Spending Programs: A Tool for Flexible Micropayments[⋆]

Josep Domingo-Ferrer and Jordi Herrera-Joancomartí

Universitat Rovira i Virgili, Department of Computer Science and Mathematics,
Autovia de Salou s/n, E-43006 Tarragona, Catalonia, Spain,
{jdomingo,jherrera}@etse.urv.es

Abstract. Micropayments are electronic payments of small amount. Given their low value, the cost of the corresponding electronic transactions should also be kept low. Current micropayment schemes allow a regular amount of money withdrawn from a bank to be split into fixed-value coupons, each of which is used for one micropayment. A more flexible mechanism is proposed in this paper, whereby coupons of variable value can be generated by a *spending program* without increasing the transaction cost. Moreover, the spending program allows one of several alternative ways of splitting the amount withdrawn into re-usable coupons to be selected in real-time.

Keywords: Micropayments, Electronic commerce, Hash functions, Hash chain, Spending program.

1 Introduction

Micropayments are electronic payments of low value and they are called to playing a major role in the expansion of electronic commerce: example applications are phone call payments, access to non-free web pages, pay-per-view TV, etc. The reason for designing specific micropayment schemes is that standard electronic payment systems (like CyberCash [3], e-cash [6] , *i*KP [2], SET [16]) for low-value payments suffer from too high transaction costs as compared to the amount of payments. The cost of transactions is kept high due to complex cryptographic protocols like digital signatures used for achieving a certain security level. However, micropayments do not need as much security as speed and simplicity (in terms of computing). For that reason, several micropayment proposals try to replace the use of digital signatures with faster operations.

1.1 Our Result

Current micropayment schemes allow a regular amount of money withdrawn from a bank to be split into fixed-value coupons, each of which is used for one micropayment. A more flexible mechanism is proposed in this paper, whereby

[⋆] This work is partly supported by the Spanish CICYT under grant no. TEL98-0699-C02-02.

coupons of variable value can be generated, several currencies can be used in successive micropayments, and larger payments can be made without computational overcost for the merchant or the buyer. Moreover, the buyer can provide input at transaction time to select, skip or re-use coupons.

1.2 Plan of This Paper

Section 2 contains some background on hash-based micropayment schemes. Section 3 introduces the concept of spending program. Section 4 discusses non-iterative spending programs (where coupons cannot be re-used). Section 5 presents iterative spending programs (where coupons are re-used). Section 6 is a conclusion. The Appendix recalls the structural program coding which is used to ensure integrity for spending programs.

2 Background on Hash-Based Micropayments

Quite a number of micropayment systems can be found in the literature that use hash functions instead of digital signatures to reduce the computational burden. On a typical workstation, it may take half a second to compute an RSA [15] signature; in that period, 100 RSA signatures can be verified (assuming a small public exponent) and, more important, 10000 hash functions can be computed. Thus, unlike digital signatures, hash functions allow high-rate verification of micropayments by the merchant without committing too many computing resources. This is a key issue since, for low-value payments to be profitable, they must be collected in an inexpensive way and possibly on a large scale (*i.e.* from a large community of buyers). On the buyer's side, replacing digital signatures with hash functions facilitates the use of smart cards, which are very convenient portable devices but have little computing power. So the advantages of dropping digital signatures in favour of hash functions should be clear.

Micropayment systems based on hash functions include NetCard [1], μ-iKP [8] and PayWord [14]. The principle behind those systems is similar. Let F be a computationally secure one-way hash function (*i.e.* easy to compute and hard to invert). Now the buyer takes a value X that will be the root of the chain and computes the sequence $T_n, T_{n-1}, \cdots, T_0$, where

$$\begin{aligned} T_0 &= F(T_1) \\ T_1 &= F(T_2) \\ &\vdots \\ T_{n-1} &= F(T_n) \\ T_n &= X \end{aligned} \tag{1}$$

The values $T_1, \cdots, T_n$ are called coupons and will be used by the buyer to perform n micropayments to the same merchant. Each coupon has the same fixed value v. Before the first micropayment, the buyer sends T_0 to the merchant together with the value v in an authenticated manner. The micropayments are thereafter made by successively revealing $T_1, \cdots, T_n$ to the merchant, who can check the validity of T_i by just verifying that $F(T_i) = T_{i-1}$.

