Plactic signatures

Daniel R. L. Brown*

September 30, 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 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 tableaus**. 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 tableaus) 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 tableaus. This amounts to first concatenating of the row reading of the two tableaus, 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 **endmatter** 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 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 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 §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 provides an implementation for plactic signatures is provided.

An implementation can clarify the practicality issues with plactic signatures. An implementation can clarify any ambiguities in the written description of the previous sections of this report.

The implementation follows the NIST PQC[2] the requirements for digital signature implementations in C.

5.1 Implementing plactic monoid multiplication

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

The C implementations share a header file `plactic.h` listed in Table 3.

```
typedef unsigned char u;
void multiply(u*ab, const u*a, int alen, const u*b, int blen);
```

Table 3: File `plactic.h`

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

Plactic signatures represent some plactic monoid elements as pseudo-random bytes strings, so our input to plactic multiplication will allow any byte string. The two byte strings are concatenated, and then a form of the Robinson–Schensted algorithm is applied to create a semistandard tableau. The semistandard tableaus is then represented as a byte string.

5.1.1 A reference implementation

File `plactic.c` in Table 4 is a reference implementation of plactic monoid multiplication.

```
/* Plactic multiplication. (c) Dan Brown, BlackBerry, 2021 */
// reference implementation
#include "plactic.h"
#define swap(a,b) (a^=b, b^=a, a^=b)
int knuth (int k, u*xyz) {
  return (xyz[2] <  xyz[k-1]) &&
         (xyz[0] <= xyz[k])   &&
    (swap(xyz[1] ,  xyz[(k+1)%3]), 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; j>=0 && d[j+2] < d[j+1]; j--)
      for(k=1; k<=2; k++)
        for(; j>=0 && knuth(k, (d+j) ) ; j--)  ;}
```

Table 4: File `plactic.c`

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 required to be, row readings of semistandard tableaus. The output byte string 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 tableaus.

The implementation assumes that input `a` and `b` and `d` point to memory locations such that plactic 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.

5.1.2 Constant-time plactic multiplication

File `plactic-constant.c` listed in Table 5 implements plactic monoid multiplication similarly to the implementation in file `plactic.c` except that it tries to run in constant-time, meaning that the sequence of computer instructions are the same for any given inputs. In other words, no branching based on the input data is used. The output depends on the input only by a sequence of arithmetic operations.

The arithmetic operations include some boolean operations and some array look-ups, and some computations modulo three. To achieve a constant-time implementation, the translation from C source code to computer machine code, the compiler must translate each C instruction into a constant-time machine, and no short-cuts of omitting C instructions should be taken (compilers often try to optimize away C instructions that it deems have no effect). Therefore, for `plactic-constant.c` to produce a constant-time implementation, some care is needed to the compilation.

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.

```
/* Plactic multiplication. (c) Dan Brown, BlackBerry, 2021 */
// Constant-time?
#include "plactic.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 5: File `plactic-constant.c`

File `plactic-constant.c` seems to be make plactic signatures with parameters `ps8000` about six times slower than using `plactic.c`. This is likely because `plactic.c` often stops computing earlier than `plactic-constant.c`, during each insertion of a new entry into the tableau, whereas `plactic-constant.c` essentially continues with dummy operations as if the worst case insertion was necessary, meaning that an entry had to moved all the way to be beginning of the string.

The constant-time nature of file `plactic-constant.c` has not been formally tested. Some tests using wall clock time suggest a smaller standard deviation.

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

One possibility is pre-randomization.

5.1.3 Speed-enhanced plactic multiplication

File `plactic-faster.c` listed in Table 6 implements plactic monoid multiplication using doubly-indexed arrays.

File `plactic-faster.c` seems to run about six times faster than file `plactic.c`. It is likely to have worse side-channel resistance. For example, it is more likely to suffer from cache-timing attacks, because it uses secret-dependent array indexing with indices larger than 2.

5.2 Application programming interface

File `api.h` in Table 7 specifies the byte sizes, the specific algorithm name, and also the C function prototypes for key generation, signing and verifying.

5.3 Signing implementation

File `sign.c` listed in Table 8 implements key generation, signing and verifying, following the interface defined in file `api.h`, and using the helper definitions from file `sign-defs.h` listed in Table 9.

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 multiplication. (c) Dan Brown, BlackBerry, 2021 */
// faster?
#include "plactic.h"
#define swap(a,b) (a^=b, b^=a, a^=b)
enum{exp=11, boxes=exp*(1<<exp), maxrows=256};
void insert(u**tableau, int*rowlen, u entry){
  int i, j;
  for(i=0;
      i<maxrows && rowlen[i] && tableau[i][rowlen[i]-1]>entry;
      i++){
    for(j=0;
        j<rowlen[i] && tableau[i][j]<=entry;
        j++)
      ; // inner for loop has empty body
    swap(entry, tableau[i][j]);}
  tableau[i][rowlen[i]++] = entry;}
void multiply (u*d, const u*a, int alen, const u*b, int blen){
  int i, j, dlen=alen+blen, rowlen[maxrows]={};
  u list[boxes], *tableau[maxrows]={list};
  for(i=0; i<dlen; i++)
    d[i] = (i<alen)? a[i]: b[i-alen];
  for(i=1; i<maxrows; i++)
    tableau[i] = tableau[i-1] + 1 + dlen/i;
  for(i=0; i<dlen; i++)
    insert (tableau, rowlen, d[i]);
  for(i=maxrows-1; i>=0; i--)
    for(j=0; j<rowlen[i]; d++, j++)
      *d = tableau[i][j];}
```

