

Plactic signatures

Daniel R. L. Brown*

August 3, 2021

Abstract

Plactic signatures use the plactic monoid (semistandard tableaux 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 monoid¹ 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

¹For 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 used.

Custom terminology for multiplicative signatures helps to discuss features specific to multiplicative signatures, for comparison to generic digital signatures.

2.1 Multiplicative semigroups, an overview

Recall the definition of a (**multiplicative**) **semigroup**. Firstly, it is a set where any two elements can be multiplied with a result in the set. In other words, 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: $a(bc) = (ab)c$.

This report fixes the semigroup to be Knuth’s **plactic monoid**. An introduction (for cryptographers) to the plactic monoid may be found in [Bro21].

Recall that elements of the plactic monoid are **semistandard tableaux**. Elements can also be represented by their row readings, a concatenation of the tableau’s rows. More generally, every sequence (with entries in same finite ordered set as entries of the tableaux) represents a unique tableau, via application of the Robinson–Schensted algorithm. Each tableau has multiple sequence representations, but the standard representation is the row reading of the tableau. The values d and e must be communicated in standard representation.

Multiplication is Knuth multiplication of semistandard tableaux. This amounts to first concatenating of the row reading of the two tableaux, and then applying the Robinson–Schensted to put the concatenation back into semistandard tableau form.

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. Occasionally, a signer is identified via the public key.

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. 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 $[c, e]$, that signer $[c, e]$ has **signed** matter a , that matter a has been signed **by** $[c, e]$, and that d is a signature **under** $[c, e]$.

The two sides of (1) are different in invalid signatures. Call ae the **end-matter** and dc the **signcheck**. 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.

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 verifiable 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 semigroup. To fully specify the multiplicative signatures scheme, the subsets A , B , and C should be specified. Furthermore, the methods to choose elements a, b, c in the subsets A, B, C must also be specified.

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	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

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 compute $f = ab = f(m)b$. To **verify** signed-message $[m, d, f]$ under public key $[c, e]$, the verifier checks that $f(m)e = dc$.

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 the key. 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 $[d, f]$ is not too long.

Sometimes, the key k can be fixed for the whole system. In this case, the signed message $[a, d, f]$ reduces to $[a, d]$, because it is actually unnecessary for the signer to transmit k to a verifier.

Sometimes, the signer will choose k randomly from a key space.

Sometimes, the signer will choose k as a deterministic, pseudorandom function of the message, like this $k = h_b(m)$.

Multiplicative signatures can be considered to be a special case of hashed multiplication signatures if we fix the hash function f to be the identity function, defined $f(m) = m$. To be clear, this only allows us to sign messages that are already elements in the semigroup.

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. This is mostly for simplicity.

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 tableaux 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 tableaux of a given length. (The Robinson–Schensted map $s \mapsto P(s)$ is not injective, so it introduces a bias (non-uniformity) in the tableaux when the input is a unbiased (uniformly distributed) byte string. For digital signatures, this bias seems quite harmless. It slightly increase 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 §5.

4 Suggested parameters

For concreteness, this report suggests some specific parameters.

4.1 Recommended parameters: ps12288

The recommended set of parameters, `ps12288`, is described below.

- Tableau entries are bytes, numbers ranging from 0 to 255.
- Tableaus are represented by byte strings. A byte string s represents the tableau $P(s)$, where P is the Robinson–Schensted function (mapping strings to semistandard tableaux).

- Values a, b, c (message digest matters, secret keys, and checkers) are each 512 bytes.
- Values d, e (signatures and endpoints) are each 1024 bytes.
- Standard representation (row readings of semistandard tableaux) is used to communicate d and e .
- Public key $[c, e]$ is the concatenation of byte string representation c and e , represented as 1536 bytes, which is 12288 bits (hence the name `ps12288`).
- The hash function is SHAKE-128.
- The hash function output length is 512 bytes.
- The embedding function is the identity function, sending byte strings (512 bytes from SHAKE-128) to byte strings (representing semistandard tableaux).

The aim for parameters `ps12288` is that any successful forgery attack (with success rate at least one half) takes computation of at least 2^{128} steps (bit operations) – or is infeasible in some other way (negligible success probability, excessive number of queries to honest signers, and so on).

The parameters `ps12288` deliberately include a large margin for error compared to the estimated costs of the best known attacks. (See §5 for various forgery attacks, and [Bro21] for the main attack, plactic division.) Plactic signatures are very new, so better attacks would not be unthinkable. The `ps12288` parameters rely on the hope that potential better attacks remain within the large margins for error. (Loosely speaking, the hope is the better attacks will still have exponential run-time.)

