BTOOLS: Trusted Transaction Generation for Bitcoin and Ethereum Blockchain Based on Crypto Currency SmartCard

Pascal Urien
LTCI
Telecom ParisTech
France
Pascal.Urien@telecom-paristech.fr

Mesmin Dandjinou
Ecole Supérieure d'Informatique
Université Nazi BONI
Burkina Faso
Tmesmin.dandjinou@univ-bobo.bf

Kodjo Edem Agbezoutsi
Ecole Supérieure d'Informatique
Université Nazi BONI & LTCI
Burkina Faso
Kodjo.agbezoutsi@telecom-paristech.fr

Abstract— **This paper presents an innovative and open software framework whose goal is to increase the trust of blockchain transactions. Transactions are signed by the Elliptic Curve Digital Signature Algorithm (ECDSA) associated with a 32 bytes secret private key. We designed a Javacard application used for key generation, storage and cryptographic procedure dealing with the secp256k1 elliptic curve. Our open software BTOOLS generates Bitcoin and Ethereum transactions whose trust is enforced by the support of a Crypto Currency SmartCard (CCSC).**

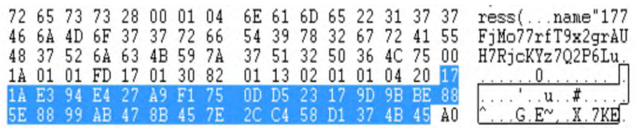
Keywords-. Blockchain; Bitcoin; Ethereum; Trust.

I. INTRODUCTION

The Bitcoin crypto currency was introduced in 2008 [1], in a famous paper written by an anonymous author Satoshi Nakamoto. This paper proposes "*a solution to the double-spending problem using a peer-to-peer distributed timestamp server to generate computational proof of the chronological order of transactions...The steady addition of a constant of amount of new coins is analogous to gold miners expending resources to add gold to circulation in our case, it is CPU time and electricity that is expended*"

Satoshi Nakamoto also wrote the win32 software Bitcoin.exe [3], about 16,000 lines of C++ code, and 6 MB binary size. This software realizes all the functions needed by the Bitcoin blockchain [2]. It manages four major tasks:

- Generation of transactions which are signed according to the *Elliptic Curve Digital Signature Algorithm*, dealing with elliptic curve private keys;
- Communication with other Bitcoin nodes running the Bitcoin application;
- Block mining;
- Blockchain management.



```
72 65 73 73 28 00 01 04  6E 61 6D 65 22 31 37 37  ress(...name"177
46 6A 4D 6F 37 37 72 66  54 39 78 32 67 72 41 55  FjMo77rfT9x2grAU
48 37 52 6A 63 4B 59 7A  37 51 32 50 36 4C 75 00  H7RjcKYz7Q2P6Lu.
1A 01 01 FD 17 01 30 82  01 13 02 01 01 04 20 17  .......0........
1A E3 94 E4 27 A9 F1 75  0D D5 23 17 9D 9B BE 88  ....'..u..#.....
5E 88 99 AB 47 8B 45 7E  2C C4 58 D1 37 4B 45 A0  ^...G.E~,.X.7KE.
```

Figure 1. The wallet.dat, a database file from Bitcoin.exe PrivateKey: 171AE394E427A9F1750DD523179D9BBE885E8899AB478B457E2CC458D1374B45. BtcAdr: 177FjMo77rfT9x2grAUH7RjcKYz7Q2P6Lu

The Bitcoin application maintains a set of data files managed by a non Structured Query Language (SQL) database, the *Berkeley Database (Berkeley DB)* [14]. In particular, the private keys are stored in the file named wallet.dat. As illustrated in Figure 1, private keys are stored in clear text in the database file.

Because all crypto currency legitimate transactions rely on private keys, their secure storage and trusted use is a major prerequisite for blockchain operations. As an illustration, the Korean Exchange *Youbit* declared bankruptcy in December 2017 after the hacking of 17% of its Bitcoin reserves, about 4,700 Bitcoins [17].

Our researches attempt to increase trust of blockchain operations, by using secure elements, enforcing secure key storage and trusted ECDSA signature. In order to reach this goal, we developed the BTOOLS (*Blockchain Tools*) open software [12], able to generate *Bitcoin* or *Ethereum* transactions, whose signature is computed by a dedicated *Crypto Currency SmartCard*, i.e. a Javacard running a Java application.

BTOOLS uses OPENSSL library and smartcard, for cryptographic operations. It provides the following services:

- Bitcoin address generation (mainnet and testnet);
- Ethereum address generation;
- Bitcoin transaction generation;
- Ethereum transaction generation;
- Simple Bitcoin node client;
- Bitcoin transaction (via the Bitcoin client or WEB APIs);
- Ethereum transaction (via WEB APIs);
- Crypto Currency SmartCard scripts for key generation and transaction signature.

The paper is constructed according to the following outline. Section 2 recalls basic notions for the generation of ECDSA signatures over elliptic curves. Section 3 details Bitcoin transactions and dedicated BTOOLS scripts. Section 4 describes Ethereum transactions and BTOOLS dedicated scripts. Section 5 introduces Crypto Currency SmartCard and its use with BTOOLS software. Finally, Section 6 concludes this paper.

