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

Pascal Urien
LTCl
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 & LTCl
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.

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:
  \[ y^2 = x^3 + 7, \quad x, y \in Z/pZ \]

- 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 \in [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 \in [1, n-1]
- kG = (xR, yR), and r = xR 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 \( s = k^{-1} (e + x r) \mod n \) 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 u1 = es^{-1} \mod n and u2 = rs^{-1} \mod n.
2) Compute R = (xR, yR) = u1G + u2P.
3) Set v = xR \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, xP, yP\}, a prefix (one byte 0x04) and a point (xP, yP) of the curve (2x32 bytes, in Z/pZ).

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

1) a1 = SHA256(uF(P)), 32 bytes
2) 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 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: CE1DBAFD7D2E8983ED60E0E1632E86237BB1627AAAB276F2037
A74A081

PublicKey: 04CFD7A542B8C823992AF51DA828E1B693CC5A64F0CAFDF08C31AEC
A471786E285BD3F0EF0A068D075656785EXF7EB149C8880CB905AF3
04182AC942E176CC
Hash160: CB643DD608FB53324A4342C1A6AC3048B409EB
BTC-Addr: 1KYSFr6CyTDMruu8wna981M4ziVyMwftcg
Double SHA2 Check OK
ID: 00
Hash160: CB643DD608FB53324A4342C1A6AC3048B409EB
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 totals 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 | coin index >= 0
sigScript length | 1 byte |
sigScript | 1 byte | contains the signature and the public key
sequence | integer 32 bits | transaction version
End of input
number of outputs | var_int | 1 byte or more
One or more outputs
value | integer 64 bits | satoshi amount
pubKeyScript length | 1 byte |
pubKeyScript | |
locktime | integer 32 bits | transaction locktime

Figure 3. Structure of a Bitcoin transaction

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> 88 // OP_EQUALVERIFY AC // OP_CHECKSIG
```

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 }
```

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

<table>
<thead>
<tr>
<th>sequence</th>
<th>ffffffff</th>
</tr>
</thead>
<tbody>
<tr>
<td>locktime</td>
<td>00000000</td>
</tr>
<tr>
<td>nb_input</td>
<td>1</td>
</tr>
<tr>
<td>input</td>
<td></td>
</tr>
<tr>
<td>transaction</td>
<td>729E344A01E52C822BDFDEC61E28D6ED</td>
</tr>
<tr>
<td>privkey</td>
<td>CE1DBAFD7D2E8983E6080E081632EB0</td>
</tr>
<tr>
<td>index</td>
<td>1</td>
</tr>
<tr>
<td>privkey</td>
<td>CE1DBAFD7D2E8983E6080E081632EB0</td>
</tr>
<tr>
<td>nb_output</td>
<td>1</td>
</tr>
<tr>
<td>output</td>
<td></td>
</tr>
<tr>
<td>fee</td>
<td>0.0005</td>
</tr>
<tr>
<td>btc</td>
<td>0.002685</td>
</tr>
<tr>
<td>hash160</td>
<td>CB643DD608FB5C323A4A6342C1A6AC8048B409EB</td>
</tr>
</tbody>
</table>

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, 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

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.

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 $uF(P)$ is a set of 64 bytes \(\{x_P, y_P\}\), i.e. the point \((x_P,y_P)\) of the curve \((2x32\text{ bytes, in } Z/pZ)\).

The Ethereum address is computed according to the following procedure:

- Compute $a1 = \text{Keccak}(uF(P))$, 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:**
0477A9A9EABCAA26F93D6022E470BDC16010AF22A5482DAB0798A5A2C2AF525810760238A33D8BA6F8E7E89EC15FD6681E7FFC7445B2A063187297592F6

**PrivateKey:**
E49344B32802138C9A250FCEA13F6AE30817BC345F107F05618AFC0E523042

**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 \((\geq 0)\) 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. The 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 \(c\) [0x00 0x7F] : a byte value
- if the string length \([0,55] \): \(0xb7 + \text{Length-of-Length} \epsilon \) \text{value, representing the fee the sender}
- \[0x80, 0xff\] || ByteArray[Length]
- \(0x80\) = NULL String
- if the string Length > 55 : \(0xb7 + \text{Length-of-Length} \epsilon \) \text{value, representing the fee the sender}
- \[0xb8, 0xff\] || ByteArray[Length]

2) List encoding

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

- if the list Length <: 55 : \(0xc0 + \text{Length} \epsilon \) \text{ListItems}
- if the list Length > 55 : \(0xf7 + \text{Length-of-Length} \epsilon \) \text{ListItems}

D. Example of transaction

```
P8 6B // list length= 107 bytes
80 // nonce = null (zero value)
85 04E3B29200 // gasPrice= 21,000,000,000 Wei)
82 9C40 // gasLimit= 40,000 Wei
94 777A07BAB1C119D74545B82A8BE72BEAFF4D447B //Recipient
87 2386F26FC10000 // value= 10,000,000,000,000 Wei
80 // data = null
1C // signature recovery parameter = 28
A0 F1D7D3B245D75368B64670C6D602138 // r value
670319955B747AC85C74FEP7BC804097 // 32 bytes
A0 772D640748007C45C7E22F84A1C1CBA // s value
DF9B3F3606A2F93149135CADD9835DF // 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 // gasPrice= 21,000,000,000 Wei)
82 9C40 // gasLimit= 40,000 Wei
94 777A07BAB1C119D74545B82A8BE72BEAFF4D447B //Recipient
87 2386F26FC10000 // value= 10,000,000,000,000 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

Figure 12. Illustration of an Ethereum transaction script in BTOOLS

**G. Sending a transaction 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.

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

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

Figure 14. Illustration of an ether transaction generation with BTOOLS

![scep256k1](image)

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

**V. CRYPTO CURRENCY SMARTCARD (CCSC)**

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.
<table>
<thead>
<tr>
<th>Command</th>
<th>ISO7816 encoding</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select</td>
<td>00A40400006&lt;@ID&gt; AID= Application IDentifier=010203045000</td>
<td>Start the CCSC application</td>
</tr>
<tr>
<td>Verify</td>
<td>0020000004&lt;UserPIN&gt;=0020000004&lt;User2PIN&gt;=00200000108&lt;AdminPIN&gt;=00200004</td>
<td>Check PIN</td>
</tr>
<tr>
<td>InitCurve</td>
<td>AdminPIN is required</td>
<td>Init curve parameters</td>
</tr>
<tr>
<td>ClearKeys</td>
<td>AdminPIN is required</td>
<td>Clear public and private keys,</td>
</tr>
<tr>
<td>GenKeys</td>
<td>AdminPIN is required</td>
<td>Generate the keys</td>
</tr>
<tr>
<td>SetKey</td>
<td>AdminPIN is required</td>
<td>Set public or private key</td>
</tr>
<tr>
<td>GetKey</td>
<td>UserPin is required</td>
<td>Get public or private key</td>
</tr>
<tr>
<td>SignECDSA</td>
<td>UserPin is required</td>
<td>Return the length (16bits ) of the ECDSA signature and its ASN.1 encoding</td>
</tr>
</tbody>
</table>

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(AdminPIN), 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.

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

- 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 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 30303030303030
// ClearKeys Key 0
APDU 0081 00 00 00
// InitCurve, Key 0
APDU 0089 00 00 00
// GenerateKeysPair 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: 04 BA 5A 71 A8 0E 90 76 9E DD D2 B9 6C B4
BA 47 OB 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 BD F4 7D 05 3B
52 EF F6 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