Plactic signatures

Daniel R. L. Brown[*]

December 16, 2021

Abstract

Plactic signatures use the plactic monoid (semistandard tableaus with Knuth's associative multiplication) and full-domain hashing (SHAKE).

1 Introduction

Plactic signatures instantiate multiplicative signatures (see Table 1, §2, and [RS93]), 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	Appendix of signed matter
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 [Bro21] or Wikipedia.

2 Multiplicative signatures

This section describes **multiplicative** signatures, which are summarized in Table 1. Rabi and Sherman [RS93] 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 with a mulitplication operaion. In more detail, 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.

This report focuses on one semigroup: Knuth's **plactic monoid**. An introduction (for cryptographers) to the plactic monoid may be found in [Bro21]. A general introductin to the plactic monoid may be found on Wikipedia.

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 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 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: concatenating all the rows (row readings) of the two tableaus; then applying the Robinson–Schensted to the resulting concatenated sequence to obtain a semistandard tableau; and then taking the row reading as the canonical representation.

2.2 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 the lock symbol often indicates several layers of security (for example, in web browsers, a lock symbol typically indicates successful verification of digital signature with a 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 PKI distributes some certificates manually as **root certificates**, and then transfers trust to other certificates using digital signatures (which could be plactic signatures).

2.3 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

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

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 $[a, d]$ is an invalid signature. To this end, we can call ae the **endmatter** and dc the **signcheck**, regardless of whether $ae = dc$ or $ae \neq dc$. In other words, a multiplicative signature is valid if and only if the endmatter equals the signcheck.

2.4 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).

In the reference implementation of plactic signatures, the value b, a tableau, will be computed as the hash of a more convenient, shorter seed key. In this case, the secret seed key can then be considered the secret key, while the tableau b is an intermediate temporary secret that is only computed during the signing and key generation procedures. But, for this section describing plactic signatures at a higher level, it suffices to consider b itself as the secret key. Furthermore, for a security analysis, the attacker can win by finding b, without even trying to find the seed key which is hashed to obtain b.

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.)

2.5 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 $[c, e]$.

2.6 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, 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 (pseduo)random); for c, the SHAKE input is a fixed byte string.

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

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

Notation	Name	Typically:
a	Matter	A message digest
b	Secret key	**SECRET** to one signer
c	Checker	System-wide
d	Raw signature	Appended to signed matter
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

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 unqiue 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 preprend 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 allows 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 results in certain types of forgery attacks against the signature scheme. Therefore, plactic signatures use hashed multiplicative signatures.

In this report and its 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** and **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 that can be represented as a single byte. In this case, every byte string represents a semistandard tableau: the Robinson–Schensted algorithm converts any byte string s into a semistandard tableau $P(s)$. The simple embedding function used in plactic signatures is 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, which can be mitigated by using longer strings.)

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 run in time quadratic in the the input length, while hash functions run in time linear in the input length. In other words, for long messages m, computing $f(m)b$ is actually 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) represent 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.

4.2 Obsoleted parameters: ps12288

A previous version of this report recommended a different set of parameters, `ps12288`. The main advantages of `ps8000` over `ps12288` are:

- Parameters `ps8000` round lengths down decimally, so instead of 512-byte hashes, parameters `ps8000` use 500-byte hashes.
- Parameters `ps8000` exclude the value c from the representation of the public key, giving the public key a smaller representation (two-thirds the size), on the grounds the c will generally be fixed, or known in advance (out-of-bound), or derived as the hash of a shorter string.
- Parameters `ps8000` place the d at the beginning of a signed message, which might slightly discourage readers of a signed message from ignoring the value d in an invalid signature.
- Parameters `ps8000` define b as the hash of a 100-byte secret key, allowing for a smaller secret key, and somewhat mitigating against the user error of choosing a weak value b instead of a randomized value for b.

5 An implementation of plactic signatures

This section presents a C implementation of plactic signatures.

The usefulness of implementations inlcude studying the practicality issues and clarifying any ambiguities in algorithm descriptions.

The NIST post-quantum cryptography project[2] requirements for digital signature implementations are followed.

5.1 Common header files

Some header files are used by various parts of the implementation.

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

File `api.h` listed in Table 4 is an application programming interface (API) header file required by the NIST PQC project requireements. It is meant to

[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`

instruct the programmers how to include the digital signature implementation into C applications. First, the file specifies the byte sizes for keys (secret and public) and signature (expansion). Second, the file specifies a formal string for algorithm name. Finally, specifies 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, as does the reference implementation for plactic monoid multiplication.