II. About the ECDSA Signature

Most crypto moneys (Bitcoin, Ethereum...) use the secp256k1 elliptic curve, whose parameters are as follow [4]:

- The p prime characteristic of the field Z/pZ, defined as:

$$p = 2^{256} + 2^{32} + 2^9 + 2^8 + 2^7 + 2^6 + 2^4 + 1$$

- The elliptic curve E defined as the set of points (x,y) satisfying the relation:

$$y^2 = x^3 + 7,\ x,y \in Z/pZ$$

- The generator G uncompressed (i.e. x and y) form which is:

```
04
79BE667E F9DCBBAC 55A06295 CE870B07
029BFCDB 2DCE28D9 59F2815B 16F81798
483ADA77 26A3C465 5DA4FBFC 0E1108A8
FD17B448 A6855419 9C47D08F FB10D4B8
```

- The n order (i.e. the number of group elements) of the curve defined as:

```
FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE
BAAEDCE6 AF48A03B BFD25E8C D0364141
```

- Finally, the cofactor is equal to 1, which means that there is only one group in E whose order is the prime n.

A. ECDSA Signature

An ECDSA signature over E [5] is a couple of two integers (r, s) such as :

Given x Є [1, n-1] the private key i.e. a 32 bytes random number, P= xG is the public key.

k is an ephemeral key, k Є [1, n-1]

kG= (x_R, y_R), and r = x_R mod n

size = number of bytes of n (size = 32)

H a hash function, e = H(M) is the hash of a message M, i.e. a set of bytes to be signed.

If H(M) has more bytes than size, then take e as the size leftmost bytes.

The couple (r, s), with $s = k^{-1}\ (e + x\ r) \bmod n$ is the signature.

B. ECDSA Signature Verification

Let the signature being (r, s).

Given the message M and H, compute e = H(M).

size = number of bytes of n (size =32). If e has more bytes than size, take the size leftmost bytes.

1) Compute $u_1 = es^{-1}$ mod n and $u_2 = rs^{-1}$ mod n.

2) Compute $R = (x_R, y_R) = u_1G + u_2P$.

3) Set $v = x_R$ mod n.

Compare v and r, and if v = r the signature is valid.

C. Canonical Signature

For a given ECDSA signature, (r, s), the signature (r, n-s) is also valid. The canonical signature is computed according to the following algorithm:

1) Compute n-s =t.

2) If s < t, then (r, s) is canonical signature.

3) Otherwise, (r, t) is the canonical signature.

Bitcoin and Ethereum blockchains request canonical signature.

D. Public Key Recovery from ECDSA Signature

Given the ECDSA [5] signature (r, s).

Find the *"positive"* point R(x=r, y=y+) on the E: $y^2=x^3+7$ curve,

Given the message M and H, compute e= H(M).

size = number of bytes of n (32). If e has more bytes than size, take the size leftmost bytes.

Compute the candidate public key $Q = r^{-1}(sR - eG)$.

Check the signature (r, s), and if valid set recovery to 27 in Ethereum.

If not verified, try with the *"negative"* point –R= (x=r, y=y-), and if valid, set recovery to 28 in Ethereum.

III. Bitcoin transactions

A. Bitcoin Address

Bitcoin addresses (BA) are computed from ECDSA public key. A private key, i.e. a 32 byte number x, is generated, according to a true random number generator (TRNG). Thereafter, a public key is computed according to the relation P = xG. The uncompressed form uF(P) is a set of 65 bytes $\{4, x_P, y_P\}$, a prefix (one byte 0x04) and a point (x_P, y_P) of the curve (2x32 bytes, in Z/pZ).

The Bitcoin address [2] is computed according to the following procedure:

1) a1 = SHA256(u_F(P)), 32 bytes

3) hash160= a2 = RIPEMD160(a1), 20 bytes

3) a3 = Network-ID || a2, 25 bytes

4) a4 = SHA256(SHA256(a3)), 32 bytes

5) a5= checksum = 4 rightmost bytes of a a4

6) a6 = a4 || a5, 25 bytes

7) Bitcoin address = a7 = encoding of a6 in base 58

The base 58 encoding uses the following digits {1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G, H, J, K, L, M, N, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, m, n, o, p, q, r, s, t, u, v, w, x, y, z}.

A BA is protected by a four bytes checksum; although the hash160 parameter has no checksum, it is used in transactions as payee's address.

Figure 2 illustrates the generation of Bitcoin address by BTOOLS.

```
btools -genmain
PrivateKey:
CE1DBAFD7D2E8983ED60E0E081632EB062737B1B1627AAAB276F2E037
A74A081
PublicKey:
04CFD7A542B8C823992AF51DA828E1B693CC5AB64F0CACF0F80C31A1E
CA471786E285BDD3F1FE0A006BD70567885EF57EB149C8880CB9D5AF3
04182AC942E176CC
Hash160: CB643DD608FB5C323A4A6342C1A6AC8048B409EB
BTC-Adr: 1KYSFr6CyTDMruu8wna981M4ziVyMwftcg
Double SHA2 Check OK
ID: 00
Hash160: CB643DD608FB5C323A4A6342C1A6AC8048B409EB
BTC-WIF:
5KP4YMxDzfv9P1WVAPZqHRSfi5FydGqqqRjr5oPvskpwTq59wiX
```

Figure 2. Generation of a Bitcoin address by the BTOOLS software

B. *Bitcoin Transaction*

A transaction is a list of inputs, associated to Bitcoin amounts (i.e. coins), and a list of outputs to which are transferred the totally of inputs. A fee is allocated to the miner if the sum of outputs is less than the sum of inputs.

A fee is usually expressed in satoshi per byte (1 satoshi = 10^{-8} Bitcoin (BTC)); since July 2017 it is expressed in weight units/byte. At the time of writing, the fee was ranging between 50,000 and 100,000 satoshi (0,0005 to 0,001 BTC).

The structure of a transaction is detailed in Figure 3.

