlen, m, mlen, sk);
  printf("Verifying same signature, under same key, of same message: '%s'\n",m);
  TIME(fail+=!!crypto_sign_open(m, &mlen, sm, smlen, pk),);
  mlen=sizeof(m);
  printf("Verifying signatures, under new keys, of new random %llu-byte messages\n",mlen);
  TIME(fail+=!!crypto_sign_open(m, &mlen, sm, smlen, pk),
      (crypto_sign_keypair(pk,sk),
       randombytes(m,mlen),
       crypto_sign(sm, &smlen, m, mlen, sk)));
  if(fail){
printf("FAILURES: %d out of %d tries !!!!\n",fail, 2*reps);
fprintf(stderr,"\a\a\a\a");}
  else printf("Sucessfully verified all %d out of %d signatures.\n", reps+1, reps+1);}
int main (int c, char**a){
  if (2<=c) sscanf(a[1],"%d",&reps);
  if (3<=c && 'n'==*a[2]) cycles=0;
  if (4<=c) all=1;
  time_sign_keypair(); time_sign(); time_sign_open();}
```

Table 25: File `timer.c`

C Experimental utilities

This section lists code for some experimental command-line utilities. The utilities can generate key pairs, sign messages, and verify and open signed messages. The utilities can run on some Linux systems.

C.1 A simplistic utility

File `ps-util.c` in Table 26 combines some standard C libraries with the plactic signature library. The message to be signed is supplied as a command-line argument. If the first argument is the name of a file, then the utility will treat the named file as the message. Otherwise (if the argument is not the name of a file), the argument is taken as the message to be signed. In a shell environment, supplying a message with space as command-line argument can be achieved by surrounding the argument with quotation marks.

File `ps-util-help.c` in Table 27 describes the user interface of the simplistic C utility. The terse instructions are addressed to a user already quite familiar signatures (especially plactic signatures) and already quite familar with command-line utilities in general. The instructions intend to remind the user about the utility's interface map to the standard digital signature interface. The instruction are too terse to teach what this interface to a digital signature means. The terms **lock** and **key** are used instead of the usual **public key** and **secret key** (or **verification key** and **signing key**) to squeeze as much information into a single screen of text.

The utility's interface intends for the user for to pipe the secret key into and out of an encryption program, and deliberately avoids reading a secret key directly from a file. For brevity, the one-line example of usage in the help message pipes the secret key directly from key generation mode to the signing mode. The secret key would then be lost forever.

Verification expects that the public key is stored in the file system, thus presuming the file system can be trusted. If the public key was originally conveyed to the user within a certificate, then the utility expects the user securely extract the public key from the certificate and store it in a file. This clearly does account for expiry or revocation of the public key, so a user might want to do certificate validation as an extra step before every signature verification.

```
#include <stdio.h>
#include <string.h>
#include "api.h"
#include "ps-util-help.c"
enum{MAX_LEN=10000000}; u buffer[MAX_LEN];
int key(void) {
  u pk[CRYPTO_PUBLICKEYBYTES], sk[CRYPTO_SECRETKEYBYTES];
  crypto_sign_keypair(pk,sk);
  fwrite(pk,1,CRYPTO_PUBLICKEYBYTES,stderr);
  fwrite(sk,1,CRYPTO_SECRETKEYBYTES,stdout); return 0;}
int sig(char *msg) {
  u sk[CRYPTO_SECRETKEYBYTES], *sig=buffer;
  ll sklen,slen,mlen;
  if(0 == memcmp(msg,"--help",6)) {help(); return 6;}
  sklen=fread(sk,1,CRYPTO_SECRETKEYBYTES,stdin);
  if (sklen != CRYPTO_SECRETKEYBYTES )
    return err(2,"Bad secret key");
  if (fopen(msg,"r")) {
    mlen=fread(sig+CRYPTO_BYTES,1,MAX_LEN-CRYPTO_BYTES,fopen(msg,"r"));
    msg=sig+CRYPTO_BYTES;
  } else mlen=strlen(msg);
  crypto_sign(sig, &slen, (u*)msg, mlen, sk);
  fwrite(sig,1,slen,stdout); return 0;}
int ver(char *pk_filename) {
  u pk[CRYPTO_PUBLICKEYBYTES], *sig=buffer, *msg=sig+CRYPTO_BYTES;
  ll pklen,mlen,slen;
  if ( fopen(pk_filename, "r")) {
    pklen=fread(pk,1,CRYPTO_PUBLICKEYBYTES,fopen(pk_filename,"r"));
    if (CRYPTO_PUBLICKEYBYTES==pklen) {
      slen=fread(sig,1,MAX_LEN,stdin);
      if (slen >= CRYPTO_BYTES) {
        if(0 == crypto_sign_open(msg, &mlen, sig, slen, pk)){
          fwrite(msg,1,mlen,stdout); return 0;}
        else return err(1,"Bad signature");}
      else return err(3,"Bad signature");}
    else return err(4,"Bad public key");}
  else return err(5,"Bad public key");}
int main (int c, char **a){
  return 1==c?key(): 2==c?sig(a[1]): 3==c?ver(a[1]): help();}
```

Table 26: File `ps-util.c` (key generation, signing, verifying)