We next mention some differences between the main micropayment systems based on hash functions.

NetCard and μ-iKP are both micropayment schemes bootstrapped with normal e-payment systems, SET and iKP:

- With NetCard the bank supplies the root X of the hash chain to the buyer. The buyer then computes the chain, signs its last element T_0, the total number of elements n and the value of each chain element v. These signed values are sent by the buyer to the merchant, who uses the SET protocol to obtain on-line authorization for the whole chain.
- With μ-iKP, the root of the chain is a random value chosen by the buyer and the payment structure is the same as in the iKP payment system. In other words, the on-line authorization of the chain is performed by authorizing a single iKP payment of regular amount.

PayWord [14] is a credit-based scheme that needs a broker. The buyer establishes an account with the broker who gives her a certificate that contains the buyer identity, the broker identity, the public key of the buyer, an expiration date and some other information. The hash chain is produced by the buyer using a random root. When the buyer wants to make a purchase, she sends to the merchant a commitment to a chain. The commitment includes the merchant's identity, the broker certificate, the last element of the chain, the current date, the length of the chain and some other information. In this scheme, the broker certificate certifies that the broker will redeem any payment that the buyer makes before the expiration date, and the buyer commitment authorizes the broker to pay the merchant. Notice that in this scheme the chain is related to a pair buyer/merchant through the commitment. After that, micropayments are made by the buyer by revealing successive elements of the chain to the merchant. PayTree [9] is an extension to PayWord which uses hash trees rather than hash chains; a hash tree is a tree whose leaf nodes are labeled by secret random values, whose internal nodes are labeled by the hash value of the nodes' successors, and whose root is signed. PayTree allows the buyer to use parts of the hash tree to pay multiple merchants, with possibly several different denominations or currencies.

Pedersen [12] also iterates a hash function with a random root to obtain a chain of coins but he does not provide much detail on what kind of system (credit or debit based) he implements nor does he give information about some other security issues.

The authors of μ-iKP emphasize that the use of hash chains implicitly assumes that micropayments take place repeatedly from the same buyer to the same merchant. Such stability assumption on buyer-merchant relationship can be relaxed at the cost of trusting an intermediate broker who maintains stable relationships with several buyers and several merchants: a buyer can send coupons to the broker and the broker is trusted to relay (his own) coupons to the merchant for the same value.

3 Basic Construction

With the exception of PayTree, the micropayment systems described in Section 2 share a lack of flexibility, which results in at least two shortcomings:

– Since all coupons have the same value v, the only way to be able to pay any amount is to let v be the minimal value, for instance one cent. But this means that the merchant must verify fifty hash functions to get paid a sum as small as fifty cents (being undesirable, note that this is still faster than verifying one RSA signature!, see Section 2). It is true that the buyer just needs to send one hash value (the 50th), but in any case she must store or compute all intermediate hashes.
– Fixed-value coupons do not allow to deal with different currencies.

PayTree mitigates the above lack of flexibility by replacing hash chains with hash trees, but still does not allow coupons to be re-used or dynamically selected. The scheme presented in this paper goes one step further and uses a structure more general than a hash tree, namely a *spending program*:

Definition 1 (Spending program). *A spending program $i_1, \cdots, i_n$ is a program whose instructions i_k are either value instructions, flow-control instructions, input-output instructions or assignment instructions.*

Definition 2 (Value instruction). *A value instruction is one that carries a specific sum of money in a currency specified in the same instruction. When a value instruction of a spending program is retrieved by the merchant, the corresponding sum of money is spent by the buyer.*

Definition 3 (Flow-control instruction). *A flow-control instruction allows to modify the flow of a spending program. Four types of flow-control instructions are used:*

1. *Forward unconditional branch*
2. *Forward conditional branch*
3. *Backward unconditional branch*
4. *Backward conditional branch*

If i_k is a branch to instruction i_j, "forward" means that $k < j$ and "backward" that $k \geq j$. Backward branches allow instruction blocks to be executed more than once.

Input-output and assignment instructions are analogous to machine language instructions of the same type.