In every input, a coin value for *Unspent Transaction Output* (UTXO) is identified by a previous transaction identifier and its output index (starting from 0). A transaction ID is equal to the double SHA256 hash of the binary content of the transaction. A signature script (*sigScript*) contains the ECDSA signature and public key of the payer's transaction.

Every output comprises an amount expressed in satoshi, and a public key script (*pubKeyScript*) including the payee's hash160 address.

Parameter	**Type**	**Comment**
version	integer 32 bits	always 1
number of inputs	var_int 1 byte or more	
	One or more inputs	
transactionID	32 bytes	coin transaction
index	integer 32 bits	coin index >=0
sigScript length	1 byte	
sigScript	contains the signature and the public key	
sequence	integer 32 bits FFFFFFFF=ignore	transaction version
	End of input	
number of outputs	var_int 1 byte or more	
	One or more outputs	
value	integer 64bits	satoshi amount
pubKeyScript length	1 byte	
pubKeyScript		
locktime	integer 32 bits 00000000=ignore	transaction locktime

Figure 3. Structure of a Bitcoin transaction

```
01000000 // Version
01 // number of inputs
DE2D211EF429909B0AB8D2E7D25826A0 //TransactionID
EDD6281EC6DEDF2B822CE5014A349E72
01000000 // index
8A // length of the signature Script
47 // ECDSA Signature length
30 44 // Sequence of (r, s) integer values
02 20 // integer r value
0772ABD5D37D0CAAB881DBC8912628F9
3461839CC8D4BC007A355831A6061ED7
02 20 // integer s value
4CCCC34B34A9075FC09C9777EAB7A6F5
612DA2130C1FF1C0E376AD9B2209D51D
01 41 // Public key length
04 // uncompressed format
CFD7A542B8C823992AF51DA828E1B693
CC5AB64F0CACF0F80C31A1ECA471786E
285BDD3F1FE0A006BD70567885EF57EB
149C8880CB9D5AF304182AC942E176CC
FFFFFFFF // sequence
01 // number of outputs
D418040000000000 // amount in satoshi
19 // Public Key Script
76 // OP_DUP
A9 // OP_HASH160
14 // hash160 length
CB643DD608FB5C323A4A6342C1A6AC8048B409EB
88 // OP_EQUALVERIFY
AC // OP_CHECKSIG
00000000 // Locktime
```

Figure 4. Binary encoding of a Bitcoin transaction

Figure 4 presents a binary dump of a transaction using a *pay-to-pubkey-hash* script; it should be noticed that all values are encoded according the a *little endian* format.

The *pay-to-pubkey-hash* script is defined as:

OP_DUP [76] OP_HASH160 [A9]
<length=14><hash160>
OP_EQUALVERIFY[88] OP_CHECKSIG[AC]

The ECDSA signature is encoded using the following ASN.1 structure (see for example RFC 3279 [15]):

Ecdsa-Sig-Value ::= SEQUENCE {
r INTEGER,
s INTEGER }

```
01000000 // Version
01 // number of inputs
DE2D211EF429909B0AB8D2E7D25826A0 // Transaction ID
EDD6281EC6DEDF2B822CE5014A349E72
01000000 // index
00 // vi= length of the Signature Script
FFFFFFFF // sequence
01 // number of outputs
D418040000000000 // amount in satoshi
19 // Public Key Script (Pk script)
76 // OP_DUP
A9 // OP_HASH160
14 // hash160 length
CB643DD608FB5C323A4A6342C1A6AC8048B409EB
88 // OP_EQUALVERIFY
AC // OP_CHECKSIG
00000000 // Locktime
01000000 // hash Type
```

Figure 5. Binary dump of a raw Bitcoin transaction

The signature computing is performed according to the following procedure:

1) Build a raw transaction (see Figure 5), in which, for every input, the sigScript is removed, i.e. the length value (vi) is set to zero.

2) For every input:

2.1) Copy the *pay-to-pubkey-hashScript* in the *sigScript* location, and modify the length (initially set to 0) accordingly (length =25 in decimal).

2.2) The hash160 inserted in *pay-to-pubkey-hash* is computed from the payer's public key.

2.3) Compute the double SHA256 of the modified transaction.

2.4) Generate the ECDSA signature with the payer's private key.

2.5) Insert the final *sigScript* in the input, and modify the length accordingly.

C. BTOOLS Bitcoin Script

```
sequence ffffffff
locktime 00000000

nb_input 1

input
transaction 729E344A01E52C822BDFDEC61E28D6ED
A02658D2E7D2B80A9B9029F41E212DDE
index 1
privkey CE1DBAFD7D2E8983ED60E0E081632EB0
62737B1B1627AAAB276F2E037A74A081
// APDU_script sAPDU.txt

nb_output 1

output
fee 0.0005
btc 0.002685
hash160 CB643DD608FB5C323A4A6342C1A6AC8048B409EB
```

Figure 6. A Bitcoin transaction script in BTOOLS

Bitcoin transactions are generated thanks to a script; the Crypto Currency SmartCard can be used to compute the ECDSA signature.

A script is a set of lines. A comment line begins by the '/' or '*' character. It defines *sequence* and *locktime* values (in hexadecimal Most Significant Bit (MSB) encoding).

