Plactic signatures

Daniel R. L. Brown*

October 22, 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).

<table>
<thead>
<tr>
<th>Notation</th>
<th>Name</th>
<th>Typically:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(Attested) Matter</td>
<td>A message digest</td>
</tr>
<tr>
<td>(b)</td>
<td>(Binding) Secret Key</td>
<td>\texttt{SECRET} to one signer</td>
</tr>
<tr>
<td>(c)</td>
<td>Checker</td>
<td>System-wide</td>
</tr>
<tr>
<td>(d)</td>
<td>(Digital) Signature</td>
<td>Appendix of signed matter</td>
</tr>
<tr>
<td>(e)</td>
<td>Endpoint</td>
<td>Signer-specific value</td>
</tr>
</tbody>
</table>

\(a, d\) 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

*\texttt{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

\[ ae = dc. \]  

(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

$$e = bc. \quad (2)$$

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 sens-
sible 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

$$d = ab. \quad (3)$$

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

$$ae = a(bc) = (ab)c = dc. \quad (4)$$

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

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

Table 2: Summary of hashed multiplicative signatures

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

$$ a = f(m). $$

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 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 $ps_{8000}$ 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: $ps_{12288}$
A previous version of this report recommended a different set of parameters, $ps_{12288}$. The main advantages of $ps_{8000}$ over $ps_{12288}$ 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 plastic.h listed in Table 3.

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

Table 3: File plastic.h

\(^2\)The NIST post-quantum cryptography (PQC) project takes its requirements from the SUPERCOP system of timing cryptography.
In this interface, the intent is that the inputs \( a \) and \( b \) to the function `multiply` are byte strings of lengths given by input \( \text{alen} \) and \( \text{blen} \). The input byte strings represent tableaus, such as row readings of semistandard tableaus.

The output \( d \) is byte string of length \( \text{alen} + \text{blen} \). The output \( d \) is intended to be the row reading of the tableau.

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

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

### 5.1.1 A reference implementation

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

```c
/* Plactic multiplication. (c) Dan Brown, BlackBerry, 2021 */
// Reference. (Needs aemax >= aelen from sign-defs.h)
#define swap(a,b) (a^=b, b^=a, a^=b)
enum{aemax=1500, rmax=256, smax=(aemax*6125/1000)};
typedef unsigned char u;
v
void insert(u**t, int*r, u v){ int i=0, j;
  for(; i<rmax && r[i] && t[i][r[i]-1]>v; swap(t[i][j], v), i ++)
    for(j=0; j<r[i] && t[i][j]<=v; j++) ;
t[i][r[i]++]= v;}
void multiply (u*d, const u*a, int alen, const u*b, int blen){
  int i, j, dlen=alen+blen, r[rmax]={0}; u s[smax], *t[rmax]={s};
  for(i=1; i<rmax; i++) t[i] = t[i-1] + dlen/i;
  for(i=0; i<dlen; i++) insert (t, r, (i<alen)? a[i]: b[i-alen]);
  for(i=rmax-1; i>=0; i--) for(j=0; j<r[i]; j++) *d++ = t[i][j];}
```

Table 4: File `plactic-ref.c`

The reference implementation does not aim for side channel resistance.
Function multiply allocates some variables. Array \( r \) records the row lengths of the tableau. Array \( r \) is initialized such that all row lengths are zero. Array \( s \) is a buffer where the tableau entries are stored. Each row of the tableau will occupy a distinct segment of \( s \), chosen so that each segment is long enough to store the maximum possible length for the given row. Array \( t \) is an array of pointers into these segments of \( s \). Double-indexing of variable \( t \) provides access to individual tableau entry: \( t[i][j] \) is the value of the tableau entry located at row \( i \) and column \( j \).

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

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

### 5.1.2 Towards constant-time multiplication

File plactic-constant.c listed in Table 5 is a step towards implementing plactic monoid multiplication in constant time, which is a step towards preventing some side channel attacks. To do this, the code tries to avoid any secret-dependent branching statements. But, the code still uses both input-dependent array-indexing and the C operator \%\%, which are both known to lead to side channel attacks.