Clearly, a spending program generalizes the hash chain concept implemented by equations (1) and the hash tree concept used in PayTree, since the value instructions are in fact coupons of arbitrary value and iterations are allowed. An essential issue is to find a way to encode spending programs such that value instructions in an encoded spending program are as unforgeable as coupons in a hash-chain.

The hash-chain idea was first published in [11] applied to password authentication. In [4][5] a generalization of hash chains called structural coding was applied to the program integrity problem. Structural coding and its properties are recalled in the Appendix (the coding is called structural because it depends on the flow-control structure of the program). Thus, both hash chains and structural coding were invented well before hash-based micropayment systems and for quite different purposes. However, just like hash chains turned out to be natural to implement fixed-value coupons, structural coding is a natural tool to implement spending programs:

Protocol 1 (Spending program micropayment)

1. [**Writing**] *The buyer writes the (unencoded) spending program $i_1, \cdots, i_n$ following her own taste. The buyer's computer or smart card can be used to edit the spending program. Alternatively, "ready-made" standard spending programs supplied by the buyer's bank or other institutions can be used.*
2. [**Coding**] *Spending program instructions $i_1, \cdots, i_n$ are encoded by the buyer into a sequence of so-called traces $T_0, T_1, \cdots, T_n$ using the structural coding [5] based on a one-way hash function F such that in general $F(X \oplus Y) \neq F(X) \oplus F(Y)$, where $\oplus$ denotes bitwise exclusive OR (MD5 [13] or SHA [17] are good candidates for F). Coding could also be performed by the buyer's computer or smart card.*
3. [**Signature**] *The first trace T_0 is signed by the buyer or the buyer's card; let t_0 be the signed form of T_0. The buyer also signs an upper bound α_{t_0} of the amount that can be spent using the spending program that starts with T_0.*
4. [**Initialization**] *The buyer sends t_0 and the signed α_{t_0} to the merchant, who uses a standard payment protocol to obtain on-line authorization for the whole spending program. Note that authorizing an upper bound of the spendable amount does not mean that the buyer is paying anything; payment will be done when value instructions of the spending program are sent by the buyer to the merchant.*
5. [**Microspending**] *Before executing an instruction i_k, it must be decoded from the corresponding trace T_k. Decoding is only successful if no modifications have been done to the program in the traces preceding T_k (run-time integrity property, see Appendix). In [5], a preprocessor for decoding traces was assumed to be pipe-lined and encapsulated with the main processor executing instructions. For spending programs, the procedure is slightly different:*

   (a) *When a value instruction (coupon) is to be paid by the buyer to the merchant, the corresponding trace is sent by the buyer to the merchant, maybe after some flow-control and input-output intermediate traces.*
   (b) *The merchant decodes the instructions in the traces received, to check whether these are valid traces. In particular, if the trace corresponding to the value instruction is correctly decoded, then payment for that value is accepted by the merchant.*
6. **[Clearing]** *The merchant relays the traces received (either one by one or in batches to save transaction costs) to his bank. The bank decodes traces the same way the merchant did, to ensure they are valid (the merchant could try to forge non-existing micropayments). For each correctly decoded instruction value, the bank credits the merchant the corresponding value.*

Note 4. The standard payment system referred to at Step 4 could be for example SET or iKP. For instance, to use iKP, t_0 and α_{t_0} would be placed in the COMMON field defined by that protocol (this is a field containing information shared by all parties involved in the payment, see [2] for more details). This approach is the same adopted by NetCard and μ-iKP, which rely on SET, respectively iKP, for authorization of hash chains.

4 Non-iterative Spending Programs

Since spending programs contain value instructions than can be accepted as payments, an important distinction is whether instructions can be executed more than once.

Definition 5 (Non-iterative spending program). *A spending program $i_1, \cdots, i_n$ is called non-iterative if it contains no backward branches,* i.e. *if no instruction can be executed more than once.*

Thus in a non-iterative spending program, value instructions cannot be reused. This simplifies matters, because the bare knowledge by the merchant of a trace encoding a value instruction is sufficient for the bank to credit the merchant the corresponding value. We next give an example of sequential spending program with value instructions for several different denominations and/or currencies.