The number of inputs is specified by the *nb_input* field. Each input must begin by the input field; it comprises:

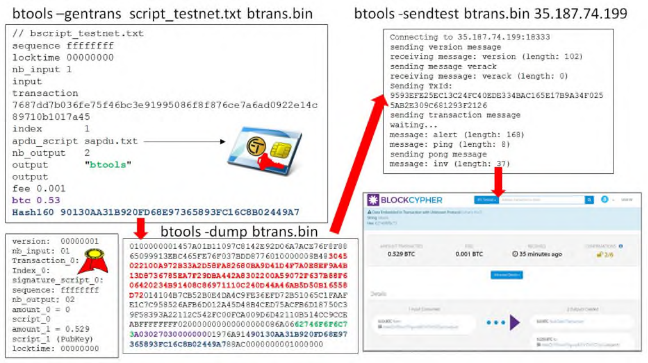
- a transaction identifier (32 bytes, hexadecimal MSB encoding);
- an index (decimal encoding);
- and a choice between the following fields :
  - privkey [private key hexadecimal MSB encoding],
  - wif [WIF],
  - APDU_script [the name of a smartcard script].

The number of outputs is specified by the *nb_output* field. Each output must begin by the output field; it comprises:

- an optional fee in decimal format, to be subtracted from the BTC (i.e. UTXO in most case) value of the current output; the character '.' is used as decimal separator;
- a BTC amount in decimal format, the character '.' is used as decimal separator;
- and a choice between the following fields:
  - adr [Bitcoin address],
  - hash160 [hash160, hexadecimal MSB encoding].

A Bitcoin transaction script is detailed in Figure 6.

D. Sending transaction to the Bitcoin blockchain


btools –gentrans script_testnet.txt btrans.bin
btools -sendtest btrans.bin 35.187.74.199
btools -dump btrans.bin
BLOCKCYPHER
0.529 BTC
0.001 BTC
35 minutes ago

```
// bscript_testnet.txt
sequence ffffffff
locktime 00000000
nb_input 1
input
transaction
7687dd7b036fe75f46bc3e91995086f8f876ce7a6ad0922e14c
89710b1017a45
index       1
apdu_script sapdu.txt
nb_output   2
output      "btools"
output
fee 0.001
btc 0.53
Hash160 90130AA31B920FD68E97365893FC16C8B02449A7
```

```
Connecting to 35.187.74.199:18333
sending version message
receiving message: version (length: 102)
sending message verack
receiving message: verack (length: 0)
Sending TxId:
9593EFE25EC13C24FC40EDE334BAC165E17B9A34F025
5AB2E309C681293F2126
sending transaction message
waiting...
message: alert (length: 168)
message: ping (length: 8)
sending pong message
message: inv (length: 37)
```

```
version:  00000001
nb_input: 01
Transaction_0:
Index_0:
signature_script_0:
sequence: ffffffff
nb_output: 02
amount_0 = 0
script_0
amount_1 = 0.529
script_1 (PubKey)
locktime: 00000000
```

```
0100000001457A01B11097C8142E92D06A7ACE76F8F88
65099913EBC465FE76F037BDD877601000000B8B483045
022100A972B33A2D58FA82680BA9D41D4F7A0E8EF9A4B
13D8736785EA7F29DBA442A8302200A59072F637B88F6
06420234B91408C86971110C240D44A46AB5D50B16558
D72014104B7CB52B0E4DA4C9FE36EFD72B51065C1FAAF
E1C7C958526AFB6D012A45D48B4CED75ACFB6D18750C3
9F58393A22112C542FC00FCA009D6D42110B514CC9CCE
ABFFFFFFFF020000000000000000086A0662746F6F6C7
3A03027030000000001976A91490130AA31B920FD68E97
365893FC16C8B02449A788AC00000000001000000
```

Figure 7. Using BTOOLS for sending Bitcoin transaction

1) Bitcoin protocol

The Bitcoin blockchain supports a protocol running over the TCP port 8333. Some Web sites list the Bitcoin nodes available over the world, for example:

https://bitnodes.earn.com/

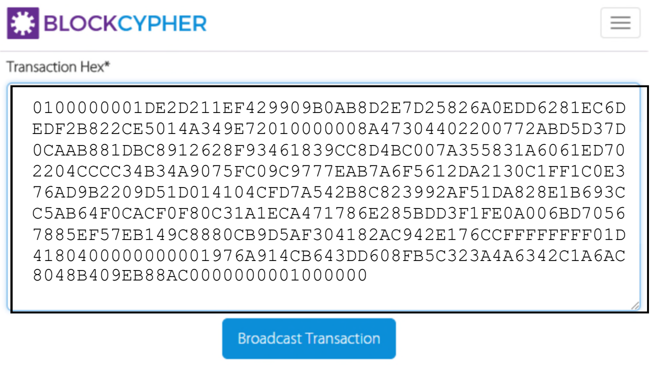
The structure of Bitcoin messages is detailed in [7][10]. The connection to a Bitcoin node requires a four way handshake, client and server exchange two *version* messages and their acknowledgment (*verack*). Afterwards, the transaction is forwarded thanks to the *tx* message.

As illustrated in Figure 7, BTOOLS realizes these operations according to the command line:

btools -sendmain transaction.bin BitcoinNode

2) Web APIs

Many full Bitcoin nodes support WEB interfaces and associated APIs. As illustrated in Figure 8 the hexadecimal representation of the transaction can be simply cut and paste in a dedicated HTML form.



```
0100000001DE2D211EF429909B0AB8D2E7D25826A0EDD6281EC6D
EDF2B822CE5014A349E72010000008A47304402200772ABD5D37D
0CAAB881DBC8912628F93461839CC8D4BC007A355831A6061ED70
2204CCCC34B34A9075FC09C9777EAB7A6F5612DA2130C1FF1C0E3
76AD9B2209D51D014104CFD7A542B8C823992AF51DA828E1B693C
C5AB64F0CACF0F80C31A1ECA471786E285BDD3F1FE0A006BD7056
7885EF57EB149C8880CB9D5AF304182AC942E176CCFFFFFFFF01D
418040000000000001976A914CB643DD608FB5C323A4A6342C1A6AC
8048B409EB88AC0000000001000000
```

Figure 8. Sending Bitcoin transaction thanks to the WEB interface https://live.blockcypher.com/btc/pushtx

IV. ETHEREUM

A. *Ethereum Address*

Ethereum addresses (EA) are computed from ECDSA public keys [8][9] . A private key, i.e. a 32 byte number x, is generated, according to a true random number generator (RNG). Thereafter a public key is computed according to the relation P=xG. The uncompressed form u'F(P) is a set of 64 bytes $\{x_P,y_P\}$, i.e. the point (x_P,y_P) of the curve (2x32 bytes, in Z/pZ).