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

A signature implementation using file plactic-constant.c failed to pass the tests require by the TIMECOP. This is unsurprising since variable-indexing of arrays was used, but there may be many more reasons.
/* 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
Because the constant-time implementation is much slower than the reference implementation, an alternative side channel mitigation might be useful.

5.2 Application programming interface

File api.h in Table 6 specifies the byte sizes, the specific algorithm name, and also the C function prototypes for key generation, signing and verifying.

```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 6: File api.h

5.3 Signing implementation

File sign.c listed in Table 7 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 8.

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 plastic signatures placticsignature8000bits.

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

A programmer using sign.c can have sign.c generate a new secret key, or the programmer can supply secret key. To supply an old, pre-existing secret key, the programmer calls function crypto_sign_keypair with two equal pointers pk and sk. Equality of these pointers indicates to sign.c
#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 7: File sign.c


```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, dcrlen = dlen + clen};
```

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 9 likely suffices for the way that a plastic signature utility would use random numbers.

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

Table 9: File rng-util.c

The header file rng.h listed in Table 10 simply specifies the prototype for the function randombytes.
int randombytes(unsigned char *x, unsigned long long xlen);

Table 10: File rng.h

A header file keccak.h for the SHA-3 Keccak hash function is listed in Table 11.

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

Table 11: File keccak.h

File keccak.c listed in Table 12 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.

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

$$ b = e/c. $$

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 $[c, e]$).
```c
#include "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);
FIPS202_SHAKE128(const u8 *in, u64 inLen, u8 *out, u64 outLen)
{Keccak(1344, /*256,*/ in, inLen, 0, out, outLen);} static int LFSR86540(u8 *R) { (*R)=((*R)<<1)^(((*R)&0x80)?0x71:0); return ((*R)&2)>>1; }
#define ROL(a,o) ((((u64)a)<<o)^(((u64)a)>>(64-o)))
static u64 load64(const u8 *x) { ui i; u64 u=0; FOR(i,8) { u<<=8; u|=x[7-i]; } return u; }
static void store64(u8 *x, u64 u) { ui i; FOR(i,8) { x[i]=u; u>>=8; } }
static void xor64(u8 *x, u64 u) { ui i; FOR(i,8) { x[i]^=u; u>>=8; } }
#define rL(x,y) load64((u8*)s+8*(x+5*y))
#define wL(x,y,l) store64((u8*)s+8*(x+5*y),l)
#define XL(x,y,l) xor64((u8*)s+8*(x+5*y),l)
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 i,b=0; FOR(1,200) s[i]=0; 
  /*absorb*/
  while(inLen>0) {
    b=inLen&7; inLen>>=1; FOR(i,1,7) s[i]=in[i]; i++; } 
  if (b==0) (KeccakF1600(s); b=0; ) } 
  /*squeeze*/
  while(outLen>0) {
    b=outLen&7; outLen>>=1; FOR(i,1,7) s[i]=out[i]; i++; } 
  if(outLen>0) KeccakF1600(s); } }
```

Table 12: File `keccak.c`
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

\[ b' = a\backslash d. \]  

(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 \( d_i = a_ib \) 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:

\[ d//b = \{a : ab = d\}. \]  

(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 \( a \div d \). For a valid signature the actual secret key used belongs to this set: \( b \in a \div d \). In this case, one can search for any \( b \in a \div d \) 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 \( d\div b \) 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 \( \ast / \) such that

\[
(y \ast / x)x = (x \ast / y)y. \tag{9}
\]