```
int help (void){ printf(
  "Usage summary:\n"
  " ./psu [message-or-filename [open]]\n"
  " task |args| arg1      | stdin     | stdout    | stderr\n"
  " -----+----+-----------+-----------+-----------+-------\n"
  " pair | 0  |           |           | key       | lock\n"
  " sign | 1  | 'message' | key       | signature |\n"
  " open | 2  | lock file | signature | message   | alert\n"
  " Example (artificial, single signature only):\n"
  " ./psu 2>pk|./psu 'Hello World'|(sleep 0.1;./psu pk open)\n"
  "Plactic signature experimental utility. Dan Brown, BlackBerry.\n"
  ); return 4;}
int err(int r, char*e){
  fprintf(stderr, "psu: %s.  Try ./psu --help\n", e); return r; }
```

Table 27: File `ps-util-help.c` (help function)

C.2 A more flexible utility (unfinished)

A previous version of this report included an implementation of plactic signature command-line with a very flexible interface.

Future versions of this report might include an improved version of that flexible utility (perhaps supporting compression).

D Layered signatures

A signer can apply more than one signature to a message by using different signature algorithms, as a defense against one of the signature algorithms being weak.

The NIST API of a signed message rather than a signature, strongly encourages that the application of multiple signature algorithms should be done by nesting signed message within other signed messages. In effect, the inner signatures of the nest are signed by the outer algorithms.

Intuitively, such nesting of signatures should be secure, provided that at least one of the signature algorithms is secure. (Presumably a formal proof of this already exists.)

A plactic signature public key is listed in Table 28 with standard base64 encoding.