The Ethereum address is computed according to the following procedure:

- Compute a1= Keccak(u'F(P)), a 32 byte value. SHA3 is this a subset the Keccak [13] algorithm.

- Extract a2, the 20 rightmost bytes of a1; a2 is the Ethereum address

Figure 9 illustrates the generation of Ethereum address by BTOOLS.

```
btools -geneth

PublicKey:
0477AAA9AE8ADCAAA26F930D6022E470BBC16E10AF22A5482DAB0798A
5A2C2AF52581076023A8B33D8BA6F8E7E89EC1C5F0D66B1EFFC744582
AF063187297592F6
PrivateKey:
E49344BD32802138C9A250FCEA13F6AE30E17BC945F107F05618AFC0E
D523042
Ether Address: 777A07BAB1C119D74545B82A8BE72BEAFF4D447B
```

Figure 9. Generation of Ethereum address by the BTOOLS software

B. *Ethereum Transaction*

A transaction encodes the transfer of ethers or data between two entities, identified by their address. It includes the following fields:

- The **recipient**'s address of the message
- **nonce**, a scalar value equal to the number (>=0) of transactions generated by the sender.
- **value**, a scalar value equal to the number of Wei (1 Wei=10^{-18} Ether) to be transferred to the message recipient, or in the case of contract creation, as an endowment for the newly created account.
- A **gasLimit** value, representing the maximum number of computational steps that the transaction execution is allowed to take.
- A **gasPrice** value, representing the fee the sender pays per computational step. A scalar value equal to the number of Wei to be paid per unit of gas.
- An optional **data** field. A contract creation transaction contains an unlimited size byte array specifying the EVM (*Ethereum Virtual Machine*) code for the account initialization procedure. A message call transaction contains an unlimited size byte array specifying the input data of the message.
- The ECDSA **signature**, used to identify the sender.

C. *RLP encoding*

All transaction attributes are encoding according [11] to the RLP (*Recursive Length Prefix*) syntax, which supports *string* and *list* items.

1) *String encoding*

A *string* is a byte array, it is encoded according to the following rules :

- for one byte ϵ [0x00 0x7F] : a byte value
- if the string length ϵ [0,55] : 0x80 + Length ϵ [0x80, 0xb7] || ByteArray[Length]
- 0x80: = NULL String
- if the string Length >55 : 0xb7 + Length-of-Lengh ϵ [0xb8, 0xbf] || Length-value || ByteArray[Length]

2) *List encoding*

A list is a set of items, either *list* or *string*.

- if the list Length <=55 : 0xc0 + Length ϵ [0xc0, 0xf7] || ListItems.
- if the list Length > 55 : 0xf7 + Length-of-Length ϵ [0xf8, 0xff] || Length-value || ListItems.

D. *Example of transaction*

```
F8 6B  // list length= 107 bytes
80 // nonce = null (zero value)
85 04E3B29200 // gazPrice= 21,000,000,000 Wei)
82 9C40 // gazLimit= 40,000 Wei
94 777A07BAB1C119D74545B82A8BE72BEAFF4D447B //Recipient
87 2386F26FC10000 // value= 10,000,000,000,000,000 Wei
80 // data = null
1C // signature recovery parameter = 28
A0 F1DD7D3B245D75368B467B06CAD61002 // r value
67031935B7474ACB5C74FE7D8C904097 // 32 bytes
A0 772D65407480D7C45C7E22F84211CB1A // s value
DF9B3F36046A2F93149135CADBB9385D // 32 bytes
```

Figure 10. Binary dump of an Ethereum transaction

Transaction values are expressed according to a *Big Endian* scheme. A transaction (illustrated in Figure 10) is a list of strings, encoded with the RLP syntax. The six transaction items (*nonce*, *gasPrice*, *gasLimit*, *recipient address*, *value*, *data*), are followed by the ECDSA signature dealing with a recovery value. The recovery value is used for the recovery of the sender's public key.

E. *Ethereum Raw Transaction*

```
E8 80 // list length = 40 bytes
80 // nonce = null (zero value)
85 04E3B29200 // gazPrice= 21,000,000,000 Wei)
82 9C40 // gazLimit= 40,000 Wei
94 777A07BAB1C119D74545B82A8BE72BEAFF4D447B //Recipient
87 2386F26FC10000 // value= 10,000,000,000,000,000 Wei
80 // data = null
```

Figure 11. Example of a raw Ethereum transaction

A raw transaction (see Figure 11) is the list of six items (*nonce*, *gasPrice*, *gasLimit*, *recipient address*, *value*, *data*), without the signature elements. The ECDSA signature is performed over this structure. The recovery parameter (either 0 or 1) is added to the 27 decimal value, and is needed for the extraction of the sender public key.

F. *BTOOLS script for Ethereum transaction*

A transaction script is a set of lines (see Figure 12). A comment line begins by the '/' or '*' character.