Table 6: File `plactic-faster.c`

```
#define CRYPTO_SECRETKEYBYTES  100
#define CRYPTO_PUBLICKEYBYTES 1000
#define CRYPTO_BYTES          1000
#define CRYPTO_ALGNAME  "placticsignature8000bits"
typedef unsigned char u; typedef unsigned long long ll;
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 7: File `api.h`

Optionally, a programmer using `sign.c` can change the value of this checker, as follows. The programmer can re-assign the global variable `name`, pointing 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`. This 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 10 likely suffices for the way that the plactic signature utility uses random numbers.

The header file `rng.h` listed in Table 11 simply specifies the prototype for the function `randombytes`.

A header file `keccak.h` for the SHA-3 Keccak hash function is listed in Table 12.

File `keccak.c` listed in Table 13 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.

```
/* Plactic signatures. (c) Dan Brown, BlackBerry, 2021. */
#include <string.h>
#include "keccak.h"
#include "rng.h"
#include "api.h"
#include "plactic.h"
#include "sign-defs.h"
int crypto_sign_keypair(u*pk, u*sk){
  u array(b), array(c), *e=pk;
  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 array(a), array(b), *d=sm;
  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 array(a), array(c), array(ae), array(dc); const u *d=sm, *e=pk;
  *mlen = 0;
  if (smlen < dlen) return -4;
  sm += dlen, smlen -= dlen; digest (a, sm), digest (c, name);
  multiply (ae, a, e), multiply (dc, d, c);
  if (compare (ae, dc)) return -1;
  *mlen = smlen; copy(m, sm); return 0;}
```

Table 8: File `sign.c`

```
#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 compare(a,b)     memcmp (a, b, b##len)
#define copy(a,b)        ((a && a!=b)? memcpy (a, b, b##len): 0)
#define random(a)        randombytes(a, a##len)
#define array(a)         a[a##len]
#define namelen          strlen((char*)name)
enum { sklen = CRYPTO_SECRETKEYBYTES, elen = CRYPTO_PUBLICKEYBYTES,
 dlen = CRYPTO_BYTES, alen = dlen/2, blen = dlen - alen,
 clen = elen - blen, aelen = alen + elen, dclen = dlen + clen};
u*name=(u*)CRYPTO_ALGNAME;
```

Table 9: File `sign-defs.h`

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

Table 10: File `rng-util.c`

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

Table 11: File `rng.h`

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

Table 12: File `keccak.h`

```
#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 13: 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 $\backslash$ 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 $i \in \{1, \ldots, n\}$, 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 Experimental utilities

This section provides an experimental command-line utility. It can generate key pair, sign messages supplied as a command-line argument, and verify and open signed messages. The utility 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 14 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 15 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.

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.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "api.h"
#include "help.c"
#define MAX_LEN 1000000
int err(int r, char*e){ fprintf(stderr, "psu: %s.\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 (isatty(fileno(stdin))) {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 14: 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 15: File `help.c` (help function)

A.2 A more flexible utility (unfinished)

This section includes an unfinished C implementation of a command-line utility with a flexible (perhaps too flexible) interface.