5 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.

5.1 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.

5.2 Left division to find a secret key from a signature

Suppose that binary operator \backslash 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 $i \in \{1, \dots, n\}$, 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 $b = [a_1, \dots, a_n] \setminus [d_1, \dots, d_n]$. 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 signatures, we use the left version of the division-set operator, \setminus , which is equivalent to the operator $//$ via the anti-isomorphism of the plactic monoid. (In other semigroups, those non anti-isomorphic, different algorithms may be needed for left division.) The attacker can compute the set $a \setminus d$. For a valid signature, the actual secret key used belongs to this set: $b \in a \setminus d$. In this case, one can search for any $b \in a \setminus d$ 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 $d//b$ has 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 a to be half the length of c .

5.3 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.

5.4 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.

5.5 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 be 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.

5.6 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.

6 A reference implementation

A reference implementation for plactic signatures is provided for additional clarity.

The reference implementation follows the NIST PQC² the requirements for digital signature implementations in C. This includes an interface that is generic across many different possible digital signature algorithms.

6.1 Implementing plactic monoid multiplication

File `plactic.c` in Table 3 implements plactic monoid multiplication (for any given size of tableaux). The obvious header file `plactic.h` is omitted.

```
#define SWAP(a,b) ( a^=b, b^=a, a^=b, 1 )
#define KNUTH(k,xyz) ( \
    (xyz[2] < xyz[k-1]) & (xyz[0] <= xyz[k]) && \
    SWAP(xyz[1], xyz[(k+1)%3]) )
void multiply (
    unsigned char *d,
    const unsigned char *a, unsigned long long alen,
    const unsigned char *b, unsigned long long blen)
/* WARNINGS: not constant time, not safe for memory overlap */
{ 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; j>=2 && d[j] < d[j-1]; j--)
      for(k=1; k<=2; k++)
        for(; j>=2 && KNUTH(k, (d+j-2) ) ; j--); }
```

Table 3: File `plactic.c` (multiplication in the plactic monoid)

The inputs `a` and `b` to the `multiply` function are byte strings of lengths given by input `alen` and `blen`. The input byte strings can be, but are not require to be, row readings of semistandard tableaux. The output byte string

²The NIST post-quantum cryptography (PQC) project takes its requirements from the SUPERCOP system of timing cryptography.

will be the row reading of a semistandard tableau (which can be considered to be the standard representation for purposes of the reference implementation).

The output `d` byte string is computed by concatenating the byte strings `a` and `b`, and then applying the Robinson–Schensted algorithm to obtain a semistandard tableau, with d being the row reading of this semistandard tableau.

Internally, the algorithm used to implement the Robinson–Schensted does not use a two-dimensional array, but rather a one-dimensional array. The Knuth relations are applied iteratively to achieve the insertions of entries the semistandard tableau. The Knuth relations applied are equivalent to the Robinson–Schensted insertion of symbols into semistandard tableaux.

The implementation assumes that input `a` and `b` and `d` point to memory locations such that multiplication runs correctly. The caller of the `multiply` must ensure this, for example, by setting `a` and `b` to point to properly allocated, non-overlapping, computer memory.

The reference implementation does not have optimized speed.

The reference implementation does not have optimized side channel resistance.

6.2 Application programming interface

File `api.h` in Table 4 specifies the byte sizes, the specific algorithm name, and also the C function prototypes for key generation, signing and verifying. The reference implementation uses public key of size 12288 bits (1536 bytes).

This reference implementation interprets the input parameters `pk` and `sk` to the the function `crypto_sign_keypair` as memory locations with potentially uninitialized data, which therefore should not be used as input (to recover a public key from a pre-existing secret key, for example).

6.3 Signing implementation

File `sign.c` in Table 5 implements key generation, signing and verifying.

The reference implementation fixes the checker to be system-wide, as the output of the hash function SHAKE-128, applied to the official algorithm name `Plactic_Signature_12288`.