The file (see Figure 12) comprises the following elements:

- the *private key* (**privkey**) or the name of a *smartcard script* (**APDU_script**) The Bitcoin and Ethereum smartcard scripts follow the same syntax.
- the ***nonce*** field. The nonce is expressed in decimal format.
- the ***gasPrice*** field. The gasPrice, in WEI unit.
- the ***gasLimit*** field. The gasLimit, in WEI unit.
- the **to** field indicates the ether destination address. It is a 20 bytes hexadecimal value.
- the ***value*** field indicates the transaction amount, in WEI unit.
- the ***data*** field. Three options are available:
  - **data**, text (ASCII) data field
  - **datab**, hexadecimal data field
  - **dataf**, a binary file

```
privkey E49344BD32802138C9A250FCEA13F
6AE30E17BC945F107F05618AFC0ED
523042
// APDU_script sAPDU.txt

nonce     0
gasPrice  21000000000
gasLimit  40000
to 777A07BAB1C119D74545B82A8BE72BEAFF4D447B
value     100000000000000000
data
```

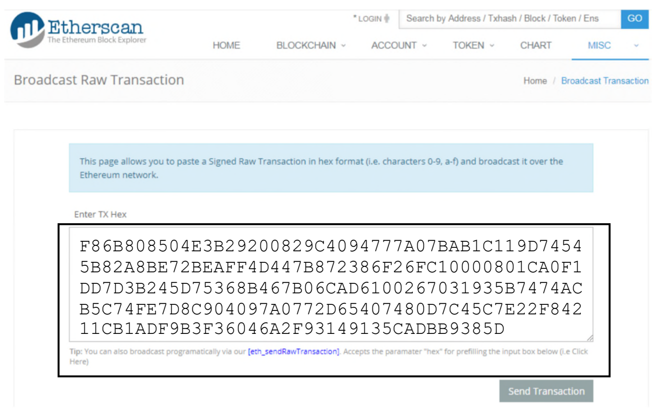
Figure 12. Illustration of an Ethereum transaction script in BTOOLS

G. *Sending a transcation to the Ethereum blockchain*

The Ethereum blockchain supports a protocol running over the TCP port 30303. Some Web sites list the Ethereum nodes available over the world, for example:

https://www.ethernodes.org

The today BTOOLS software doesn't implement the Ethereum protocol. Nevertheless many full Ethereum node support WEB interfaces and associated APIs. As illustrated in Figure 13 the hexadecimal representation of the transaction can be simply cut and paste in a dedicated HTML form.


Etherscan
The Ethereum Block Explorer
LOGIN
Search by Address / Txhash / Block / Token / Ens
GO
HOME
BLOCKCHAIN
ACCOUNT
TOKEN
CHART
MISC
Home / Broadcast Transaction
Send Transaction

Broadcast Raw Transaction

This page allows you to paste a Signed Raw Transaction in hex format (i.e. characters 0-9, a-f) and broadcast it over the Ethereum network.

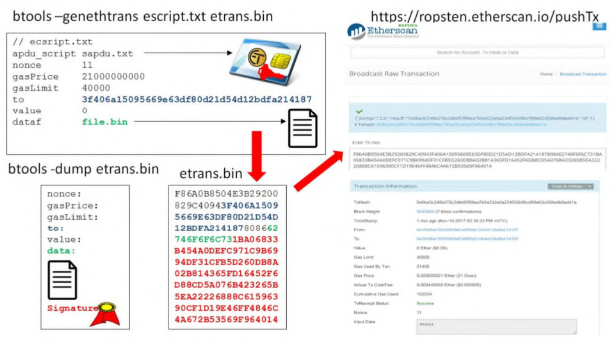
Enter TX Hex

```
F86B808504E3B29200829C4094777A07BAB1C119D7454
5B82A8BE72BEAFF4D447B872386F26FC10000801CA0F1
DD7D3B245D75368B467B06CAD6100267031935B7474AC
B5C74FE7D8C904097A0772D65407480D7C45C7E22F842
11CB1ADF9B3F36046A2F93149135CADBB9385D
```

Tip: You can also broadcast programatically via our [eth_sendRawTransaction]. Accepts the paramater "hex" for prefilling the input box below (i.e Click Here)

Figure 13. Sending an Ethereum transaction thanks to a Web API on the website https://etherscan.io/pushTx

Figure 14 illustrates an Ethereum transaction generation and forwarding thanks to BTOOLS software facilities.


https://ropsten.etherscan.io/pushTx
Etherscan
Broadcast Raw Transaction
Transaction Information

btools –genethtrans escript.txt etrans.bin

```
// ecsript.txt
apdu_script sapdu.txt
nonce       11
gasPrice    21000000000
gasLimit    40000
to          3f406a15095669e63df80d21d54d12bdfa214187
value       0
dataf       file.bin
```

btools -dump etrans.bin

```
nonce:
gasPrice:
gasLimit:
to:
value:
data:
Signature
```

etrans.bin

```
F86A0B8504E3B29200
829C40943F406A1509
5669E63DF80D21D54D
12BDFA214187808662
746F6F6C731BA06833
B454A0DEFC971C9B69
94DF31CFB5D260DB8A
02B814365FD16452F6
D88CD5A076B423265B
5EA22226888C615963
90CF1D19E46FF4846C
4A672B53569F964014
```

Figure 14. Illustration of an ether transaction generation with BTOOLS

V. CRYPTO CURRENCY SMARTCARD (CCSC)

The Crypto Currency SmartCard application (CCSC), illustrated in Figure 15, is written in Javacard, a subset of the Java language. It has three PINs: administrator, user, and user2. The default values are 8 zeros (3030303030303030) for administrator and 4 zeros (30303030) for user and user2. It is able to generate or to import elliptic curve keys (up to 8), used for the generation of ECDSA signatures used by Bitcoin and Ethereum crypto currencies. A Read/Write non volatile memory, protected by a dedicated PIN (User2), is available for the storage of any sensitive information.