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 a header file `plactic.h` listed in Table 6.

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

```
#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`

input byte strings represent tableaus. such as row readings of semistandard tableaus.

The output `d` is byte string of length `alen`+`blen`. The ouptut `d` is to be be represent a tableau, the product of tableaus represented by byte strings `a` and `b`. Generally, `d` is intended to be the row reading of the tableau.

A different tableau representation might be acceptable too – if it does not leak the input tableaus. A column reading might be fine. It is possible to generate a randomized reading too. It is unclear (to me) if any alternate can safely offer any advantage over the row reading representation.

The caller of function `multiply` (such as one of the functions used for digital signatures), needs to ensure that enough memory is allocated for byte strings at locations

5.2.2 A reference implementation

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

The reference implementation allocates a memory buffer to store the the rows of the product tableau, in such a way that insertion of new entries does not require shifting of entire rows. The buffer's size is proportional to

```
/* 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*d, const u*a, int alen, const u*b, int blen){
  int i, j, dlen=alen+blen, r[rmax]={0}; u s[smax], *t[rmax]={s};
  for(i=1; i<rmax; i++) t[i] = t[i-1] + dlen/i;
  for(i=0; i<dlen; i++) insert (t, r, (i<alen)? a[i]: b[i-alen]);
  for(i=rmax-1; 0<=i; i--) for(j=0; j<r[i]; j++) *d++ = t[i][j];}
```

Table 7: File `plactic-ref.c`

the signature parameters, such as the `ps8000` parameters. Including the file `lengths.h`, provides `aelen`, deduced from the signature parameters, which is then used to calculate the buffer size. (The defined type `u` is provided by `lengths.h`.)

The reference implementation assumes that the tableau entries are confined to byte values, 0–255 (as represente by type `u`). This assumption implies that are at most 256 rows (because the tallest column has distinct entries), which the reference implemtation 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 some variables. Array `r` records the row lengths of the tableau. Arrar `r` is initialized such that 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`. Double-indexing of variable `t` provides access to individual tableau entry: `t[i][j]` is the value of the tableau entry located at row i and column j.

The first `for` loop in function `muliptly` sets up the pointers `t[i]` into

s. The second for loop of function multiply iterates Robinson–Schensted insertion, first 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 d.

The function insert implements Robinson–Schensted insertion. The first for loop bumps entries into the rows. Nested inside the first for loop is a second inner for loop, which finds the entry that needs to be bumped. When there is no entry to bump, the for loops are done. The last line of function insert appends an 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) to find the entry of a row that needs to be displaced.

```
/* 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*d, const u*a, int alen, const u*b, int blen){
  int i, j, dlen=alen+blen, r[rmax]={0}; u s[smax], *t[rmax]={s};
  for(i=1; i<rmax; i++) t[i] = t[i-1] + dlen/i;
  for(i=0; i<dlen; i++) insert (t, r, (i<alen)? a[i]: b[i-alen]);
  for(i=rmax-1; 0<=i; i--) for(j=0; j<r[i]; j++) *d++ = t[i][j];}
```

Table 8: File plactic-search.c

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 this choice of the lowest 64 rows is that sometimes scanning is faster than binary search, especially for short rows. Timing versions based on row length instead of row index (in the tableau) were slower, perhaps due to some optimization capabilities of the compiler. The choice of 64 for the number of lowest rows to use binary search was tuned approximately minimize verification time.

For the recommended parameters `ps8000`, this modification gives a small speed-up. For signing, the speed-up is about 5%, and for verifying it is about 15%.

For larger parameters, a greater speed-up would be expected, but a retuning of the condition over whether to scan to do binary search should be made.

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 Plactic multiplicattion with a given row reading

To verify a signature d, the verifier needs to compute dc (the signcheck). Generally, the signer will have computed d as the row reading of a tableau.

The verifier can use this fact to speed up the computation of dc. Because d is a row reading of a tableau, the tableau form of d can be recovered quickly, more quickly than using the Robinson–Schensted algorithm.

File `plactic-prep.c` lised in Table 10 implements this trick to speed up verification by about 15% compared the plactic signtures using file `plactic-search.c`.