```
$--$ base64 public # plactic signature public key
```

```
9/b179zb+crw/sjp/cDP6vy9yOn6s8Lj+bK92t/8n7nS2/aen8/Y7Jacvs/Tipuxw9H2iJaqtNDs
hZKfrbPg8vn/couboqzNze/xbYeUoau3yefp+muDipiotsjZ3uvranSFkae1wdXX2en8WGh/ipiO
wMHL1+jrVGd6fZSutbrA0uHpUmF4eoCVsLe8yNzh+FFgbnN+i6WuucHG1ff//OlTWmV4ipmipq+9
xef2+OBOVWJ1h5WXn5+2vuLl9DdGUFlug4WNmJ6uutDU3Ps1REtXbXWAhoqcq6+8wsfv/S5CSVZg
b3KDhpGjqbe+xeDs7Cw6SFVcZXGBhI+Wn6usu97i4/QrOUJMW2Bsf4ONlJidqrjMOOLz9yctPEVY
WWdOeoGHj5ymp7TA4fHx9SYsNkRNWGNlbn2Ah5mbpbCyu7zp7f4jKzE/RlBiZGdxeoSHipesrLOz
OeXl5+/+ICcuOEVKW2FmaXKAhYaUoKutrrzU2drg6u8eJCk1Q0dRVltkZXV1eYSFn5+ktrnDxcnb
4+v2HR4oMj5FSVVXYmNuc3R5e3+YorO3ucHIOtfi9PQWGSMxNzpERkxRX2lucHh5e3uVnKeytcPD
y9Lp7/D6FBggLDI3OUJFRUlaam9vc3Z5hpCdo6e3ubnGzdXW7O75ExcdJSOzNzo/QENGXmRlZm1y
eX6ElqWorrC3xsbOOdjkERQbJCssMDI4PkJEUVtbW2BqcnWChZWZnq6uv8LDzM/R3/HODxAXHROd
JiksPD5CQk9TV11ia2uAhImVmqqqsLq9vcPNOd7q/g4OFBQcHCMnKjE7QEBDQ0xQVWRmcXSFhoiQ
pKeusrvAxsfT2Ors+/7+BwOPExUWHiQkJSo8PkJCROhNW2FhY2t6goOGkJSioqytu7/R09zh4fr/
BgsMEBMVGRsbJCQvOTs7RkdIUFVbX2ZrdnyAhIWWoKGksra5yszNO9rl5+3y/AUHCgoQFBYYGRod
IywwOEVFRkpRUVpcXG9wf4GEjI2PmZ2utrm6xcXTO9XW5ufyAwQGBw8REhIXGBgdHSYsNTo8QORE
RUxQXWNlZW52e4WGkpOXnJ6wur3HztTV5erq8fsCAwMGBwoLDREREREscHiAqLjQOOOBAQkZISU9R
V2drb3aMjJKanJ2etLbAz9DZ3ufn9vYBAgICAwYHCwsNDxQUFBgYGyMvMDU7QUJDRUZGSUpSWmlv
dnZ2d3h9naS2ucPP19fk5fT19/f7/P39AAAAAAEBAQUKCgoLDBESExMUFBgcJCksLDU/QkVGROtL
Tk9QUlVkbHB2dnmCiJKXl5qfpbe3usPPONbi6/H+/w==
```


Table 28: Plactic signature public key, base64

Similarly, a Dilithium2 public key is listed in Table 29 with standard base64 encoding, which was generated using a command-line interface utility downloaded from `libpqcrypto.org`.