There are two files in the C implementation. (Both files are larger than the other files in listed in this report, so they are shown in smaller fonts, without being marked as captioned floating tables.) The file `ps-flex-help.c` listed next deals with input and output, including a help message.

```
#define rarg(i)(fopen(arg[(i)], "r"))
#define warg(i)(fopen(arg[(i)], "w"))
#define rtty() (fopen("/dev/tty", "r"))
#define wtty() (fopen("/dev/tty", "w"))
#define ttyi() (isatty(fileno(stdin )))
#define ttyo() (isatty(fileno(stdout)))
#define ttye() (isatty(fileno(stderr)))
#define getf(var,file)(var##read=fread(var, 1, var##buf, file))
#define putf(var,file)(fwrite(var, 1, var##read, file))
#define get(var)  getf(var,stdin)
#define put(var)  putf(var,stdout)
#define pute(var) putf(var,stderr)
#define geta(var,i) getf(var,rarg(i))
#define puta(var,i) putf(var,warg(i))
#define move(new,old)(new##read=old##read, memcpy(new, old, new##read), old##read=0)
#define grab(new,old)(move(new, old), \
  new##read+=fread(new+new##read, 1, new##buf - new##read, stdin))
#define append(var,str)(memcpy(var+var##read,str,strlen(str)), var##read+=strlen(str))
enum {
  msgmax=(1<<24),                          msgbuf=msgmax+1,
  sigmin=CRYPTO_BYTES, sigmax=sigmin+msgmax, sigbuf=sigmax+1,
  pklen=CRYPTO_PUBLICKEYBYTES,             pkbuf =pklen+1,
  sklen=CRYPTO_SECRETKEYBYTES,             skbuf =sklen+1};
unsigned char sig[sigbuf];
void usage (void){ printf("Usage (experimental only): ./psfu [arg1 [arg2 [...]]]\n"); }
void more (void){ printf (" ./psfu --help  # for more example command lines \n");}
void examples (void){ printf (
 "# Generating secret keys and public keys (do as set-up): \n"
 " ./psfu >secret 2>public ; chmod u=r,go-wx secret \n"
```

```
 " ./psfu <secret pk3 ; ./psfu <secret >pk3 \n"
 " rm -i pk4 sk4 && ./psfu pk4 sk4 ; chmod u=r,go-wx sk4 \n"
 " enc='openssl enc -aes-256-cbc -iter 1000'  \n"
 " ./psfu 2>pk2 | $enc >sk2.enc \n"
 " rm -i pk5 && ./psfu pk5 | $enc >sk5.enc \n"
 "# Signing messages (do solemnly): \n"
 " ./psfu <secret 'Hello World'   > hello1.signed \n"
 " ./psfu <secret Hello   World   > hello2.signed \n"
 " ./psfu secret <<<'Hello World' > hello3.signed \n"
 " ./psfu secret                  > typed-message.signed \n"
 " ./psfu <secret /dev/tty        > typed-msg2.signed \n"
 " ./psfu --help | ./psfu secret  > help.signed   \n"
 " ./psfu <secret  public         > public.signed \n"
 " ./psfu <secret  psfu           > psfu.signed   \n"
 " ./psfu secret   psfu  sig2 ; ./psfu psfu sig2 secret \n"
 " ./psfu secret  <psfu >sig2 ;  ./psfu secret   psfu >sig2 \n"
 "  $enc -d <sk2.enc | ./psfu 'Hello ' World > h4.sn \n"
 "# Verifying, viewing, or processing signed messages (do often): \n"
 " ./psfu <public hello2.signed ; ./psfu public <hello2.signed \n"
 " ./psfu public  hello2.signed ; ./psfu hello2.signed public  \n"
 " ./psfu <public * ; ./psfu <hello1.signed *  \n" );}
void title (void){ printf(TITLE); }
int help (int level){
  usage(), (level? examples(): more()), title();
  return 101;}
int failure(int code, char*msg){
  fprintf(stderr,"psfu: Error, %s, sorry!  (Try ./psfu --help)\n", msg);
  return code;}
int success(int aloud, char *msg){
  if(aloud && (1==aloud || wtty()))
    fprintf(1==aloud? stderr: wtty(), "psfu: Success, %s.\n", msg);
  return 0;}
void warn(char *msg){
  fprintf(stderr, "psfu: Warning, %s, check results!\n", msg);}
int proceed(void){
  fprintf(stderr, "psfu: Proceed anyway? [y/n] \a");
  return rtty() &&  'y'== fgetc(rtty());}
```