Optionally, a user of reference implementation `sign.c` can change the value of this checker. The user can re-assign the global variable `name`, point-

```

#define CRYPTO_SECRETKEYBYTES 512
#define CRYPTO_PUBLICKEYBYTES 1536
#define CRYPTO_BYTES 1024
#define CRYPTO_ALGNAME "Plactic_Signature_12288"
int crypto_sign_keypair(unsigned char *pk, unsigned char *sk );
int crypto_sign(unsigned char *sm, unsigned long long *smlen,
  const unsigned char *m, unsigned long long mlen,
  const unsigned char *sk);
int crypto_sign_open(unsigned char *m, unsigned long long *mlen,
  const unsigned char *sm, unsigned long long smlen,
  const unsigned char *pk);

```

Table 4: File `api.h` (bytes sizes and name)

ing to a string of the user’s choice. The reference implementation will hash this string instead of the official algorithm name.

7 Towards an optimized implementation

The reference implementation of plactic signatures is not ideal, and has several rooms for improvement.

Critically, the reference implementation for multiplication is not constant-time. The runtime depends on the values of the factors. Signing and key generation should be constant because, otherwise, the secret b might get leaked through a timing side channel. Two modifications might help achieve constant time multiplication.

- The Knuth relations on three array elements should be re-implemented as a function that outputs a Knuth state. The Knuth state determines the next Knuth action, which might be to leave the next three array entries unmodified. The main loop of multiplication would simply pass along the state, and would not stop early.
- The swaps of pairs of elements should be implemented as a constant-time conditional swap.

Perhaps the byte sizes chosen are too small, and perhaps a larger range of entries for the tableaux is not needed, not just a single byte. On the other

```

#include <string.h> /* memcpy, memcmp, strlen */
#include "keccak.h" /* FIPS202_SHAKE128 */
#include "rng.h" /* randombytes */
#include "api.h" /* CRYPTO_SECRETKEYBYTES */
#include "plactic.h" /* multiply */
#define L CRYPTO_SECRETKEYBYTES
unsigned char *name=CRYPTO_ALGNAME;
int crypto_sign_keypair (unsigned char *pk, unsigned char *sk)
{ unsigned char *b=sk, *c=pk, *e=pk+L;
  if (sk == pk) return -2; /* TO DO: check for other overlaps */
  if (sk != pk) randombytes(b,L);
  FIPS202_SHAKE128(name,strlen(name), c,L);
  multiply(e, b,L, c,L); return 0; }
int crypto_sign (unsigned char *sm, unsigned long long *smlen,
  const unsigned char *m, unsigned long long mlen, const unsigned char *sk)
{ unsigned char a[L], *d=sm+mlen;
  unsigned char const *b=sk;
  if ( sk==sm ) return -3; /* TO DO: check for other overlaps */
  *smlen = mlen + 2*L; memcpy(sm,m,mlen);
  FIPS202_SHAKE128(m,mlen, a,L);
  multiply(d, a,L, b,L); return 0;}
int crypto_sign_open (unsigned char *m, unsigned long long *mlen,
  const unsigned char *sm, unsigned long long smlen, const unsigned char *pk)
{ unsigned char const *c=pk, *d=sm+smlen-2*L, *e=pk+L;
  unsigned char a[L], ae[3*L], dc[3*L];
  *mlen=smlen-2*L;
  FIPS202_SHAKE128(sm,*mlen, a,L);
  multiply(ae, a,L, e,2*L);
  multiply(dc, d,2*L, c,L);
  if (0 == memcmp(ae,dc,3*L)) {
    memcpy(m,sm,*mlen); return 0;
  } else {
    randombytes(m,*mlen); *mlen=0; return -1;}}

```

Table 5: File `sign.c` (key generation, signing, verifying)

hand, perhaps the byte sizes are too large, and smaller tableaus could be used, for a faster implementation.

Verification perhaps does not need constant-time multiplication. For example, in applications where the verified message is public, then all inputs to verification will usually be public, so there is no need to hide them. In this case, it may make sense to optimize the speed of multiplication. Some possible modification might help speed up multiplication:

- Store the tableaus being multiplied as two-dimensional arrays. (The division algorithm in [Bro21] does this, for example.)
- When scanning where to insert entries into rows of tableaus, use a binary search from the beginning of the row to the entry just above the entry below being bumped.

Optimizations should be tested for their effectiveness, because costly overhead of optimization complications might outweigh the intended benefits.

The brevity of the reference implementation for multiplication in file `plactic.c`, suggests that the most suitable hash function would be one whose reference implementation is similarly brief. Perhaps one of the recent lightweight cryptographic hash functions has a briefer specification than the SHAKE128, and would be more suitable in this regard. (Similarly, since `plactic` signatures are likely to be used in layered diverse cryptography, a distinct hash function for `plactic` signatures would help avoid a single point of weakness.)