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 \( \ast / \) is not familiar, but convenient for the following discussions.

Some semigroups have fast cross-multipliers.

In a commutative semigroup, \( x \ast / y = x \) defines a cross-multiplier. In a semigroup with a zero element \( 0 \) (such that \( 0z = 0 \) for all \( z \)), \( x \ast / y = 0 \) defines a cross-multiplier. In a group with efficient inversion, \( x \ast / y = y^{-1} \) 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

\[
[a, d] = [c \ast / e, e \ast / c]. \tag{10}
\]
Because this forger uses the cross-multiplier $\ast /$ as an oracle, the forger has no control over the matter $a$ (it is whatever the $\ast /$ 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 $f(m) = c \ast / e$. 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

$$a = a_2 a_1$$

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 Some timing results

A file `timer.c`, listed in Table 13, is a simplistic timing program for plactic signatures.

For an example run of the timing program, see Table 14. This was run on a regular personal computer without making special adjustments for accurate benchmarking (pausing all other process, disabling hyperthreading and overclocking).

The timing results seem fairly consistent with SUPERCOP timing results, run locally on the same device, under similar conditions.
/* Time key generation, signing and verifying */
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "rng.h"
#include "api.h"

int reps= 3456, cycles=1, all=0;
static ll ns (void) {
    struct timespec t;
    clock_gettime(CLOCK_REALTIME, &t);
    return t.tv_sec * (ll)1e9 + t.tv_nsec ;}

static ll cy(void) {
    unsigned int lo, hi;
    __asm__ __volatile__ ( "xorl %%eax,%%eax 
    cpuid 
    : "=r" (lo), "=r" (hi));
    return (unsigned long long)hi << 32 | lo;}

static ll tm (void){return cycles? cy(): ns();}

static double sqrt (double x){double y=x; int i=200;
    while(i--) y = (y+x/y)/2;
    return y;}

static void report_stats (ll sum, ll sum2){
    double avg=sum, dev=sum2;
    avg /= reps; dev /= reps;
    dev -= avg*avg; dev = sqrt(dev); dev /= avg;
    printf ("Average %e %s, relative deviation %f
", avg, (cycles? "cycles": "nanoseconds"), dev);}

#define TIME(CODE,...) for(s2=s=0, i=reps; i--; ){ \
    __VA_ARGS__; \ 
    o=tm(); CODE; n=tm(); \
    d = n-o; s += d; s2 += d*d; \ 
    if (all) fprintf(stderr,"%llu %s",d, i?"
" : "");
else fprintf(stderr,"%5d\r",i);} \
    report_stats(s,s2);

static void time_sign_keypair (void){
    u pk[CRYPTO_PUBLICBYTES], sk[CRYPTO_SECRETBYTES ];
    ll o,n,d,s,s2; int i;
    printf ("Key pair generation\n");
    TIME(crypto_sign_keypair(pk,sk),);}

static void time_sign (void) {
    u pk[CRYPTO_PUBLICBYTES], sk[CRYPTO_SECRETBYTES ];
    u m[200]="Time me!"; ll mlen=strlen((char *)m), smlen;
    u sm[CRYPTO_BYTES+sizeof(m)];
    ll o,n,d,s,s2; int i;
    printf("Signing, under same key, same message: "\"ts1\"ts1\n");
    crypto_sign_keypair(pk,sk);
    TIME(crypto_sign(sm, &smlen, m, mlen, sk),);
    printf("Signing, under new keys, new random %llu-byte messages
",mlen);
    TIME(crypto_sign(sm, &smlen, m, mlen, sk),
    crypto_sign_keypair(pk,sk),randombytes(m,mlen));
    report_stats(s,s2);
}
```bash
$--$ gcc plactic-ref.c keccak.c rng-util.c sign.c timer.c \
$--$ -o Timers/time-ref-O3 -O3

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

$--$ Timers/time-ref-O3

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

Table 14: A timing run
Timing results, under similar testing conditions, using implementation \texttt{plactic-constant.c} of plactic multiplication is 22 times slower for key generation and signing, and 35 times slower for verification.

B  Experimental utilities

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

B.1  A simplistic utility

File \texttt{ps-util.c} in Table 15 combines some standard C libraries with the plactic signature library. The message to be signed is supplied as a command-line argument, which would usually hand-typed (and quoted if it contains spaces).