Example 6. The structural coding corresponding to a sequential block of n value instructions with values $v_1, v_2, \cdots, v_n$ is next given (sequential block means that no flow-control instructions are present). Let V_i be the amount v_i with some redundancy (in fact the currency name can be used as redundancy). Let F be a one-way hash function as specified in Section 3 and let $\oplus$ denote bitwise exclusive OR. Then the buyer computes the trace sequence $T_n, T_{n-1}, \cdots, T_1, T_0$ as follows:

$$T_0 = F(T_1) \oplus V_1$$
$$T_1 = F(T_2) \oplus V_2$$

$$\begin{aligned} &\vdots \\ T_{n-1} &= F(T_n) \oplus V_n \\ T_n &= F(V_n) \end{aligned} \tag{2}$$

After the above computation, the buyer signs T_0 to get t_0. Before the first micropayment, the buyer sends t_0 and the signed value $\alpha_{t_0} = \sum_{i=1}^{n} v_i$ to the merchant for authorization. Micropayments of values $v_1, v_2, \cdots, v_n$ can be then made by successively revealing $T_1, \cdots, T_n$ to the merchant. For instance, when the merchant gets T_1, then he can retrieve v_1 by computing $F(T_1) \oplus T_0 = V_1$. The same check is performed by the merchant's bank before crediting V_1 to the merchant's account. ◇

The same structure presented in Example 6 could accomodate multicurrency micropayments. For example, v_1 could be in euros, v_2 in dollars, etc. Of course, with a sequential block, this means that euros should be spent first and dollars second, which is rather rigid. Anyway, current micropayment systems offering only fixed-value coupons are worse in that they must rely on an intermediate broker to deal with multiple currencies; this has at least two drawbacks:

- The broker must be trusted
- The broker cannot be expected to do the currency conversion for free

A solution to increase the flexibility offered by the spending program of Example 6 is to use input-output and forward flow-control instructions (forward branches). In this way, *the buyer could provide input at transaction-time which would be used to select some value instructions and skip some others.* Note that such dynamical selection frees the buyer from knowing before signing the first trace which value instructions he will use. For example, if multicurrency value instructions are being used, the selected currency would be a transaction-time input. Note also that skipping value instructions does not mean losing money because money is only spent when the merchant retrieves a value instruction. See the Appendix on how to encode forward branches while ensuring instruction unforgeability.

A second advantage of forward flow-control is that we can include in the spending program value instructions for large sums (which can be optionally used instead of "micro-value" instructions). In this way, the distinction between micropayments and standard payments becomes rather fuzzy.

5 Iterative Spending Programs

Iterative spending programs are programs in which some instructions can be executed more than once.

Definition 7 (Iterative spending program). *A spending program $i_1, \cdots, i_n$ is called iterative if it contains at least a backward branch,* i.e. *if some instructions can be executed more than once.*

Clearly, iterative spending programs add new flexibility advantages:

Storage reduction For example, an arbitrary number of value instructions of equal value v can be stored as a two or three instruction loop using a backward branch.

Complex spending patterns Case-like structures can be created in a spending program so that value instructions can be repeatedly selected or skipped as a function of the buyer's transaction-time input.

Just as clearly, iterative spending programs pose a new security problem. Since value instructions can be re-used, once the merchant has completed a whole loop from an iterative spending program, he could present the value instructions in that loop to his bank an arbitrary number of times pretending to be credited more than he is entitled to.

A way to repair the above security flaw is to require that some additional information should accompany the value instructions in a loop if they are used more than once. Specifically, we propose that a Kerberos-like ticket [10] be sent by the buyer to the merchant along with a re-used value instruction. The following protocol can be run in parallel to Protocol 1:

Protocol 2 (Ticket usage)

1. [**Initialization**] *To use Kerberos tickets, the buyer and the merchant's bank must share a symmetric encryption key K. Such key can be agreed upon by the buyer and the merchant's bank using their public-key certificates at Step 4 of Protocol 1 (initialization). It should be noted that, since micropayment assumes stability in buyer-merchant relationship, the overhead of a key exchange between the buyer and the merchant's bank should be affordable.*

2. [**Microspending**] *Each time a value instruction is re-used (and this happens for instructions in a loop always but the first time), a ticket must be sent by the buyer along with the instruction. The ticket has the form $E_K(T||t)$, where T is the trace corresponding to the value instruction, t is a time stamp, $||$ denotes concatenation and $E_K(\cdot)$ denotes encryption using a symmetric encryption algorithm with key K.*

