Software Run-Time Protection: A Cryptographic Issue

Josep Domingo-Ferrer
Departament d'Informàtica
Universitat Autònoma de Barcelona
08193 Bellaterra, Catalonia-Spain

Abstract. *A new method is featured which solves the software integrity problem by properly coding rather than enciphering. Adopting the lengthy and expensive solution which consists of having the whole program signed/encrypted by an authority would require full decryption and secure storage for the whole program before execution, whereas one signed instruction, pipe-lined decoding-executing, and secure recording of a few of the last read instructions suffice in our case. A general use of the proposed system could practically prevent any viral attack with minimum authority operation.*

0. Introduction

Our goal is *program integrity for the user*, i. e. ensuring that, given an image code, any instruction insertion, deletion or modification *before or during* execution, will cause execution to stop. This requires that the image be stored under a suitable structure, which can be almost completely worked out by the same user who wrote the program. A one-way *function F* [Diff76], such that in general $F(X \oplus Y) <> F(X) \oplus F(Y)$ (where $\oplus$ denotes addition modulo 2 on the binary representations of the operands), and a *public-key signature scheme* must be agreed upon before implementing the method. The signature consists of a private transformation *D*, *exclusively owned by an authority*, and a publicly registered inverse transformation *E*. Also, a *normalized instruction format* must be defined. Suppose an algorithm *A* consisting of machine-code executable instructions $i_1, i_2, \ldots, i_n$. Assume that i_n *is not a branch instruction* (it can be for instance an **END** or a **RET** instruction). Call I_j the instruction resulting from padding i_j to a fixed length and adding a redundance pattern to i_j.

I.B. Damgard (Ed.): Advances in Cryptology - EUROCRYPT '90, LNCS 473, pp. 474-480, 1991.

1. User Preparation Phase

In order for the user to turn a program he has written into a *trusted program*, he first normalizes it into a sequence I_1, ..., I_n, where n is the number of instructions in the program. Then he replaces each I_j with a *trace* T_j. The traces are computed in a *reverse order*, from T_n to T_1. In this way, the sequential program I_1, ..., I_n looks like

$$\begin{aligned} T_1 &= F(T_2) \oplus I_2 \\ T_2 &= F(T_3) \oplus I_3 \\ &\dots \\ T_{n-1} &= F(T_n) \oplus I_n \\ T_n &= F(I_n) \end{aligned} \tag{1}$$

I_1 does not appear in the sequence (1): it will be dealt with in section 2. Once the structure for a sequential program has been designed, we must solve the forward unconditional, forward conditional and subroutine branchings in order to be able to treat any program having no backward branches. *Although for clarity we will present I_k as located in a sequential trace T_{k-1}, this need not be true*, as it will become evident. For the same reason, in the rest of the paper we will also sometimes write the traces following a branch as sequential ones. A *forward unconditional branch* at instruction I_k to instruction I_j is translated as

$$\begin{aligned} \dots \quad T_{k-1} &= F(T_k) \oplus I_k \\ T_k &= F(T_j) \oplus I_j \\ T_{k+1} &= \dots \\ &\dots \\ T_j &= \dots \end{aligned} \tag{2}$$

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

$$\begin{aligned} \dots \quad T_{k-1} &= F(T_k) \oplus I_k \\ T_k &= F(T_{k'}) \oplus F(T_{k+1}) \oplus I_{k+1} \\ T_{k'} &= F(T_j) \oplus I_j \\ T_{k+1} &= \dots \\ &\dots \\ T_j &= \dots \end{aligned} \tag{3}$$

For a *branch to subroutine* (machine-code subroutine) we must also guard against the right subroutine being replaced at run-time; so, assuming that the instructions $I^\wedge_1, ..., I^\wedge_m$ of the subroutine are already encoded as $T^\wedge_0, T^\wedge_1, ..., T^\wedge_m$, we retrieve $F(T^\wedge_1) \oplus I^\wedge_1$ from $T^\wedge_0 = D(F(T^\wedge_1) \oplus I^\wedge_1)$ (see section 2 about the heading trace $T^\wedge_0$) and include it in the calling program as follows

$$\begin{aligned} \ldots \quad & T_{k-1} = F(T_k) \oplus I_k \\ & T_k = F(T_{k'}) \oplus F(T_{k+1}) \oplus I_{k+1} \\ & T_{k'} = F(T^\wedge_1) \oplus I^\wedge_1 \\ & T_{k+1} = \ldots \end{aligned} \tag{4}$$