```
/* Plactic multiplication. (c) Dan Brown, BlackBerry, 2021 */
// Binary search (in lower rows), using tableau prepartion of d
#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, k, 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]>a[k+1];
    for (j=k=0; t[i][j]=a[k], r[i]++, k<alen-1; k++)
      if (a[k] > a[k+1]) i--, j=0; else j++;
    i=alen; } else i=0;
  for(; i<ablen; i++) insert (t, r, (i<alen)? a[i]: b[i-alen]);
  for(i=rmax-1; 0<=i; i--) for(j=0; j<r[i]; j++) *ab++ = t[i][j];}
```

Table 10: File `plactic-prep.c`

5.2.5 Low-memory plactic multiplication

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

The function `multiply` does not allocate any new memory for an array, and instead assumes that the caller of the function `multiply` has allocated

```
/* 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*d, const u*a, int alen, const u*b, int blen){
  int i,j,k;
  for(i=0; i<alen+blen; i++)
    d[i] = (i<alen)? a[i]: b[i-alen];
  for(i=0; i<alen+blen; i++)
    for(j=i-2; 0<=j && d[j+2] < d[j+1]; j--)
      for(k=1; k<=2; k++)
        for(; 0<=j && knuth(k, (d+j) ) ; j--)  ;}
```

Table 11: File `plactic-lowmem.c`

the necessary memory.

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

In some previous versions of this report, a version of the file `plactic-lowmem.c` served as the reference implementation of plactic monoid multiplication. The current reference implementation is based on the Robinson–Schensted algorithm, which pre-dates Knuth relations defining plactic monoid as used in our low-memory implementation. Furthermore, the Robinson–Schensted algorithm seems to be considered more widely than the Knuth relations, so is arguably more established, and thus more appropriate as a reference implementation.

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. To do this, file `plactic-constant.c` is constructed by modifying file `plactic-lowmem.c`. The secret-dependent branching statements from `plactic-lowmem.c` are replaced by non-branching statement that use boolean-based arithemtic, instead of conditional instructions.

```
/* 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*d, const u*a, int alen, const u*b, int blen){
  int g,h,i,j,k;
  for(i=0; i<alen+blen; i++)
    d[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, d+j);
      g  = k | (2==h);
      h -= k & (0==h);}}
```

Table 12: File `plactic-constant.c`

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 remdinder 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, `plactci-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.

5.3 Signing implementation

File `sign.c` listed in Table 14 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 13.

```
#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 13: File `sign-defs.c`

The reference implementation fixes the checker to be system-wide, as the output of the hash function SHAKE-128, applied to the official name parameters of the plactic signatures `placticsignature8000bits`.

```
/* 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 14: File `sign.c`

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 [Bro21]) 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 [Bro21]). 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 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.

A Code size

Arguably, more complicated algorithms are more difficult to study. In particular, a security analysis might need more time for complicated algorithms. Quantifying how complicated an algorithm is difficult.

This section listed some preliminary measurements for plactic signatures. It uses the C reference implementation, the word-count program `wc`, the C indentation program `indent`, the size of ojbect files, and executable.

Of course, comparison to the other post-quantum algorithms might not be appropriate, if their reference implmenetations are written with very different objectives.

Table 19 lists the C source files (including headers) that would be unique to plactic signatures, excluding files common to other signatures (such as random number generation and hashing). There are under 60 lines and under 2500 bytes.

```
$--$  files="[talp]*.h s*.c *ref*.c"; wc -lcL $files
   8  352   69 api.h
   8  238   44 lengths.h
   1   56   55 types.h
   2   82   62 plactic.h
  21  825   69 sign.c
  14  533   72 sign-defs.c
  14  679   65 plactic-ref.c
  59 2471   72 total
```

Table 19: Source code size (packed)

But the source in these files is packed in an unconventional packed style, which arguably underrates their complexity. The program `indent` can somewhat correct for this, with the results shown in Table 20.

```
$--$  files="[talp]*.h s*.c *ref*.c"; fmt='%4d%5d%5d %s\n' ; \
      for file in $files; do
      printf "$fmt" $(indent < $file | wc -lcL ) $file
      done ; \
      printf "$fmt" $(cat $files | indent | wc -lcL) total

   8  373   78 api.h
  10  242   42 lengths.h
   2   89   69 plactic.h
   2   56   30 types.h
  42  908   73 sign.c
  13  522   72 sign-defs.c
  28  786   72 plactic-ref.c
 107 2978   78 total
```

Table 20: Source code size, after automated indentation

Another way measure how complicated the algorithm is with the size of the object files that implement the algorithms. Table 21 lists the object sizes, when a maxium level of optimization is applied.