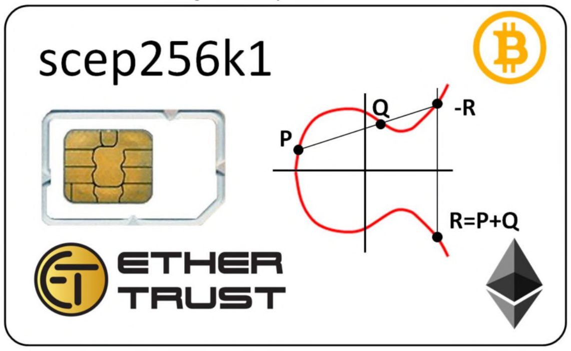


Figure 15. Illustration of a Crypto Currency SmartCard (CCSC)

The CCSC application main ISO7816 services are the following: Init Curve, Clear Key Pair, Generate Key Pair, Get Key Parameters, Set Key Parameters, Sign ECDSA.

A. *The CCSC ISO7816 interface*

According to the ISO7816-4 standard [16], a smartcard command, also called *Application Protocol Data Unit* (APDU), comprises at least five bytes named CLA, INS, P1, P2, P3; P3 is the length of data to be written or the length of information to be read. The response comprises an optional payload (up to 256 bytes) and two status bytes (SW1, SW2). The available commercial version of Javacard is 3.0.4, which API framework supports elliptic curve facility, in particular the *secp256k1* curve, and the ECDSA signature. The ISO7816 interface of the CCSC application is detailed in Figure 16.

Command	ISO7816 encoding	Comment
Select	00A4040006<AID> AID= Application IDentifier=010203040500	Start the CCSC application
Verify	0020000004<UserPIN> 0020000204<User2PIN> 0020000108<AdminPIN>	Check PIN
InitCurve AdminPIN is required	008900$P_2$00 P2 is the key index	Init curve parameters
ClearKeys AdminPIN is required	008100$P_2$00 P2 is the key index	Clear public and private keys,
GenKeys AdminPIN is required	008200$P_2$00 P2 is the key index	Generate the keys
SetKey AdminPIN is required	008800$P_1P_2P_3$<value> P2 is the key index P1=6 for the public key P1=7 for the private key P3 is the key length Value is the key value	Set public or private key The public key is in the uncompress format
GetKey UserPin is required	008400$P_1P_2P_3$<value> P2 is the key index P1=6 for the public key P1=7 for the private key AdminPIN is required for the private key	Get public or private key, return the length (16bits) of the key and its value
SignECDSA UserPIN is required	008000P_2P_3<value> P2 is the key index P3 is the length of the hash to be signed (32) value is the hash (e)	Return the length (16bits) of the ECDSA signature and its ASN.1 encoding

Figure 16. ISO7816 interface of the CCSC application

The cryptographic keys can be generated and optionally exported, or imported.

- The procedure for key generation and export deals with the following commands: Select(AID), Verify(AdminPIN), InitCurve, ClearKeys, GenKeys, GetKeys.
- The procedure for key import uses the following commands: Select(AID), Verify(AdminPIN), ClearKeys, InitCurve, SetKey(PublicKey), SetKey(PrivateKey).

The ECDSA signature is performed according to the following sequence: Select(AID), Verify(UserPIN), GetKey(PublicKey), SignECDSA(HashValue).

B. *BTOOLS APDU script*

The *BTOOLS* software manages APDU script in order to communicate with Crypto Currency SmartCards. It is a set of lines. A comment line begins by the '/' or '*' character.

The main script token are as follow:

- ***start <optional AID>*** which initializes the ISO7816 context, and detects the first available smartcard;

- **APDU <hexadecimal value>** which sends an ISO7816 request to the smartcard. For error free operation, the response should end by the 9000 status;

- ***pub <offset>*** which MUST be specified before the APDU command used to collect the public key. It is the offset in the response of the public key (after the byte 04);

- ***signature <offset>*** which MUST be specified before the APDU command used to collect the signature. It is the offset in the response of the ASN.1 encoding of the ECDSA signature;

- ***hash <offset>*** which MUST be specified before the APDU command used to collect the signature. It is the offset in the ISO7816 request of the hash (or data) to be signed.

Figures 17 and 18 give an example of APDU script, dealing with a pair of keys identified by the index 5.

```
// script file name: sAPDU.txt
start
// Select CCSC
APDU 00A4040006 010203040500
// Verify UserPIN= 0000
APDU 0020000004 30303030
// Get PublicKey index=5
pub 3
APDU 0084 0605 43
// ECDSA Signature, index=5
signature 2
hash 5
APDU 0080 0005 20
```

Figure 17. An APDU script use for the generation of ECDSA signature

```
// start
Opening the APDU script sAPDU.txt
Reader: Broadcom Corp Contacted SmartCard 0
T=0  - ATR
// Select(CCSC)
Tx: 00 A4 04 00 06 01 02 03 04 05 00
Rx: 90 00
// Verify(UserPIN)
Tx: 00 20 00 00 04 30 30 30 30
Rx: 90 00
// GetKey(PublicKey)
Tx: 00 84 06 05 43
Rx: 00 41 04 A6 FC 0C 5F 46 7C 3D B8 C1 58 18 05 E7
C6 2C 5F AE A1 90 63 B0 1F 58 45 AD 68 DE 9D 84
38 5F 32 1E BF 3A 26 B2 99 12 41 89 92 DC DC 1F
E6 9C 28 2E FF 65 86 0E 10 9F 53 AD 27 A2 96 24
98 4B 6A 90 00
// SignECDSA(hash)
Tx: 00 80 00 05 20 DC AF B4 6D 7F 57 1D 87 C2 34 B3
20 8E 68 86 AD F4 85 AC 98 20 EA A5 67 7C 6D 37 6A
32 13 6F 34
Rx: 61 48
Tx: 00 C0 00 00 48
Rx: 00 46 30 44 02 20 65 A3 1E 14 88 20 61 82 1E A8
B7 27 C4 A8 D1 E2 CB 59 29 20 88 6B DD 70 84 B9
C1 C5 D6 6F 7D 30 02 20 5B 83 A4 69 E5 6D 3B B1
C2 77 6B 16 A3 7B C1 19 0F 6A C9 85 F7 03 54 B6
58 1B 6F 46 21 C7 63 3B 90 00
```