If I_k is a branch to I_j with $j < k$, the branch trace structures proposed so far cannot be used to compute T_k (for a backward unconditional branch) or $T_{k'}$ (for a backward conditional branch), since a trace T_j is needed which has not yet been computed and depends on T_k (resp. $T_{k'}$). So a *backward unconditional branch* at instruction I_k to instruction I_j is translated as

$$\begin{aligned} \ldots \quad & T_{j-1} = F(T^-_j) \oplus F(T_j) \oplus I_j \\ & T^-_j = F(T_j) \oplus T^-(j) \\ & T_j = \ldots \\ & \quad \ldots \\ & T_{k-1} = F(T_k) \oplus I_k \\ & T_k = F(T^-(j)) \oplus I_j \\ & T_{k+1} = \ldots \end{aligned} \tag{5}$$

Finally, the trace structure for a *backward conditional branch* is straightforward

$$\begin{aligned} \ldots \quad & T_{j-1} = F(T^-_j) \oplus F(T_j) \oplus I_j \\ & T^-_j = F(T_j) \oplus T^-(j) \\ & T_j = \ldots \\ & \quad \ldots \\ & T_{k-1} = F(T_k) \oplus I_k \\ & T_k = F(T_{k'}) \oplus F(T_{k+1}) \oplus I_{k+1} \\ & T_{k'} = F(T^-(j)) \oplus I_j \\ & T_{k+1} = \ldots \end{aligned} \tag{6}$$

Both in (5) and (6), $T^-(j)$ has been computed by applying a one-to-one function to j.

2. Authority Endorsement Phase

After user trace computation, the authority owning the private transformation D endorses the trace sequence by computing a closing trace $T_0 = D(F(T_1) \oplus I_1)$. Notice that the missing instruction I_1 appears now in the trace sequence, and that *the whole program need not be supplied to the authority, but just* $F(T_1) \oplus I_1$.

3. Program Execution with Controlled Instruction Flow

Theorem 1 (Correctness). The program $i_1, i_2, ..., i_n$ can be retrieved and executed from its corresponding trace sequence $T_0, T_1, T_2, ..., T_n$.

Proof (sketch). We have six cases: (a) sequential instruction blocks, (b) forward unconditional branchings, (c) forward conditional branchings, (d) subroutine branchings, (e) backward unconditional branchings, and (f) backward conditional branchings. Due to lack of space, we only prove the first case here. The run-time setting used consists of a coprocessor p', whose task is retrieving the instructions I_k and forwarding them to a usual processor p; it is assumed that p' and p are pipe-lined. The path between p and p' must be a secure one, so that it is advisable that both processor and coprocessor be encapsulated in a single chip (with a hybrid circuit [Ebel86], this is achieved at low redesign cost).

Now, for a **sequential instruction block**, operation at cycle k is: T_k is being read, T_{k-1} is available in a coprocessor internal register, T_{k-2} is being *evaluated* by p' and I_{k-2} is being executed by p (actually the i_{k-2} stripped from I_{k-2} is executed, after redundance checking). Evaluating a trace T_m means to retrieve the instruction contained in the trace (I_{m+1} for a sequential trace, I_j for a branch trace to T_j, see section 1). Then, following this scheme, after reading T_0 and T_1 during cycles *0* and *1*, at cycle *2* T_2 is read and p' evaluates T_0 by computing $F(T_1) \oplus E(T_0) = I_1$. It must be pointed out this computation is feasible because of the transformation E being easy and public and T_1 being available to p'. Thus instruction I_1 has been retrieved, *if its redundance pattern is all right*, of course. Now suppose that at cycle k-*1* I_1 through I_{k-2} have been retrieved and the program has been executed till I_{k-3}. Then at cycle k, I_{k-2} is executed and I_{k-1} is retrieved from T_{k-2} by computing $F(T_{k-1}) \oplus T_{k-2} = I_{k-1}$. Again this is possible because of T_{k-1} being available at cycle k and F being public and easily computable. The result follows by induction. Execution stops at cycle $n+2$ after executing I_n, which means that only two overhead cycles have been introduced (the first read at cycle 0 is unavoidable even for a conventional execution, see diagram 1). As for T_n, this trace is only used during evaluation of T_{n-1} for I_n.

p EXECUTE	*	*	*	I_1	...	I_{n-2}	I_{n-1}	I_n
p' EVALUATE	*	*	T_0	T_1	...	T_{n-2}	T_{n-1}	T_n
p' READ	T_0	T_1	T_2	T_3	...	T_n	*	*
CYCLE	0	1	2	3	...	n	$n+1$	$n+2$

DIAGRAM 1. Sequential Block. $n+2$ Usable Cycles for n Instructions.