3. [**Clearing**] *To clear a re-used value instruction, the merchant must send it to his bank along with a fresh ticket,* i.e. *a ticket that was not used in any previous clearing of the same value instruction. Freshness is ensured by the timestamp t in the ticket. The bank will not accept a re-used value instruction unless it is linked to a fresh ticket.*

The reader will notice that a re-used value instruction plus its ticket resemble a Millicent [7] piece of scrip and require a similar amount of processing. The advantage of our scheme over Millicent is that tickets are only required for re-used value instructions, which are typically a (small) fraction of the total number of instructions.

Note 8 (On clearing tickets). Clearing tickets one by one may cost too much as compared to the value of the associated value instruction. On the other hand, if the merchant chooses to clear tickets by batches, then he incurs in some risk of being paid with invalid tickets (*i.e.* not being paid at all). Note that tickets cannot be understood by the merchant (K is unknown to him), so the only way for the merchant to check the validity of a ticket is to try to have it cleared. Clearly there is a tradeoff between cost-effectiveness and security. It is up to the merchant's discretion to find the optimal strategy to solve this (economic) problem.

Note 9 (On coding backward branches). The structural coding used to encode instructions into traces requires that the targets of backward branches be signed (see Appendix). So each new iteration of a loop requires a run-time signature verification. This may be a computational burden if loops are too short, *i.e.* if they contain too few value instructions. Again, an economic tradeoff between verification costs and the flexibility offered by short loops is to be solved.

6 Conclusion

The new concept of spending program has been introduced in this paper. A spending program generalizes hash chains and hash trees used in several current micropayment schemes. The advantages of the new concept are twofold:

Flexibility Unlike hash chains, spending programs can offer multicurrency support, variable-valued coupons and selective use of such coupons. Case-like structures for choosing coupons at transaction-time can be implemented by the buyer or the buyer's smart card. One hot topic where this increased flexibility may be especially useful is intelligent autonomous payments by paying agents.

Efficiency With fixed-value coupons, the value of coupons must be the smallest fraction for the buyer to be able to pay any amount. This implies that verifying a lot of hashes is likely to be necessary for most micropayments. Variable-valued coupons permitted by spending programs allow micropayments of arbitrary value to be performed with a single hash verification. In fact, the distinction between micropayments and standard payments fades away with the availability of variable-valued coupons. On the other hand, iterative spending programs can be considerably more compact than hash chains and hash trees by allowing coupons to be re-used.

Regarding security, the structural coding mechanism sketched in the Appendix can be used to encode spending programs and obtain a degree of unforgeability similar to that of hash chains. However, we would like to stress that the concept of spending program is independent of the actual coding procedure used, as long as integrity is guaranteed.

References

1. R. Anderson, C. Manifavas and C. Sutherland, "NetCard - A practical electronic cash system", 1995. Available from author: `Ross.Anderson@cl.cam.ac.uk`

