**SANDIA REPORT**

SAND2003-2328
Unlimited Release
Printed July 2003

# Principles of Faithful Execution in the Implementation of Trusted Objects

Philip L. Campbell, Lyndon G. Pierson, Thomas D. Tarman

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.



Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/ordering.htm

SAND 2003-2328
Unlimited Release
Printed July 2003

# Principles of Faithful Execution in the Implementation of Trusted Objects[1]

Philip L. Campbell
Networked Systems Survivability & Assurance

Lyndon G. Pierson
Advanced Networking Integration

Thomas D. Tarman
Advanced Networking Integration

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0785

Abstract

Faithful Execution (FE) is a type of software protection that begins when the software leaves the control of the developer and ends within the trusted volume of a target processor. That is, FE provides program integrity, even while the program is in execution. This report amplifies that definition of FE by describing several simple designs. The implementation of one of those designs is described in a companion report, “Prototyping Faithful Execution in a Java Virtual Machine” SAND 2003-2327.

1. The research described in this report was sponsored by the Defense Advanced Research Projects Agency (DARPA) under Project Kernel.

# Table of Contents



# List of Figures



# List of Tables

# 1 Introduction

We begin with the following definitions:

> Definition: A *trusted volume* is the computing machinery (including communication lines) within which data is assumed to be physically protected from an adversary. A trusted volume provides both integrity and privacy.
>
> Definition: *Program integrity* consists of the protection necessary to enable the detection of changes in the bits comprising a program as specified by the developer, for the entire time that the program is outside a trusted volume. For ease of discussion we consider program integrity to be the aggregation of two elements:
>
> > *instruction integrity* (detection of changes in the bits within an instruction or block of instructions),
>
> and
>
> > *sequence integrity* (detection of changes in the locations of instructions within a program).
>
> Definition: *Faithful Execution* (FE) is a type of software protection that begins when the software leaves the control of the developer and ends within the trusted volume of a target processor.[2] That is, FE provides program integrity, even while the program is in execution. (As we will show below, FE schemes are a function of trusted volume size.)

FE is a necessary quality for computing. Without it we cannot trust computations. In the early days of computing FE came for free since the software never left a trusted volume. At that time the execution environment was the same as the development environment. In some circles that environment was referred to as a "closed shop:" all of the software that was used there was developed there. When an organization bought a large[3] computer from a vendor the organization would run its own operating system on that computer, use only its own editors, only its own compilers, only its own debuggers, and so on. However, with the continuing maturity of computing technology, FE becomes increasingly difficult to achieve

- as the divergence between the development and execution environments increases,
- as the openness of the channel between those environments increases, and
- as the dependence on and volume of network traffic increases.

The discussion below shows that the particular machinery required for FE depends on the size

2. A more intuitive definition is that FE is the assurance that the software that executes is the same as the developer intended. We presume, of course, that the software is in fact what the developer intended.

3. Computers were always large in those days.

of the trusted volume (see Figure 1).