4. Run-Time Integrity

Theorem 2 (Run-Time Integrity). If a program $i_1, \ldots, i_n$ is stored as $T_0, \ldots, T_n$, and is evaluated as described in section 3, any instruction substitution, deletion or insertion before or during execution will be detected at run-time, thus causing the processor to stop executing *before* the substituted, deleted or inserted instruction(s). Moreover, only the last five read traces must be kept in the internal secure memory of the processor (they are kept even in case of interrupt).

Proof (Sketch). Since the arithmetic link between two consecutively executed instructions in the sequential case (I_k and I_{k+1}) is essentially the same as in a forward unconditional branching (I_k and I_j), both cases can be reduced to a single one. Thus five cases must be considered for the proof: 1) sequential instruction blocks and forward unconditional branchings, 2) forward conditional branchings, 3) subroutine branchings, 4) backward unconditional branchings, and 5) backward conditional branchings. Because of space reasons, we will only develop the proof for a **sequential block**, which uses only the last read trace (no need for all five last ones); some additions to the main idea are used for the other cases. First consider that an intruder attempts a **substitution**, by replacing I_k with I_k^*; it follows from (1) that he can then choose either to maintain T_{k-1} or to modify it. If he tries the first thing, he must find a T_k^* s. t. $F(T_k^*) \oplus I_k^* = T_{k-1}$, but this is unfeasible given the unidirectionality of F. Consequently T_{k-1} is changed to T_{k-1}^*; now if nothing more is done it will not be possible for the processor p' to retrieve I_{k-1} by evaluation of T_{k-2} (the resulting garbage is not likely going to be a valid instruction because of the redundance field). Thus a change in I_k causes p to stop after execution of I_{k-2}, which is a good behaviour. On the other hand if we recompute $T_{k-2}^* = F(T_{k-1}^*) \oplus I_{k-1}$ then p' will not be able to recover I_{k-2} and execution will stop earlier. Eventually if we proceed the backward recomputation, we see by induction that T_1 will be replaced by T_1^*, and this will be detected since T_0 cannot be replaced (it is signed by the authority), and it will not be possible to retrieve a valid I_1 by evaluation of T_0 at cycle 2. Thus a modification of I_k enforces a modification of T_{k-1}, due to the unidirectionality of F. For this change to remain undetected, backward recomputation of traces should be made, but this is stopped by the signature in T_0; in any case, a defective instruction is *never* executed. If a change of a subset of instructions $I_{k1}, \ldots, I_{kr}$ is attempted a similar argument can be used because this also implies changing some traces. As for **deletion** of a trace T_k from a sequential flow, it also breaks the natural arithmetic link because it amounts to substituting T_{k+1} for T_k, and we have shown that substitutions are detected. Finally, **insertion** of a new trace T_{k^*} between T_k and T_{k+1} is neither feasible: we can compute $T_{k^*} = F(T_{k+1}) \oplus I_{k^*}$ so as to link with T_{k+1}, but this is useless for we cannot link T_k and T_{k^*} without changing the former or having a garbage I_{k+1} in T_k, which will be both detected, as shown above.∎

5. Applications and Conclusion

Our system allows branches and guarantees full integrity while requiring secure storage only for the last five read traces; also decoding and execution are pipe-lined. The work performed by the authority in our scheme is rather small, so that *a great deal of users may share a single authority*, which simplifies most of applications. For example, imagine a large software company, where all programmers write and prepare their programs as specified in sections 1 and 2 in order to protect against computer viruses. Then a single authority can be used to endorse every program. Finally, our proposal requires that only $F(T_1) \oplus I_1$ be supplied to the authority for endorsement; the preparation phase can be carried out by the user himself, so that *the authority need not know what is being signed*, but just a user's valid identification (software privacy).

References

[Diff76] W. Diffie and M. E. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644-654, Nov. 1976.

[Ebel86] G. H. Ebel, "Hybrid Circuits: Thick and Thin Film", in *Handbook of Modern Microelectronics and Electrical Engineering*, ed. C. Belove, Wiley, New York, 1986.

[Park89] G. Parkin and B. Wichmann, "Intelligent Modules", in *The Protection of Computer Software - Its Technology and Applications*, ed. D. Grover, Cambridge University Press, Cambridge, 1989.

[RiAD78] R. L. Rivest, L. Adleman and M. L. Dertouzos, "On Data Banks and Privacy Homomorphisms", in *Foundations of Secure Computation*, ed. R. A. DeMillo et. al., Academic Press, New York, 1978.