The file `ps-flex.c` listed next has many lines of code for a decision processs that tries to be very flexible, so that many different command lines can lead to useful actions (keypair generation, signing or verification). In other words, the program tries to find a useful action consistent with the user's command line, and act accordingly.

```
#define TITLE "Plactic signature flexible utility 0.0, Dan Brown, BlackBerry\n"

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "api.h"
#include "ps-flex-help.c"

// To do: fix up failure messages to better address user intent
// Re-factor, to simplify decision tree

int main (int c, char **arg){

  int args=c-1, termin = ttyi(), termout= ttyo(), termerr= ttye();
  unsigned long long skread=0, pkread=0, msgread=0, sigread=0;
  unsigned char *msg=sig+sigmin, pk[pkbuf], sk[skbuf];

  if(! (sklen < pklen))
    return failure(11, "secret keys not smaller than public keys");
  if(sigmin < pklen)
    return failure(12, "signed messages can be smaller than public keys");

  if(0 == args){
    if(termout)
      return help(0);
    if(termin){
      if(termerr)
```

```
      return failure(33, "public key (on stderr) to terminal unsupported");
    crypto_sign_keypair(pk, sk), pkread=pklen, skread=sklen;
    pute(pk), put(sk);
    return success(2, "new secret key and public key generated");}
  if(sklen == get(pk)){
    crypto_sign_keypair(pk, pk), pkread=pklen;
    put(pk);
    return success(1,"public key computed from secret key");}
  return failure(56, "wrong size of secret key");}

if(1 == args){
  int argnotreadable=!rarg(1);
  if(argnotreadable && termin){
    if(termout ||
       0 == memcmp(arg[1], "help", strlen("help")) ||
       0 == memcmp(arg[1], "-", strlen("-")) ||
       0 == memcmp(arg[1], "?", strlen("?")) )
      return help(1);
    if(! warg(1))
      return failure(101, "could not write to public key file");
    crypto_sign_keypair(pk,sk), pkread=pklen, skread=sklen;
    put(sk), puta(pk, 1);
    return success(1, "public key written to file, secret key to stdout");}
  if(termin && termout)
    return help(0),
      failure(77, "cryptographic data to/from terminal unsupported");
  if(termin)
    geta(sk, 1),  get(msg);
  if(!termin && termout){
    get(sk);
    if(sklen == skread){
      if(!argnotreadable)
        return failure(65, "public key file already exists");
      if(! warg(1))
        return failure(66, "public key file not writable");
      move(pk, sk);
      crypto_sign_keypair(pk,pk), pkread = pklen;
      puta(pk, 1);
      return success(1, "computed public key from secret key");}
    if(argnotreadable)
      return failure(23, "no signed message provided");
    grab(pk, sk);
    if(pklen == pkread)
      geta(sig, 1);
    else {
      grab(sig, pk);
      geta(pk, 1);}}
  if(!termin && !termout && argnotreadable){
    get(sk);
    msg=(unsigned char*)arg[1], msgread=strlen((char*)msg);}
  if(!termin && !termout && !argnotreadable){
    get(sk);
    if(sklen == skread)
      geta(msg, 1);
    else {
      grab(pk, sk);
      if(pklen == pkread)
        geta(sig, 1);
      else {
        geta(sk, 1);
        if(sklen == skread)
          grab(msg, pk);
        else {
          grab(sig, pk);
          geta(pk, 1);}}}}
  if(0 == pkread){
    if(termout)
      return help(0),failure(383,"will not write signture or key to terminal");
    if(sklen != skread)
      return failure(1,"secret key wrong size");
    if(msgmax < msgread)
      return failure(8, "message too long");
    if(!termin && !argnotreadable && sklen == msgread){
      warn("dangerously confused over secret key and message");
      if(! proceed())
        return failure( 545, "no signature generated");}
    crypto_sign(sig, &sigread, msg, msgread, sk);
```

```
    put(sig);
    return success(1,"signed message generated");}
  if(0 < pkread){
    if(pklen != pkread)
      return failure(3, "wrong size of public key");
    if(sigmin > sigread || sigmax < sigread)
      return failure(2, "wrong size of signed message");
    if(0 != crypto_sign_open(msg, &msgread, sig, sigread, pk))
      return failure(1, "invalid signature");
    put(msg);
    return success(!termout,"signed message verified and accessed");}
  return failure(77,"could not determine a key (this should never happen)");}

if(2 <= args){
  int f,m,i,p,r,s;
  if(termin){
    if(termout){
      if(3 < args)
        return failure(222, "too many command-line arguments");
      if(2 == args){
        if(! rarg(1) && ! rarg(2)){
          if(!(warg(1)  &&    warg(2)))
            return failure(747, "could not write a key file");
          crypto_sign_keypair(pk, sk), pkread=pklen, skread=pklen;
          puta(pk, 1), puta(sk, 2);