Compression of public keys and signatures might be possible. The NIST API suggests fixed length public keys and signatures, so compression is not immediately compatible with this API. The portion c in a public key is usually fixed system-wide (or derived as a hash of the signer's name). This suggests that it can be omitted from the public key, as a **compression** operation. Other possible compression methods may be possible. For example, a semistandard tableau contains some redundant information, suggesting further compression is possible. Back of the envelope calculations suggest that compression down to about half the number of bytes in a row reading might be possible.

A Experimental utilities

This section provides experimental command-line utilities: executable programs for plactic signatures, that can generate keys, sign messages, and verify signatures. The utilities can run on a Linux system.

The intended messages to sign are hand-typed text, not arbitrary data.

A.1 A simplistic utility

File `ps-util.c` in Table 6 combines some standard C libraries with the plactic signature library. The message to be signed is supplied as a command-line argument.

File `help.c` in Table 7 described 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.

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.

The utility could be re-written to handle arbitrary messages. Perhaps if the signing mode, indicated by supplying single argument mode, signing, it could first test if the command-line argument is a filename, and if so, sign the file. Otherwise it would sign the command-line argument.

A.2 A more versatile front-end utility

A second utility is written as a shell script (bash), as a more flexible (perhaps more friendly) front-end user interface for the first utility (which the second utility uses).

File `ps-util.sh` in Table 8³ uses the `ps-util` utility as a backend, to provide a more sophisticated user interface.

³Like other files in this report, manual line-merging and space-deletion has been used to squeeze the file into a single page. Shells are more delicate than C in handling blank space, fewer spaces can be deleted and some newlines need to be replaced by semi-colons. So, this version is more likely to have bugs.

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "api.h"
#include "help.c"
#define MAX_LEN 1000000
int key(void) {
    unsigned char 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) {
    unsigned char sk[CRYPTO_SECRETKEYBYTES], sig[MAX_LEN];
    unsigned long long sklen,slen;
    if (isatty(fileno(stdin))) {help (); return 6;}
    sklen=fread(sk,1,CRYPTO_SECRETKEYBYTES,stdin);
    if (sklen != CRYPTO_SECRETKEYBYTES ) {
        fprintf(stderr,"\nBad secret key\n"); return 2;}
    crypto_sign(sig,&slen, msg,strlen(msg), sk);
    fwrite(sig,1,slen,stdout); return 0;}
int ver(char *pk_filename) {
    unsigned char pk[CRYPTO_PUBLICKEYBYTES], msg[MAX_LEN], sig[MAX_LEN];
    unsigned long long 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 {fprintf(stderr,"\nBad signature\n");return 1;}}
            else {fprintf(stderr, "\nBad signature (too short)\n");return 3;}}
            else {fprintf(stderr,"\nBad public key\n");return 4;}}
            else {fprintf(stderr,"\nBad public key (could not open file)\n");return 5;}}
int main (int c, char **a){
    return 1==c?key(): 2==c?sig(a[1]): 3==c?ver(a[1]): help();}

```

Table 6: File ps-util.c (key generation, signing, verifying)

```

int help (void){ printf(
    "Plactic signature utility, by Dan Brown (BlackBerry).\"
    "\nUsage summary:\"
    "\n ps-util [message-or-filename [...]]"
    "\n\"
    "\n task   |args| arg1       | stdin   | stdout  | stderr\"
    "\n -----+-----+-----+-----+-----+-----\"
    "\n new     | 0 |           |         | key     | lock\"
    "\n sign   | 1 | 'message' | key     | signature | \"
    "\n verify | 2 | lock file | signature | message  | alert\"
    "\n help   |1,3+|           | terminal | this     | \"
    "\n\"
    "\n Note: signature has message as prefix.\"
    "\n ./ps-util 2>pk|./ps-util Hello|(sleep 0.1;./ps-util pk -);echo\"
    "\n See demo script ps-util.sh, for more realistic example.\n\"
    ); return 4;}

```

Table 7: File `help.c` (help function)

For signing, the shell script utility takes the message as either the command-line arguments or the terminal `stdin` if there are no command-line arguments. The last word of the message is presumed to be the signer's name, and is used to derive the filename of the encrypted secret signing key.

If secret key file does not exist, then the script asks if the user would like to create a new key. In other words, there is no dedicated interface for key pair generation. If the user says no, then the utility assumes that the user wants to verify a signed message. Also, if any words in the message look like the filename of a signed message created by the utility, then the utility assumes the verification is needed.

The secret keys are encrypted for security. The utility uses the OpenSSL utility for encryption and decryption of the secret keys.

The verification tries to verify all words of the input as though they were filenames, and asks the user which public key files to be used for verification.