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

File \texttt{ps-util-help.c} in Table 16 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 \texttt{lock} and \texttt{key} are used instead of the usual \texttt{public key} and \texttt{secret key} (or \texttt{verification key} and \texttt{signing key}) to squeeze as much information into a single screen of text.
#include <stdio.h>
#include <string.h>
#include "api.h"
#include "ps-util-help.c"
define MAX_LEN 1000000
int err(int r, char*e){
    fprintf(stderr, "psu: %s. Try ./psu --help\n", e); return r ; }
int key(void) {
    u pk[CRYPTO_PUBLICKEYBYTES], sk[CRYPTO_SECRETKEYBYTES];
    crypto_sign_keypair(pk,sk);
    fwrite(pk,1,CRYPTO_PUBLICKEYBYTES,stderr);
    fwrite(sk,1,CRYPTO_SECRETKEYBYTES,stdout); return 0;}
int sig(char *msg) {
    u sk[CRYPTO_SECRETKEYBYTES], sig[MAX_LEN];
    ll sklen,slen;
    if(0 == memcmp(msg,"--help",6)) {help(); return 6;}
    sklen=fread(sk,1,CRYPTO_SECRETKEYBYTES,stdin);
    if (sklen != CRYPTO_SECRETKEYBYTES )
        return err(2,"Bad secret key");
    crypto_sign(sig, &slen, (u*)msg, strlen(msg), sk);
    fwrite(sig,1,slen,stdout); return 0;}
int ver(char *pk_filename) {
    u pk[CRYPTO_PUBLICKEYBYTES], msg[MAX_LEN], sig[MAX_LEN];
    ll pklen,mlen,slen;
    if ( fopen(pk_filename, "r")) {
        pklen=fread(pk,1,CRYPTO_PUBLICKEYBYTES,fopen(pk_filename,"r"));
        if (CRYPTO_PUBLICKEYBYTES==pklen) {
            slen=fread(sig,1,MAX_LEN,stdin);
            if (slen >= CRYPTO_BYTES) {
                if(0 == crypto_sign_open(msg, &mlen, sig, slen, pk)){
                    fwrite(msg,1,mlen,stdout); return 0;}
                else return err(1,"Bad signature");}
            else return err(3,"Bad signature");}
        else return err(4,"Bad public key");}
    else return err(5,"Bad public key");}
int main (int c, char **a){
    return 1==c?key(): 2==c?sig(a[1]): 3==c?ver(a[1]): help();}

Table 15: File ps-util.c (key generation, signing, verifying)
```c
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"
  "Plastic signature experimental utility. Dan Brown, BlackBerry.\n"
  ); return 4;}

Table 16: File ps-util-help.c (help function)

B.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.

```
# 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.

```c
#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 argc, char **argv){
  int argc--; int termo=ttyo(), termerr=ttye();
  if(argc<1 || argc>2 || termo<0) return failure(1, "Invalid argument count");
  if( argc<1 || termo<0) return failure(1, "Invalid number of arguments");

  int sklen, pklen, sigmin, sigmax;
  if(argc<2) return failure(1, "Missing argument(s)");
  sklen = atoi(argv[1]);
  if( sklen <= 0) return failure(11, "Wrong first argument");
  pklen = atoi(argv[2]);
  if( pklen <= 0) return failure(11, "Wrong second argument");

  sigmin = atoi(argv[3]);
  if( sigmin <= 0) return failure(11, "Wrong third argument");
  sigmax = atoi(argv[4]);
  if( sigmax <= 0) return failure(11, "Wrong fourth argument");

  for(int i=0; i<argc; i++)
    if(argv[i] == "--help")
      return help(1, "Help");
  return help(0, "User error");
```