```
$--$ base64 dilith-pk2 # Dilithium2 public key
```

```
lU3N5aD/xlzXpS/9hKWppZkIjXeByzVjk41SWn19zI9Y/q41ui+rHIVfMcsSjY/dNgZZmgSn3dil
6unTyOxz27P1OZoY485T4xqt2xQTXgAFBqpGOE58IO9EF25zy4jXyFqbxJt86kfSCiUH/62dYnCK
owvH9od6hRmORaZAkgcOFZ2H5SvSOPn3BVjzDD46OtSp2VNuFPN5RjCQI7rE7qgU436640c+4luT
7jlul2tdAUJ80ajd7QB+F4uJ99B8nhmKHfqLLQ7ONsyhL5G5eYDxQ0UtJNCre63TwWY/pV7ydyqd
8+gwyT2YX2t8CTZ9Ny2OX+mUEaKjUhvEp+S4salxt54Bq4E14/C1pPqd7FWB72jRFVBaogoJAbsW
hvYymUE6CpdVDEGzVEvCATB1VCSAy2rvVBYwi+iF74LBGC2sLfmo8pKJH+xslajPt8iNkgEh+SHM
MvFAcwIutNdxbAVmOi2NmiOO4uVW266LvxBk99xwjEgrhWTiNBlkUILcieINlWBwWZ9wr5BHslf+
RWTh8f+mHJCroYs6MoGoh15T/KHjqR1azfnxu3tbGSv7xqHzKUkAIn8OUW98p94bjdKLRhNUQFub
Q0xrFvMRWq5T+qsnUEt8dKF6ViRnWgpTRrWqiEphRvJZwkG4uYmACAJ+NUNj18KW/IOnwn892f3K
PhZSBDkhPjOI/SRiPkPIBmye98tgipukUQP7ggZ/8iE9pLq5Y8ymjcWiTBrr+Cp457Y/dfhPf4C+
pByJaIrL77jeOQ84rQ1QJIATonT9ysJXC3UxnYOP5bE87NQytz5uJ6Llw9ESDq9ZXqkhPq8RnnqG
zOSiEypn9Ieb6K2ZDht4GpD27id5O34OPjp1HKus+S/QBDTMyvXifz5psK1LKJkAo1FEIHxFA9Gq
SNG7cgGT1lK8wHc+RoLcJoXxoVTjeaZJdqFrrrZJsueHyKKCNeOM9vsN4Uil6YIYiTWSeEAo2g3Y
8jLFuZO7Yor5kRArJy2LaDJvlXpEey/9PjTrB3oQC5PWBtjOhOX6AK99Wgu5PyJ2Lb8N/FlDldc/
l7Irdsohq+IxGedFJTXXlA6+TVOAmaBYlcDcMxeHmHcrWEUF3oOykB1V97KNcKR+3cpx92SLfIIl
dNOfAvJVp/zJwao5GQAsGacO/CyT/UPGit+7bR2c/ZKN1/xEy9tm1MsP9hk2kOO6LMow948bKe+4
aejO84OpUauwPZ6j/5Xh1hqH7nouuPYtI1hEG5KUaGT7pUi5xUsLpwRCZT5HCCrlho1ha/KbfbON
ABxl/iiSrsnG9ec3XRMYpkkfM4pO+V2/TmeNEPsbaN71pjHBcbGkRwrfsWgaGL8l2OX++dkOSOPB
s8xD699pY/vIegZWtYmI4KPMa4q5soy/PXct4MO7RvScLjVb79qEQIYp2X6Q943lIxHci9xFSXeA
W+NSjOQxcuXfpueyW13gF7OWzyI1gUAj8A+eQqr65tGpJYpVJSluJ28fLCiTuKZiPguNUQn+1YYu
7GdypxiuuirJRURZVy+rOdtcAzvEnO5BYf3DWHx/htR6uXNOfH4v3B1CUpM=
```


Table 29: Dilithium public key, base64

A doubly-nested signed message is listed in Table 30 with base64 encoding. The outer signature algorithm is Dilithium2; the inner algorithm is plactic signature.

Verification and opening of the doubly-nested signature can be achieved by running the two utilities in a pipeline, as shown Table 31.