```
$--$  gcc -c -O3 plactic-ref.c sign.c ; wc -c [ps]*.o
2448 plactic-ref.o
3288 sign.o
5736 total
```

Table 21: Object file sizes

Table 22 list object sizes when the compiler optimizes for small sizes. For comparison, similar object files for auxiliary files (hashing and randomization) used by the reference implementation are listed too.

B Some timing results

A file `timer.c`, listed in Table 23, is a simplistic timing program for plactic signatures.

```
$--$  gcc -c -Os plactic-ref.c sign.c ; wc -c [ps]*.o
2016 plactic-ref.o
3152 sign.o
5168 total
$--$  gcc -c -Os keccak.c rng-util.c ; wc -c [kr]*.o
2784 keccak.o
1824 rng-util.o
4608 total
```

Table 22: Smaller object file sizes

For an example run of the timing program, see Table 24. This was run on a regular personal computer without making special adjustments for accurate benchmarking (pausing all other process, disabling hyperthreading and overclocking).

Timing results for the other implementations are given in Table 25

The timinig results seem fairly consistent with SUPERCOP timing results, run locally on the same device, under similar conditions.

Timing results, under similar testing conditions, using implementation `plactic-constant.c` of plactic multiplication is 22 times slower for key generation and signing, and 35 times slower for verification.

```
/* 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;
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, (excluding max)\n",
  avg, (cycles? "cycles": "nanoseconds"), dev);}
#define TIME(CODE,...) for(t=s2=s=0, i=reps; i--; ){ \
__VA_ARGS__; \
if (!all) fprintf(stderr,"%5d\r",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");} \
  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("\aFAILURES: %d out of %d tries !!!!\a\a\a\n",fail, 2*reps);
  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 23: File `timer.c`

```
$--$ 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 24: A timing run of the reference implementation

```
$--$ 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 4.980304e+05 cycles, relative deviation 0.063767, (excluding max)
Signing, under same key, same message: 'Time me!'
Average 4.567768e+05 cycles, relative deviation 0.017078, (excluding max)
Signing, under new keys, new random 200-byte messages
Average 4.741554e+05 cycles, relative deviation 0.021355, (excluding max)
Verifying same signature, under same key, of same message: 'Time me!'
Average 1.019566e+06 cycles, relative deviation 0.011711, (excluding max)
Verifying signatures, under new keys, of new random 200-byte messages
Average 1.027112e+06 cycles, relative deviation 0.018946, (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 25: Other timing results

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 a Linux system.

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, which would usally hand-typed (and quoted if it contains spaces).

Because a command line argument is terminated by a byte of value zero, the message to be signed cannot contain a zero-valued byte. Large binary files such as images, videos, executable programs, might easily contain such zero-valued bytes. (Large files without zero bytes can sometimes be signed by using shell parameter expansion.)

File `ps-util-help.c` in Table 27 describes the user interface of the simplistic C utility. The terse instructions are intended as a reminder about the utility's interface to a user already well-versed in (plactic) signatures. 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.

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).

```
#include <stdio.h>
#include <string.h>
#include "api.h"
#include "ps-util-help.c"
#define MAX_LEN 1000000
int err(int r, char*e){
  fprintf(stderr, "psu: %s.  Try ./psu --help\n", e); return r; }
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[MAX_LEN];
  ll sklen,slen;
  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");
  crypto_sign(sig, &slen, (u*)msg, strlen(msg), sk);
  fwrite(sig,1,slen,stdout); return 0;}
int ver(char *pk_filename) {
  u pk[CRYPTO_PUBLICKEYBYTES], msg[MAX_LEN], sig[MAX_LEN];
  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):\n"
  " ./psu 2>pk|./psu 'Hello World'|(sleep 0.1;./psu pk open)\n"
  "Plactic signature experimental utility. Dan Brown, BlackBerry.\n"
  ); return 4;}
```

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

D 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;
}
```

E 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.

References

[Bro21] Daniel R. L. Brown. Plactic key agreement. Cryptology ePrint Archive, Report 2021/625, 2021. https://eprint.iacr.org/2021/625. 1, 2.1, 6.1, 6.2

[RS93] Muhammad Rabi and Alan T. Sherman. Associative one-way functions: A new paradigm for secret-key agreement and digital signatures. Technical Report CS-TR-3183/UMIACS-TR-93-124, University of Maryland, 1993.

https://citeseerx.ist.psu.edu/viewdoc/versions?doi=10.1.1.118.6837.
1, 2