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.
return failure(33, "public key (on stderr) to terminal unsupported");
crypto_sign_keypair(pk, pk), pkread=pklen, skread=sklen;
puta(pk), put(a(pk), 1);
return success(2, "new secret key and public key generated");
if(sklen == get(pk)){
crypto_sign_keypair(pk, pk), pkread=pklen;
put(a(pk), 1);
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(0);
if(! warg(1))
return failure(101, "could not write to public key file");
crypto_sign_keypair(pk, sk), pkread=pklen, skread=sklen;
put(a(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 && termin){
geta(pk);
if(sklen == skread){
if(!argnotreadable)
return failure(55, "public key file already exists");
if(! varg(1))
return failure(56, "public key file not writable");
move(pk, sk);
crypto_sign_keypair(pk, pk), pkread=pklen, skread=sklen;
put(pk, 1);
return success(1, "computed public key from secret key");
}
if(!argnotreadable)
return failure(23, "no signed message provided");
grab(pk, sk);
if(!varg(1))
geta(sig, 1);
else {
grab(sig, pk);
geta(pk, 1);}}
if(termin && termin && argnotreadable){
geta(pk); msg=(unsigned char*)arg[1], msgread=strlen((char*)msg);}
if(termin && termin && !argnotreadable){
geta(pk);
if(sklen == skread)
geta(sig, 1);
else {
grab(sig, pk);
geta(pk, 1);}}
if0 == pkread{
if(termout)
return help(0), failure(383, "will not write signature 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,s;
if(termout){
if(termin){
if(3 < args)
return failure(222, "too many command-line arguments");
if(2 == args){
if(! rarg(1) &amp; rarg(2)){
return failure(747, "could not write a key file");
crypto_sign_keypair(pk, sk), pkread=pklen, skread=sklen;
puta(pk, 1), puta(sk, 2);
return success(1, "wrote two key files");}
if(! (rarg(1)) &amp; rarg(2))
return help(0);
if(! termout){
p=1;
geta(pk, p);
if(pklen != pkread){
return 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");}
else // 3 args ... (termin and termout)
for(s=0,i=1; i<=3; i++)
if(rarg(i)){
geta(sk, i);
if(0 == i) s=i;
}
else // s==0 (so verify with 3 args)
for(p=0,i=1; i<=3; i++)
if(rarg(i)){
geta(pk, i);
if(0 == i) p=i;
}
else // verify with 3 args
for(s=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(n=0,i=1; i<=3; i++)
    if(1 != 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 && ! rarg(i))
        geta(sig, i);
    m = 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");

for(c=0,v=0,i=1; i<=args; i++)
        if(rarg(i)){
            r+=1;
            geta(sk, i);
            if(sklen == skread)
                r = i;)
if(0 == r)
    return failure(765, "no secret key found");
if(1 == r)
    for( i=1 ; i<=args ; i++)
        if(rarg(i))
            if(! proceed())
                return failure(700, "no signing");
    geta(sk, 1); // in case sk read another file
    crypto_sign(sig, &sigread, msg, msgread, sk);
    puta(sig);
    return success(1, "non-key args signed");
if(2 == r)
    if(3 < args)
        return failure(876, "too many command-line arguments");
    geta(sig, 3-s);
    if(sigmax < sigmax)
        return failure(777, "message too long");
    if(! proceed())
        return failure(700, "no signing");
    geta(sk, 1); // in case sk read another file
    crypto_sign(sig, &sigread, msg, msgread, sk);
    puta(sig);
    return success(1, "signed file");

if(0 == skread)
    return failure(797, "too many files in command-line");
}
// ! termout
geta(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]=/quotesingle.ts1
/quotesingle.ts1;
    crypto_sign(sig, &sigread, msg, msgread, sk);
    puta(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(sigmax < sigmax)
            if(0 == crypto_sign_open(NULL,&msgread,sig,sigread, pk))
                puta(arg[i]),});})
if(pklen > pkread)
    return failure(99,"public key too small");
30
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

\[
hG = sR - rQ
\]  

(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:

\[
[a, b, c, d, e] = [h, 1/u, Q, [R, s], G].
\]  

(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