```
$--$ base64 hello2.signed.dilithium # Nested signed message
```


```
3wtkvEDW1OxMzUjAXQbcL+Q3Q35djBzFyOSNttto85pIrYm/Bv1uOMSRIhopsRA5CpH2gbYeaEKv
c/Q6ch6H43EnSXOQNE3Kf6f8YfWDVj91VidpdXUXRHdBwXULXLd4wOcEfFFK4KQESgQLRLDDYQse
tg9FoVMUn6/8+8rIeaCcB+/J0KIM7JBOEOl1Irgkgqjf8/s9Jc/58JybjfnZmBzQYQMduFJdLNUL
xm5c7HMGrGbAcFOeN+VkOlarY9BeOOyFAhJcI8EuKxSxQm4mwWQS5MbWb2wh2W6S7QGP1e5YuM9i
/CytbnnrlpgDqzawj2JaJ6ZuJ3h2griIR4p4Xs7B5Bx5Oo2oO+iU0UwZ8V4l71WWMNEGEzTkpEFO
xrcaequYyceafBTSaSOdPKfncXqQ5MlLnnjKLKrMwnOkOkfDiveseGIt5dNSVH8p+peOjSzayONZ
/6P12HWOXOc1F2JwIV1KlaNUa7/49YtWhiVoWO179L1HdLrjpTvPvX3Y20wrM7gy8WtqvKZnLuLl
Ir6OF+p4N7/B8xv/oKYOalmmB2R6d7+k90v0IyM+MjQSgCUBs9CRCh2zmPqLqUC5kpIrNNQKR63O
bPHIFr57nrznWf+m8m8IU0sstTh+F+V3C3AIBIdB26G2Hf8WsmohFBDKBKEjf9fvd+Ur1b1SKtCu
/eyafNzLx13JgZUcdYC5CJg8+2uzvxK9mtGrHWAtANZ+gcwUyL3QdQ9dNg/J85CfCu+v8EjZwo4t
Jd9Y9o0eg0swuMQpUJ4C7jt8wMN4VRVZVULb0LENjXRZQ5cF59RxTvWPw/w4E1VeUeXu/3D6D/za
cLMN/fISLS9lCpamYFJh7xf2nqUWW2rDNDdQcy6FF1J4ESoUoEJbFhtksY++eOmLuBnLnrGnqzy6
yvFNVd5U+5OI4IlMtHxIy7HwaQifDGM0N0Y2tchUi3bvDvbujaSWRmpJ4ckQnhFRURQVMHgdgdeF
jDzIyKncrM3NpuJwJg486UGUHjtBlxR0iXqlqFfdJLxOwiZokIEG18N2OmwqH5nzkUZMGcSUluRj
SD40c4sxt1GcRsFpAvGpD5/8FO5P4V+mpWJaWQHPEvBfLzj1gvzyiC6PC/EoDo/aQUogpPnXHYLd
AHKAHgeKShOlNId4rovntyoOr+IGDAK/IIHj+jiux2BiDCYQ8981/2s79rdjCTt1r1P5JqltkiWa
kaQZqdZsasODBv1eOMmvV/t6sAxGzmC6yMKKLHOWUGcJlt8W+W3JEfgcgdkKBa5Q36HOGUrQ4dog
rsMskckSD6/48IqaoToYqrJdiNuFd4IQJAjwtgxxywuekeMZKW+n8na+EL0K0Rq0ez+7kDDNCdNP
gZOT2EK7BLdLQhUtU9Ffk/wwyfhMO8+0Pp+u9OnjDt7u4OGN8N0I7/X1Xl8hakWnVoRrGrWm1q7X
6HpYTLrGpGCT4jQpS3gDgjfuNUlikzoZPpLhhwsIuIA83ZrTrV67LLfLqYNyOyi3q3686kqEqECI
nkHmGmRGLbDVAurVkmItBwHIEID2SBeKdPJL48A0iKZ/afogwIj/i3M8p8tzIwkqkqCBjX7X6GTz
7DTPkstCuiz133QHTsOJsZ4Yc1xDyDUrpQdTeoVxMhwnUVlwlQVYPSfVc/uXHIDJpu2L2r4R/+zv
z8Yzej6j8F/n/3U4yByCzCUB1Q9QbXdkdkdQlLJFKf8G6W+QydW+XeOOIGMEMqERmBeBz+4f7v5l
KnGnEtWuZe1auxtQvgGTmvmnmWmcVsZpavwaxK+mKwi7ghXCZx98awiRhxBKbnHmFgjNdtNsQdtS
si1VOGCFAykyfiLjkOF9Gt4GZKlAlvrerPDNH/JbHr8kOaeTc+vdPN7NkS4M48LEFphsgUd6qaWX
Ogf1cVDQpx9++uUuJfNSkAhpiJCGnSDWCYfYD4r9OGeNc9aZW/m4lUay9yV7XnZ1Zldx/MDJDzOF
t1F4WIABCRgSzo/i/FMhoRgSBX+08keFd7x3x5WNnNvIniQcSsK8ke4f6YeP+vSoHRPjMjE5qa+R