          return success(1, "wrote two key files");}
    if(! (rarg(1)) && rarg(2))
          return help(0);
        p=1;
        geta(pk, p);
        if(pklen != pkread){
          p=2;
          geta(pk, p);
          if(pklen != pkread)
            return help(0), failure(444, "no public key found");}
        geta(sig, 3-p);
        if(sigmin > sigread  || sigmax < sigread)
          return failure(585, "wrong size signed message");
        if(0 != crypto_sign_open(msg, &msgread, sig, sigread, pk))
          return failure(1,"signed message has invalid signature");
        put(msg);
        return success(0,"verified and accessed signed message");}
      // 3 args ... (termin and termout)
      for(s=0,i=1; i<=3; i++)
        if(rarg(i)){
          geta(sk, i);
          if(sklen == skread)
            s=i;}
      if(s){
        for(m=0,i=1; i<=3; i++)
          if(i!=s && rarg(i)){
            geta(msg, i);
            m=i;}
        if(0 == m)
          return failure(543,"no message file found");
        if(msgmax < msgread)
          return failure(975,"message file too long");
        for(f=0, i=1; i<=3; i++)
          if(f!=s && f!=m  && ! rarg(i))
            f=i;
        if(f == 0)
          return failure(434, "output file (for sm) already exists");
        if(sklen == msgread){
          warn("dangerouly confused over secret key and message");
          if( ! proceed())
            return failure(545, "not signing"); }
        crypto_sign(sig, &sigread, msg, msgread, sk);
        if(! warg(f))
          return failure (343, "output file not write-able");
        puta(sig, f);
        return success (1, "signed message written to file");}
      // s==0 (so verify with 3 args)
      for(p=0,i=1; i<=3; i++)
        if(rarg(i)){
          geta(pk, i);
          if(pklen == pkread)
            p=i;}
```

```
    if(0 == p)
      return failure(767, "no public key file found");
    for(m=0,i=1; i<=3; i++)
      if(i != p && rarg(i)){
        geta(sig, i);
        m = i;}
    if(0 == m)
      return failure(676, "no signed message file found");
    for(f=0,i=1; i<=3; i++)
      if(f!=p && f!=m  && ! rarg(i))
        f=i;
    if(0 == f)
      return failure(434, "output file already exists");
    if(pklen == sigread)
      warn("same size public key and signed message");
    if(0 == crypto_sign_open(msg, &msgread, sig, sigread, pk)){
      if(! warg(f))
        return failure(322, "valid signature, but could not write to file");
      puta(msg, f);
      return success(1, "signed message valid, message written to file");}
    return failure(864, "invalid signature");}
  // !termout
  for(s=0,r=0,i=1; i<=args; i++){
    if(rarg(i)){
      r+=1;
      geta(sk, i);
      if(sklen == skread)
        s = i ;}}
  if(0 == s)
    return failure(765, "no secret key found");
  if(1 == r){
    for( i=1 ; i<=args ; i++)
      if(i != s && msgread+strlen(arg[i])+1 < msgmax )
        append(msg,arg[i]), append(msg," ");
    msg[msgread-1]='\n';
    crypto_sign(sig, &sigread, msg, msgread, sk);
    put(sig);
    return success (1, "non-key args signed");}
  if(2 == r){
    if(3 < args)
      return failure (876, "too many command-line arguments");
    geta(msg, 3-s);
    if(msgmax < msgread)
      return failure (777,"message too long");
    if(sklen == msgread){
      warn("dangerously confused over secret key and message");
      if(! proceed())
        return failure (700, "no signing");}
    geta(sk, s);  // in case sk read another file
    crypto_sign(sig, &sigread, msg, msgread, sk);
    put(sig);
    return success(1,"signed file");}
  return failure( 797, "too many files in command-line");}
// ! termin
get(sk);
if(sklen == skread){
  if(termout)
    return help(1);
    for( i=1 ; i<=args ; i++)
      if(msgmax > msgread+strlen(arg[i]) + 1)
        append(msg,arg[i]), append(msg," ");
    msg[msgread-1]='\n';
  crypto_sign(sig, &sigread, msg, msgread, sk);
  put(sig);
  return success(!termout,"command line message signed");}
if(sklen > skread)
  return failure(16,"secret key too small");
grab(pk, sk);
if(pkread == pklen){
  for(i=1 ; i<=args ; i++){
    if(rarg(i)){
      geta(sig, i);
      if(sigmin <= sigread && sigmax >= sigread)
        if(0 == crypto_sign_open(NULL,&msgread,sig,sigread,pk))
          puts(arg[i]);}}}
if(pklen > pkread)
  return failure(99,"public key too small");
```

```
  grab(sig, pk);
  for(i=1 ; i<=args ; i++){
    if(rarg(i)){
      geta(pk, i);
      if(pklen == pkread)
        if(0 == crypto_sign_open(NULL,&msgread,sig,sigread,pk))
          puts(arg[i]);}}
  return success(1,"matching arg files listed");}
return failure(999, "no count of arguments -- this should never happen!");}
```

B 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