```

#!/bin/bash
echo () { builtin echo "$*" > /dev/stderr ; }
if [ $# -gt 0 ] ; then input="$*" ; else
  if [ -t 0 ] ; then
    echo 'Type a message to sign (or files to verify).' ; echo 'Ctrl-D to finish.'
    mapfile input_array ; input=$(printf %s "${input_array[@]}") ; input="$input"$'\n'
  else
    echo This script assumes input from a terminal, sorry. ; exit 1 ; fi ; fi
set -- $input # TO DO: this might be insecure: user input --> shell!!!
last_word="${!#}"; signer="$last_word" # TO DO: remove hyphens --Dan
sk=$signer.sk ; pk=$signer.pk # TO DO: search the path for $signer.sk
if [ -f $sk ] ; then signing=yes ; fi
if which openssl > /dev/null ; then
  aes="openssl enc -aes-256-cbc -iter 1000"
  encrypt () { $aes ; } ; decrypt () { $aes -d ; }
else
  encrypt () { cat ; } ; decrypt () { cat ; } ; fi
if [ -z $signing ] && grep -v [-]signed-message[.] <<< "$input" ; then
  echo "No signer key file $sk exists here."
  read -p "Do you want to generate a new keypair for $signer? [y/n] "
  if [ $REPLY = y ] ; then
    if [ -f $pk ] ; then
      echo $pk already exists, making a backup ; cp -f -b -v $pk $pk ; fi
    echo Generating a new key pair
    ./ps-util 2> $pk |
      (echo The secret key should be encrypted. ; encrypt > $sk )
    echo Public key '(lock)' stored in $pk ; echo Secret key stored in $sk
    signing=yes ; fi ; fi
if [ $signing ] ; then
  if [ -t 1 ] ; then # stdout it a terminal, signed message > file
    sm=$(mktemp $signer-signed-message.XXXXXX)
    echo Secret key must decrypted to sign ; decrypt < $sk |
      ./ps-util "$input" > $sm ; echo Signed message stored as file $sm
  else # stdout is a file or pipe, just write
    ./ps-util < $sk "$input" ; fi
  exit 0 ; fi
message=$(mktemp TEMP-recovered-message.XXXXXX)
echo Choose a public key file to verify signed message files: ; echo '(q or Ctrl-D to quit)'
select pk in *.pk ; do
  if [ -z $pk ] ; then break ; fi
  for file in $* ; do
    if [ -f $file ] ; then
      if ./ps-util <$file $pk verify > $message ; then
        if [ $messages_printed ] ; then printf '\n--\n' ; fi
        echo File $file verifies under $pk.
        echo The verified message is: ; echo
        cat $message ; messages_printed=yes
      else
        echo '!!!!'; echo ERROR: "$file" does not verify ; echo '!!!!'; fi
    else
      echo ERROR: No file "$file" found ; fi ; done
  echo ; echo Choose a public key file to verify signed message files:
  echo '(q or Ctrl-D to quit)' ; done ; \rm $message

```

Table 8: File ps-util.sh (key generation, signing, verifying)

B Auxiliary implementations

File `rng-util.c` in Table 9 likely suffices for the way that the plactic signature utility uses random numbers. The header file `rng.h` can then be as simple as specifying the prototype for the function `randombytes`.

```
#include <stdio.h>
#include "keccak.h"
int randombytes(unsigned char *x, unsigned long long xlen)
{
    fread(x,1,xlen,fopen("/dev/urandom","r"));
    FIPS202_SHAKE128 (x,xlen,x,xlen);
}
```

Table 9: File `rng-util.c`

File `keccak.c` in Table 10 is an indentation-added excerpt of one of the implementations of SHAKE-128 from the official github source code for Keccak.

C 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

```

#define FOR(i,n) for(i=0; i<n; ++i)
typedef unsigned char u8;
typedef unsigned long long int u64;
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);}
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)
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 10: File keccak.c, implementing the SHAKE-128 full-domain (extendable output) hash function

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, 4.1, 5.1, 5.2, 7
- [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/download?doi=10.1.1.118.6837&rep=rep1> 1, 2