+p2wXw37fV112lW1RChJhMlb3bnVv9SUTU2zEOI8IRyxmhGrQjPjNDOUcUgah1D/RPdPK4AXA3h/
hDJHKMPuZ+t+NtZCZi33ZBlSljB70rInNzIgJQMIj6XuWNJlY142b6/c88pkh+V3WE4lgVYSkE2z
2DQ/lMhGiTXlEVMesLxlx1s9fv3h17RIP4v0ql716FVKUTYVZdlGzGrEsmTdSNalZsVlVitJF5J0
p5KQJgl6G6a3YTHCUyM87juWvmPmrpfuegoQHAHBIAglhlAcsg0i27PB4hosOIq9otjTkIAJCZL+
YOkPOLZ3b3tkSJqApCXVTVLdCnJ+IOenmnGqGYHssssuessEu0yr6z253imbq7G5g7A6B6uar6n6
mj7vKPSOYlR8RcXvY+A8BlJr3rjmvjsAvAM5JCk9klSADQXYHpi6ganSAZsbsMdx8RsXdL8C9ytV
7iPkPtgH73zwjMhYhYyuti9r+4dNgd0UTWYMZMYZPTPQM+PxOBqPGT5OXGB8hpqhuMrLzOT6HCMk
RlRidpemqLLV2f4qK19sfYGEi5635ebv+h0kKD1xdnd6gZGrz9js7/P8AAAAAAAAAAAAAAAAAAAA
AAAAAAAPHSs8CBgCBUwGwUAKsEQoEQglCKCMIrAACKADAYQSFMFAEAYn4ocSv/qyCf705+T54ezY
69fY1dfC1Pq+0/S50OC3ytqzwdnp/ZS70+X3kbnO4vH7kLHM1O/6/Y+uydHg5O+MnsDQ2ODs/4KW
mM/R2Ojyf5WXwszU5PF5jZS1y9Pj8Pxqh42yvNDe7vFnf4KvtMHZ6ez6/19+fpaoqcLf3/f7XWV0
jJeku9Tc8/P/TVhtd3yetMzY3fL7SVRsdHmLobS5wtrtR1Brb3eEk6CusMza4PxBTmRlb4KOlKis
urrC9DRMYWFodomSlaO4uMHmMEhXX2NxeYqKoK+1utzq9ixGTVRebXB4fJ+fs7PDytsnRUxTVl5p
c3mPmaays8HQ5vz+Iz9JTk5ZZW5zi42fprC+w+Ln/SI4Qk1NVFtscYmKm56rvb2+z+fp/Bw2QEpL
UlZla4CBmZygtbW5ucDZ2tr6GzQ6QElJU1VmeHuKjpqbra2xsb3Cwt/tGiMyPOZHT1NYX3eAgoSS
nKOjpqy3vdLj8/kXIjA6REZOUVRcam6AgYqWlp+iqbO2x9rq7/kVICktNUNGS1BSWGVoe3uGiIqb
n6iOvdjb6fATHSQrLzpARktOUmRnaHmChIeSmJ+qtcPP4On19vwRHCMmKTg7QEhLS1VcZW1yeYGP
kZShp7PByNPn7Oz2DxQeIScqMDA8Rkl0UVpbYXR/g4WImJmlvsDR2d7p7/L5+g4RHR4kJyguNjc4
OkJOWl9hc3h7fYCLi6uwwcjNz9DV6+sNEBgdICMnKywtLTI5RlJWWGJudXp6foONoq60tbrJzdnp
8f8JDRcYHSIlKCgsLCw2PUhTV1xlbnJ4eoCKip+frre8yMvX4eH4CAwVFRYgIiUmJycrMDxERVBV
WVtjbW91hYeKlJWXub3Bxtfc3OL7BwkUFBUYGBoeHiYpLjVAQERGSVZgYGdscXF0g4WKkaavtrq7
vsjr/wYIDRIUFhYWGR0jJCcrLD9AQkZHTFVgYmRnboGEhoaVo66vsLLHOODi7O/8BAYIDhATExQW
GR0jIyUoMTU4QkNFTVhZW2VpdXV/hIeWnp+rq7O3vNLV5fQDBAcHCw4PEBEWGBsdHSYrLTI6PkRF
SlBWYWJkbnJzeYCDh4+YnKystMDF3uns+wIDAwYICA4OEBERFBccICQkKSw3P0JCSUxRUVhbW2Nl
bnR6fZSYmZ2srbO+wsPQ5fcBAQICBAcKCgoLEBETFBQdIyQqMjM3Nzo+QEBCRUZGSU9XW2lqe4GN
j5yfpqepqri8xdTU6Pb/AAAAAAMFBgcHCg8QERERFRYXGRscHR0kLDE5OzxDQ0NFVVpbX295eXuA
hYWGh5egp6eurrm5u8PFyczN0dLZ5+1IZWxsbyBXb3JsZAo=
```


Table 30: Nested signature, outer layer is Dilithium2, inner plactic