2. M. Bellare, J. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik and M. Waidner, "iKP - A family of secure electronic payments protocols", in *First USENIX Workshop on Electronic Commerce*, New York, July 1995.
3. CyberCash Inc., `http://www.cybercash.com`
4. J. Domingo-Ferrer, "Software run-time protection: a cryptographic issue", in *Advances in Cryptology - Eurocrypt'90*, ed. I. B. Damgaard, LNCS 473, Springer-Verlag, 1991, pp. 474-480.
5. J. Domingo-Ferrer, "Algorithm-sequenced access control", *Computers & Security*, vol. 10, no. 7, nov. 1991, pp. 639-652.
6. e-cash, `http://www.digicash.com`
7. S. Glassman, M. Manasse, M. Abadi, P. Gauthier and P. Sobalvarro, "The Millicent protocol for inexpensive electronic commerce", in *World Wide Web Journal, Fourth International World Wide Web Conference Proceedings*, O'Reilly, 1995, pp. 603-618.
8. R. Hauser, M. Steiner and M. Waidner, "Micro-payments based on iKP", IBM Research Report 2791, presented also at SECURICOM'96. `http://www.zurich.ibm.com/Technology/Security/publications/1996/HSW96.ps.gz`
9. C. Jutla and M. Yung, "PayTree: " Amortized-signature" for flexible micropayments", in *Second USENIX Workshop on Electronic Commerce*, Oakland CA, Nov. 1996.
10. J. Kohl, B. Neuman and T. Ts'o, "The evolution of the Kerberos authentication service", in *Distributed Open Systems*, eds. F. Brazier and D. Johansen, Los Alamitos, CA: IEEE Computer Society Press, 1994. See also *RFC 1510: The Kerberos Network Authentication System*, Internet Activities Board, 1993.
11. L. Lamport, "Password authentication with insecure communications", *Communications of the ACM*, vol. 24, no. 11, 1981, pp. 770-772.
12. T. P. Pedersen, "Electronic payments of small amounts", in *Security Protocols (April 1996)*, ed. M. Lomas, LNCS 1189, Springer-Verlag, 1997, pp. 59-68.
13. R. L. Rivest, *RFC 1321: The MD5 Message-Digest Algorithm*, Internet Activities Board, 1992.
14. R.L. Rivest and A. Shamir, "PayWord and MicroMint: Two simple micropayment schemes", Technical Report, MIT LCS, Nov. 1995.
15. R. L. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, vol. 21, no. 2, 1978, pp. 120-126.
16. Secure Electronic Transactions. `http://www.mastercard.com/set/set.htm`
17. *Secure Hash Standard*, U. S. National Institute of Standards and Technology, FIPS PUB 180-1, April 1995. `http://csrc.ncsl.nist.gov/fips/fip180-1.txt`

Appendix: Structural Coding for Program Integrity

We recall in this appendix the basic operating principles of the structural coding for program integrity given in [5]. Let $i_1, \cdots, i_n$ be a program, where i_k is a machine-language instruction (for a spending program, i_k can be either a value, a flow-control or an input-output instruction); i_n is not a branch (it can be set to be an **end** instruction). A normalized instruction format is defined: the length of all i_k is made equal by appending a known filler and then each i_k is appended a redundance pattern, whose length depends on the desired security level. Call the normalized program $I_1, \cdots, I_n$.

Let F be a one-way hash function, such that in general $F(X \oplus Y) \neq F(X) \oplus F(Y)$. Now, if $I_1, \cdots, I_n$ is a *sequential block* (containing no branches) it will be encoded into a *trace sequence* $T_0, T_1, \cdots, T_n$, where traces are computed in reverse order according to the following equalities:

$$\begin{aligned} T_0 &= F(T_1) \oplus I_1 \\ T_1 &= F(T_2) \oplus I_2 \\ &\vdots \\ T_{n-1} &= F(T_n) \oplus I_n \\ T_n &= F(I_n) \end{aligned} \tag{3}$$

Now consider branches. If I_k is a *forward unconditional branch* to I_j (*i.e.* $k < j$), it is translated as:

$$\begin{aligned} \cdots T_{k-1} &= F(T_k) \oplus I_k \\ T_k &= F(T_j) \oplus I_j \\ T_{k+1} &= \cdots \\ &\vdots \\ T_j &= \cdots \end{aligned} \tag{4}$$

When I_k is a *forward conditional branch* to instruction I_j, the following traces are computed (also in an index decreasing order):

$$\begin{aligned} \cdots T_{k-1} &= F(T_k) \oplus I_k \\ T_k &= F(T_{k'}) \oplus F(T_{k+1}) \oplus I_{k+1} \\ T_{k'} &= F(T_j) \oplus I_j \\ T_{k+1} &= \cdots \\ &\vdots \\ T_j &= \cdots \end{aligned} \tag{5}$$

Backward branches from I_k to I_j (*i.e.* $k \geq j$) are slightly more complicated to encode, since the trace T_j corresponding to I_j cannot be used to compute T_k or $T_{k'}$ as in forward branches, because T_j follows T_k in the trace computation and depends on T_k (reverse trace computation). Let $\tilde{T}(j)$ a one-to-one integer function on j, for example the identity function. A *backward unconditional branch* at instruction I_k to instruction I_j is translated as:

$$\begin{aligned}
\cdots T_{j-1} &= F(\tilde{T_j}) \oplus F(T_j) \oplus I_j \\
\tilde{T_j} &= F(T_j) \oplus \tilde{T}(j) \\
T_j &= \cdots \\
&\vdots \\
T_{k-1} &= F(T_k) \oplus I_k \\
T_k &= F(\tilde{T}(j)) \oplus I_j \\
T_{k+1} &= \cdots
\end{aligned} \tag{6}$$

