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 $F(X \oplus Y) \neq F(X) \oplus F(Y)$ (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 $l_i$ the instruction resulting from padding $i_j$ to a fixed length and adding a redundancy pattern to $i_j$. 
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

\[
\begin{align*}
T_1 & = F(T_2) \oplus I_2 \\
T_2 & = F(T_3) \oplus I_3 \\
\vdots & \\
T_{n-1} & = F(T_n) \oplus I_n \\
T_n & = F(I_n)
\end{align*}
\]

\( I_j \) 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 \( T_{k,l} \), 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

\[
\begin{align*}
T_{k+l} & = F(T_k) \oplus I_k \\
T_k & = F(T_j) \oplus I_j \\
T_{k+l} & = \ldots \\
T_j & = \ldots
\end{align*}
\]

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

\[
\begin{align*}
T_{k+l} & = F(T_k) \oplus I_k \\
T_k & = F(T_j) \oplus F(T_{k+l}) \oplus I_{k+l} \\
T_j & = F(T_j) \oplus I_j \\
T_{k+l} & = \ldots \\
T_j & = \ldots
\end{align*}
\]
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 $I^\wedge_1, \ldots, I^\wedge_m$ of the subroutine are already encoded as $T^\wedge_0, T^\wedge_1, \ldots, T^\wedge_m$, we retrieve $F(T^\wedge_i) \oplus I^\wedge_i$ from $T^\wedge_0 = D(F(T^\wedge_i) \oplus I^\wedge_i)$ (see section 2 about the heading trace $T^\wedge_0$) and include it in the calling program as follows

\[ \begin{align*}
    T_{k,l} &= F(T_k) \oplus I_k \\
    T_k &= F(T_k) \oplus F(T_{k+1}) \oplus I_{k+1} \\
    T_k' &= F(T_k') \oplus I_k' \\
    T_{k+1} &= \ldots
\end{align*} \]  

(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

\[ \begin{align*}
    T_{j,l} &= F(T_j) \oplus F(T_j') \oplus I_j \\
    T_j' &= F(T_j') \oplus T'(j) \\
    T_j &= \ldots \\
    T_{k,l} &= F(T_k) \oplus I_k \\
    T_k &= F(T_k') \oplus I_k' \\
    T_{k+1} &= \ldots
\end{align*} \]  

(5)

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

\[ \begin{align*}
    T_{j,l} &= F(T_j) \oplus F(T_j') \oplus I_j \\
    T_j' &= F(T_j') \oplus T'(j) \\
    T_j &= \ldots \\
    T_{k,l} &= F(T_k) \oplus I_k \\
    T_k &= F(T_k') \oplus F(T_{k+1}) \oplus I_{k+1} \\
    T_k' &= F(T_k') \oplus I_k' \\
    T_{k+1} &= \ldots
\end{align*} \]  

(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 $T_0 = D(F(T_i) \oplus I_i)$. Notice that the missing instruction $I_i$ appears now in the trace sequence, and that the whole program need not be supplied to the authority, but just $F(T_i) \oplus I_i$.

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 $I_n$ 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 redundancy checking). Evaluating a trace $T_n$ means to retrieve the instruction contained in the trace ($I_{m+1}$ for a sequential trace, $I_j$ for a branch trace to $T_p$, 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 $F(T_i) \oplus E(T_0) = I_i$. It must be pointed out this computation is feasible because of the transformation $E$ being easy and public and $T_i$ being available to $p'$. Thus instruction $I_i$ has been retrieved, if its redundancy pattern is all right, of course. Now suppose that at cycle $k-1$ $I_j$ 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 $I_{k+3}$ 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 $I_0$, 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 $T_0$, this trace is only used during evaluation of $T_{n+1}$ for $I_n$.

\[
\begin{array}{cccccccc}
p & \text{EXECUTE} & * & * & * & I_1 & \ldots & I_{n+2} & I_{n+1} & I_n \\
 \end{array}
\]

\[
\begin{array}{cccccccc}
p' & \text{EVALUATE} & * & T_0 & T_1 & \ldots & T_{n+2} & T_{n+1} & T_n \\
 \end{array}
\]

\[
\begin{array}{cccccccc}
p' & \text{READ} & T_0 & T_1 & T_2 & T_3 & \ldots & T_n & * & * \\
 \end{array}
\]

\[
\begin{array}{cccccccc}
\text{CYCLE} & 0 & 1 & 2 & 3 & \ldots & n & n+1 & n+2 \\
 \end{array}
\]

**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 $T_k$, or to modify it. If he tries the first thing, he must find a $T_k^*$ s. t. $F(T_k^*) \oplus I_k = T_k$, but this is unfeasible given the unidirectionality of $F$. Consequently $T_k$ is changed to $T_k^*$; now if nothing more is done it will not be possible for the processor $p'$ to retrieve $I_k$ by evaluation of $T_k^2$ (the resulting garbage is not likely going to be a valid instruction because of the redundancy 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 $T_k^2 = F(T_k) \oplus I_k$, 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 $T_i$ will be replaced by $T_i^*$, 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 $I_i$ by evaluation of $T_0$ at cycle 2. Thus a modification of $I_k$ enforces a modification of $T_k$, 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 $I_k, \ldots, I_i$ is attempted a similar argument can be used because this also implies changing some traces. As for deletion of a trace $T_i$ from a sequential flow, it also breaks the natural arithmetic link because it amounts to substituting $T_k$ for $T_i$, and we have shown that substitutions are detected. Finally, insertion of a new trace $T_i$ between $T_k$ and $T_i$ is neither feasible: we can compute $T_i = F(T_i) \oplus I_i$ so as to link with $T_i$, but this is useless for we cannot link $T_i$ and $T_i$ without changing the former or having a garbage $I_i$ in $T_i$, 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 $F(T_i) \oplus I_i$ 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