```
$--$ pq-open-dilithium2 < hello2.signed.dilithium 4<dilith-pk | ./psu  public open
Hello World
```

Table 31: Verification and opening of the nested signed message

Both Dilithium2 and plactic signature place the signature as a prefix at the beginning of the signed message.

E Tableau compression (to be completed)

In some cases, sending a byte might cost approximately as much as running a 1000 cycles on a device. Therefore, it might be worth compressing signatures, and public keys. In the case of plactic signatures, the canonical representation, row readings of tableaus, are highly redundant.

An ideal compression algorithm might be capable a 1000-byte row reading to less than 500 bytes.

The ad hoc proof-of-concept compression algorithm, implemented below, seems ready to compress 1000-byte tableaus to an average of approximately 570-ish bytes, because it outputs 1520-ish octal characters (0-7). The compressed lengths vary with the tableau, which does not fit well the NIST API, so padding might be necessary to meet a requirement for fixed lengths.

```
/* Ad hoc octal-based tableau (de)compression */

/* This works... Compressed a 1000-byte tableau down to 1500-ish octal
   digits (0-7), which on is 570-ish bytes.  Pretty good for such an
   ad hoc implementation. */

#include <stdio.h>
#define max(a,b) ((a)>(b)?(a):(b))
enum {rowmark=10};
typedef unsigned char u;

// solo
void allocate_tableau(u**t,u*T,int l){int i;
  for (t[0]=T, i=1; i<256; i++)
    t[i] = t[i-1] + l/i ; }

// pair of near-opposites
void set_tableau(u**t,int*r,u*s,int l){ int i,j,k;
  for (i=k=0; k<l-1; k++)
    i+=s[k]>s[k+1];
  for (j=k=0; t[i][j]=s[k], r[i]++, k<l-1; k++)
    if (s[k] > s[k+1]) i--, j=0; else j++; }
void put_tableau(u**t,int*r){ int i,j;
  for(i=255;0<=i;i--)
    for(j=0; j<r[i]; j++)
      putchar(t[i][j]);}

u min(u**t, int i, int j){return
    max(j>0? t[i  ][j-1]  : 0,
        i>0? t[i-1][j  ]+1: 0);}

// pair of near-opposites
int diff_tableau(int*d, u**t, int*r) {int i,j,k;
  for (k=i=0; r[i]>0; i++, d[k++]=-1)
    for (j=0; j<r[i]; j++)
      d[k++] = t[i][j] - min(t,i,j);
  d[k++]=-1;
  return k;}
void sum_tableau(u**t, int*r, int*d) {int i,j,k;
  for (k=i=0; d[k]>=0; i++, k++)
    for (j=0; d[k]>=0; j++, r[i]++)
      t[i][j] = d[k++] + min(t,i,j);}
```

```
// pair of near-opposites
void mark_rows (int*d, int k) { int i;
  for (i=0; i<k; i++) {
    if (d[i] >= rowmark) d[i]++;
    if (d[i] == -1)      d[i]=rowmark;}}
int unmark_rows (int *d, int k) { int i,l;
  for (l=0,i=0; i<k; i++) {
    if (d[i] == rowmark) d[i]=-1, l++;
    if (d[i] >= rowmark) d[i]--; }
  return l-1;}

// pair of near-opposites
void putoctal (int o) { printf("%o", o); }
int  getoctal (void)  { int o;
  do {
    o=getchar();
    if (EOF==o) break; }
  while (o<'0' || o>'7'); // allow for newlines, comments???
  return o-'0'; }