Finally, the trace structure for a *backward conditional branch* is as follows:

$$\begin{aligned}
\cdots T_{j-1} &= F(\tilde{T_j}) \oplus F(T_j) \oplus I_j \\
\tilde{T_j} &= F(T_j) \oplus \tilde{T}(j) \\
T_j &= \cdots \\
&\vdots \\
T_{k-1} &= F(T_k) \oplus I_k \\
T_k &= F(T_{k'}) \oplus F(T_{k+1}) \oplus I_{k+1} \\
T_{k'} &= F(\tilde{T}(j)) \oplus I_j \\
T_{k+1} &= \cdots
\end{aligned} \tag{7}$$

In [5], the coding for non-recursive and recursive branch to subroutine is also detailed.

After the previous trace computation, it is up to the protection system (the buyer in the case of a spending program) to endorse the trace sequence with a signature on the first trace T_0 and on every $\tilde{T_j}$ in a backward branch target. So the protection system replaces T_0 with $t_0 = sk_{ps}(T_0)$ and the $\tilde{T_j}$'s with $\tilde{t_j} = sk_{ps}(\tilde{T_j})$, respectively, where $sk_{ps}(\cdot)$ denotes encryption under the protection system's private key.

Now the trace sequence $t_0, T_1, \cdots, T_n$ is fed to a special processor instead of the executable instructions $i_1, \cdots, i_n$. This special processor contains a preprocessor for decoding the above trace structure and retrieving the executable instructions. The following properties hold:

Theorem 10 (Correctness). *The program $i_1, \cdots, i_n$ can be retrieved and executed from its corresponding trace sequence $t_0, T_1, \cdots, T_n$.*

The proof of Theorem 10 is a construction and it shows that, if the preprocessor is pipelined to the main processor, then there is no significant increase in the execution time. A few examples follow to give a flavour of the construction for decoding traces (details can be found in [4]):

– For a sequential instruction block, i_1 is retrieved by computing $I_1 = F(T_1) \oplus pk_{ps}(t_0)$ (where $pk_{ps}(\cdot)$ denotes encryption under the public key of the protection system); the rest of instructions i_k are retrieved by computing

$$I_k = F(T_k) \oplus T_{k-1}$$

While i_k is being retrieved by the preprocessor, i_{k-1} can be executed by the main processor.

– If the instruction i_k is an unconditional forward branch to i_j, at cycle $k+2$, T_{k+2} is read, T_k is decoded by the preprocessor to obtain a valid instruction, and I_k is executed by the processor and recognized as an unconditional forward branch. At cycle $k+3$, only a read operation on T_j is performed; at cycle $k+4$ T_{j+1} is read, and T_k is evaluated by the preprocessor to obtain I_j

$$F(T_j) \oplus T_k = I_j$$

Here $k+3$ and $k+4$ are idle cycles for the processor. From $k+5$ execution continues in sequence from I_j.

– For conditional forward branches, the decoding procedure depends on whether the condition is true or not. In the first case, it is very similar to decoding an unconditional forward branch. If the condition is false, decoding is very similar to the sequential instruction block case.

– For backward branches, decoding proceeds in a similar way, but a few more execution cycles are wasted.

Theorem 11 (Run-time integrity). *If a program $i_1, \cdots, i_n$ is stored as t_0, T_1, $\cdots$, T_n and is decoded as described above, any instruction substitution, deletion or insertion will be detected at run-time, thus causing the main processor to stop execution before the substituted, deleted or inserted instruction(s). Moreover, only the last five read traces need be kept in the internal secure memory of the processor.*

The detailed proof of Theorem 11 can be found in [4]. The key point is that the one-wayness of the function F which connects traces makes it infeasible to successfully substitute, delete or insert traces ("successfully" means that no valid instructions can be decoded by the preprocessor beyond the point where traces have been substituted, deleted or inserted). It is important to stress that valid trace sequences can only be created by the protection system (the one who can sign with sk_{ps}).