Figure 18. An APDU script used by a transaction script

In Figure 18, the public key is in blue characters, the value to sign in bold characters, and the ASN.1 signature encoding in red characters.

BTOOLS also provides an option that starts APDU scripts, typically used for used key generation.

Figure 19 gives an example of such a script, and Figure 20 illustrates its execution.

```
start
// select
APDU 00A4040006 010203040500
// Verify PinAdmin
APDU 0020 0001 08 3030303030303030
// ClearKeys Key 0
APDU 0081 00 00 00
// InitCurve, Key 0
APDU 0089 00 00 00
// Generate KeysPair Key 0
APDU 0082 00 00 00
// GetPublicKey Key0
APDU 0084 06 00 00
// GetPrivateKey Key 0
APDU 0084 07 00 00
```

Figure 19. Example of a script used for key generation

```
// select
Tx: 00 A4 04 00 06 01 02 03 04 05 00
Rx: 90 00
//Verify(AdminPIN)
Tx: 00 20 00 01 08 30 30 30 30 30 30 30 30
Rx: 90 00
Tx: 00 81 00 00 00 // Clear Key index 0
Rx: 90 00
Tx: 00 89 00 00 00 // Init curve index 0
Rx: 90 00
Tx: 00 82 00 00 00 // Generate Keys index 0
Rx: 90 00
Tx: 00 84 06 00 43 // Get Public Key index0
Rx: 00 41 04 BA 5A 71 A8 0E 90 76 9E DD D2 B9 6C B4
BA 47 0B 45 C6 3B 01 F5 A9 FB FC 3F 95 37 43 23
18 15 5D 59 F3 F1 75 26 08 4E 5A CC 7D 17 4D 68
AB 39 57 C4 F6 D8 5D 38 43 95 EF 8D F4 7D 05 3B
FE E6 F9 90 00
Tx: 00 84 07 00 00 // Get Private Key index 0
Rx: 6C 22
Tx: 00 84 07 00 22
Rx: 00 20 85 1F 6D 62 0B 87 FC 27 FC 9A 00 42 8F C6
01 37 D8 6B 14 07 E4 B6 8F 77 30 A4 BF AC CE 7D
A3 91 90 00
```

Figure 20. Illustration of a key generation script at run time. The public key is in blue characters. The private key is in red characters.

VI. CONCLUSION

In this paper we present the BTOOLS open software [12] that targets the generation of trusted blockchain transactions, based on smartcard cryptographic services. BTOOLS is available for Win32, Linux or Raspberry PI environments. Our future projects will address the definition of innovative services based on this trusted platform.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", www.bitcoin.org, 2008, [retrieved: June, 2017]

[2] A. M. Antonopoulos, "Mastering Bitcoin", O'REILLY, 2015

[3] P. Huang, "A Dissection of Bitcoin", ISBN 9781329754812, January 2016

[4] Standards for Efficient Cryptography "SEC 2: Recommended Elliptic Curve Domain Parameters", Certicom Research, January 27, 2010 Version 2.0

[5] Standards for Efficient Cryptography, "SEC 1: Elliptic Curve Cryptography", Certicom Research, May 21, 2009, Version 2.0

[6] https://github.com/Bitcoin, [retrieved: July, 2017]

[7] https://en.Bitcoin.it/wiki/Protocol_documentation#Message_structure [retrieved: July, 2017]

[8] V. Buterin, "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform", 2013, http://Ethereum.org/Ethereum.html, [retrieved: July, 2017]

[9] G. Wood, Ethereum Yellow Paper, "Ethereum : a Secure Decentralized Generalized Transaction Ledger", EIP-150, 2015, http://yellowpaper.io/, [retrieved: July, 2017]

[10] https://qbitninja.docs.apiary.io/#reference/transactions/retrieve-a-transaction/get, [retrieved: June, 2017]

[11] https://github.com/Ethereum/wiki/wiki, [retrieved: June, 2017]

[12] "BTOOLS, blockchain tools", https;//github.com/purien/btools, , [retrieved: November 2017]

[13] G. Bertoni, J. Daemen, M. Peeters, G. Assche, "The Keccak SHA-3 submission", 2011, http://keccak.noekeon.org/Keccak-submission-3.pdf, [retrieved: July, 2017]

[14] M. A. Olson, K. Bostic, M. Seltzer, "Berkeley DB", Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, Monterey, California, USA, June 6-11, 1999.

[15] W. Polk, R. Housley, L. Bassham, "Algorithms and identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) profile", RFC 3279, April 2002.

[16] ISO/IEC7816-4:2013, "Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange", 2013

[17] https://www.reuters.com/article/us-bitcoin-exchange-southkorea/south-korean-cryptocurrency-exchange-to-file-for-bankruptcy-after-hacking-idUSKBN1ED0NJ, [retrieved: December, 2017]