// pair of near-opposites
void encode_octal (int *d) { int i,k;
  for (i=0; i<k ; i++) {
    if (d[i]<06)
      putoctal(d[i]);
    else {
      d[i] -= 06;
      if (d[i] < 016)
        putoctal(060 + d[i]);
      else {
        d[i] -= 016 ;
        if (d[i] < 0160)
          putoctal( 07600 + d[i]);
        else {
          d[i] -= 0160;
          if (d[i]< 0160)
            putoctal(077600 + d[i]);
          else {
            d[i] -= 0160;
            putoctal(0777600 + d[i]);
          }
        }
      }
    }
  }
}
int decode_octal (int*d) {int i,j;
  for (j=i=0;  ; i++) {
    d[i]=getoctal();
    if (d[i]<0)
      break;
    if (d[i]<06)
      continue;
    else {
      d[i] *= 010;
      d[i] += getoctal();
      if (d[i] < 076){
        d[i] -= 060;
        d[i] +=  06;  }
      else {
        d[i] *= 010 ;
        d[i] += getoctal();
        d[i] *= 010;
        d[i] += getoctal();
        if (d[i] < 07760) {
          d[i] -= 07600;
          d[i] += 06 + 016;
        }
        else {
          d[i] *= 010 ;
          d[i] += getoctal();
          if (d[i] < 077760) {
            d[i] -= 077600;
            d[i] += 06 + 016 + 0160;
          }
          else {
```

```
            d[i] *= 010;
            d[i] += getoctal();
            d[i] -= 0777600;
            d[i] += 06 + 016 + 0160 + 0160;
          }
        }
      }
    }
  }
  return i;
}

void to_octal (void) {
  int i, j, k, l, r[256]={0}, d[2000];
  u s[1000], T[6125], *t[256]={T};
  allocate_tableau(t,T,1000);
  l = fread(s,1,1000,stdin);
  set_tableau(t,r,s,l);
  k=diff_tableau(d,t,r);
  mark_rows(d,k);
  encode_octal(d); }
void from_octal(void) {
  int i, j, k, l, d[2000], r[256]={0};
  u s[1000], T[6125], *t[256]={T};
  allocate_tableau(t,T,1000);
  k = decode_octal(d);
  l = unmark_rows(d,k);
  sum_tableau(t,r,d);
  put_tableau(t,r); }

int main (int argc, char**args )
{
  if ((argc-1) > 0) from_octal();  else to_octal();
  return 0;
}
```

F Generality of multiplicative signatures

Multiplicative signatures are arguably quite general. To informally illustrate this generality, consider ECDSA.

An ECDSA signature of the form $[R, s]$ is valid for message h and public key $Q = uG$ if

$$hG = sR - rQ \tag{12}$$

where r is a conversion of elliptic curve point R to an integer. (Strictly, an ECDSA signature is $[r, s]$, but the point R can be recovered from r in a few trials.) Let:

$$[a, b, c, d, e] = [h, 1/u, Q, [R, s], G]. \tag{13}$$

Reconstruct multiplication operations acting on variables a, b, c, d, e such that $Q = uG$ is equivalent to $e = bc$, while ae represents hG and dc represents $sR - rQ$.

To get a full semigroup, add an artificial zero element 0 in addition to those of the forms a, b, c, d, e. Then define all other multiplication to take the value 0. In other words, define multiplication as the operations matching the ECDSA operations as explained in the previous paragraph, and 0 otherwise. Associativity of this multiplication is ensured by the nature of the verification equation, or by the product of any other three elements being 0.

In the case of ECDSA, the value of e, representing G, is chosen before the value c, representing Q. This situation corresponds to the secret key b being an invertible element of the semigroup. In other semigroups, such as the plactic monoid, the secret keys are not invertible, so value c must be chosen before e. In ECDSA, the checker c is signer-specifier, while the endpoint e is system-wide, but that is only possible for multiplicative signatures in which the secret keys b are easily invertible.

A signature scheme is **separable** if the verification consists comparing two data values, and the public key is effectively two values, one determined by the other via an efficient trapdoor. For example, ECDSA is separable, and multiplicative signature are separable. It seems that several separable signature schemes can be considered instances of multiplicative signatures.