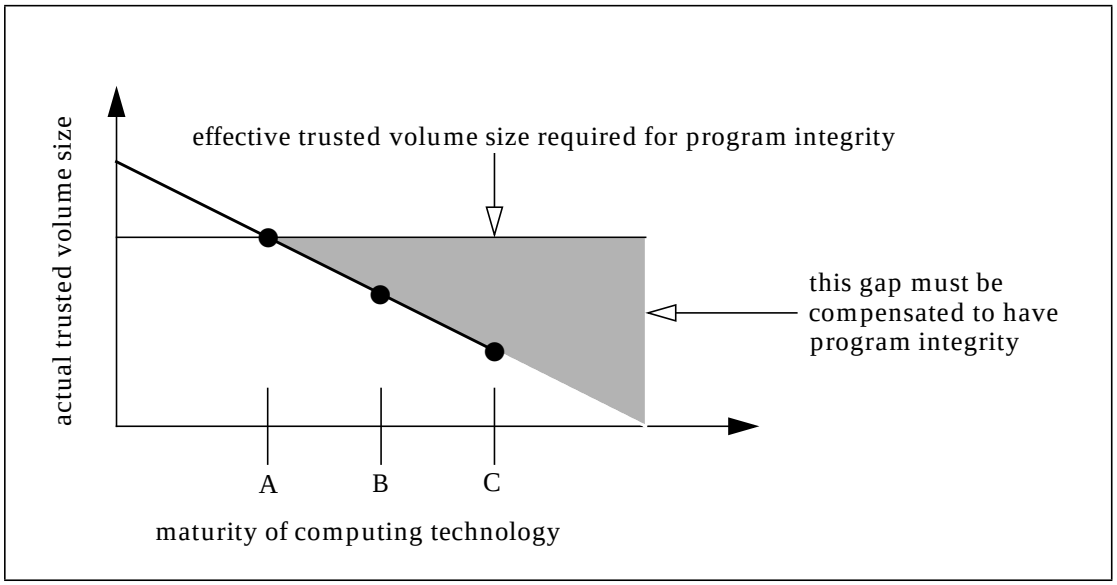


Figure 1 The Shrinking Trusted Volume

As the size of the trusted volume shrinks, there comes a point (see point A in Figure 1) beyond which the actual size of the trusted volume drops below the minimum effective size required for program integrity. Beyond this point a given program can no longer fit within the trusted volume. The program must be stored outside of that volume, and since it is too big to fit all at once, it must be brought into that volume for execution in blocks.[4] The adversary has control over the program while it is outside of the trusted volume. As a result program integrity is lost. In order to regain that integrity, a scheme must be introduced that compensates for the decrease in the actual size of the trusted volume by increasing the effective size of the trusted volume. Such a scheme could create a hash for each block of the program. We presume that the set of hashes would be small enough to fit within the trusted volume and thus be safe from the adversary. When a block is brought in to the trusted volume, the scheme would generate a hash for that block and compare that generated hash with the corresponding stored hash, thereby enabling detection of loss of either instruction or sequence integrity. Transporting programs between trusted volumes would require transporting the set of hashes as well, suitably protected from the adversary via encryption during transmission. We could call the schemes that satisfy these requirements "$FE_A$ schemes," using point A in Figure 1 as our reference point.

As the size of the trusted volume continues to shrink, there comes a second point (see point B in Figure 1) beyond which the set of hashes for our given program can no longer fit within the trusted volume and thus they too must be stored outside of that volume. Since the adversary now has control over each hash, as well as its corresponding program block, the adversary can re-order blocks with impunity (or even create new programs), and with the loss of sequence integrity, program integrity is lost again. (Undetectably breaking instruction integrity would still require breaking the hash scheme.) In order to regain program integrity, a scheme that is a

4. We do not limit or describe these "blocks" here except to say that they are smaller than the size of the program.

superset of $FE_A$ schemes must be introduced that compensates for the additional decrease in the actual size of the trusted volume by increasing the effective size of that volume. Such a scheme could bind each hash to its corresponding block. When a block is brought in to the trusted volume, such a scheme would (1) request the corresponding hash, (2) confirm the integrity of that hash, and then, for comparison, generate a hash from the block, as in the $FE_A$ schemes described above. We presume that the confirmation of the hash would involve cryptography of some sort. We could call the schemes that satisfy these requirements "$FE_B$ schemes," using point B in Figure 1 as our reference point.

As the size of the trusted volume shrinks even further, there comes a third point (see point C in Figure 1) beyond which even the cryptographic keys required for $FE_B$ schemes can no longer fit within the trusted volume. It is not yet clear that FE for general purpose functions is even possible beyond this point, as indicated by the dotted line to the right of point C in the Figure. However, it *has* been shown that for a restricted set of functions FE is possible beyond point C ([20], [21]).

The focus of this report is $FE_B$ schemes. We will first provide context for FE, then we will present sample $FE_B$ schemes using stateless and stateful protection methods. We will then present a sample key management scheme, followed by a discussion of related work and our conclusions.

# 2 Context

In this section we provide context for FE in two ways. First, we show how FE relates to the implementation of "Trusted Objects." And second, we show how FE relates to the software lifecycle; how that lifecyle relates to a working model of protection in general and the use and placement of a "Protection Engine" in particular.

## 2.1 Trusted Objects

Much of our work on FE began with the goal of designing a digital message that is "active." For example, we would like a message to record who had requested read access as the message made its way from originator to destination. We would like the message to be able to authenticate those requesting access and to actively guard itself by demanding communication with the originator before granting requests. We would like the message to be able to hide some of its contents, depending on who was reading it. At a minimum, we would want the message to be able to perform simple duties, such as to assist the reader by displaying the data in various ways, or providing translations for different terminals or printers.

"Trusted Object" [7] is the name we gave to the generalized scheme we developed for a digital message with the capabilities described above. A Trusted Object is a collection of data and programs that is designed to run properly even in an untrusted environment, such as on an adversary's machine. The adversary has access to the binary image of the object as it resides on disk, but the adversary cannot read the data or decipher the program. The value of such objects is that they can carry their security with them, so to speak. Such objects require FE (at least $FE_A$) since they are presumed to spend part of their time outside of a trusted volume and still be protected.

## 2.2 Software Lifecycle

There are three general models of software development: waterfall [19], spiral [6], and evolutionary [16]. The first is the "classic" model, the latter two could be called variations. The five phases in the waterfall model are Requirements, Specification, Implementation & Unit Testing, Integration & System Testing, and Operations & Maintenance. The last phase can be sub-divided into five sub-phases: Build, Package, Distribute, Install, and Execute.

$FE_B$ requires that the Build, Package, and Execute sub-phases occur within a trusted volume but that the other two do not. So explicit protection would need to be provided for the Distribute

and Install sub-phases, as shown in Figure 2.

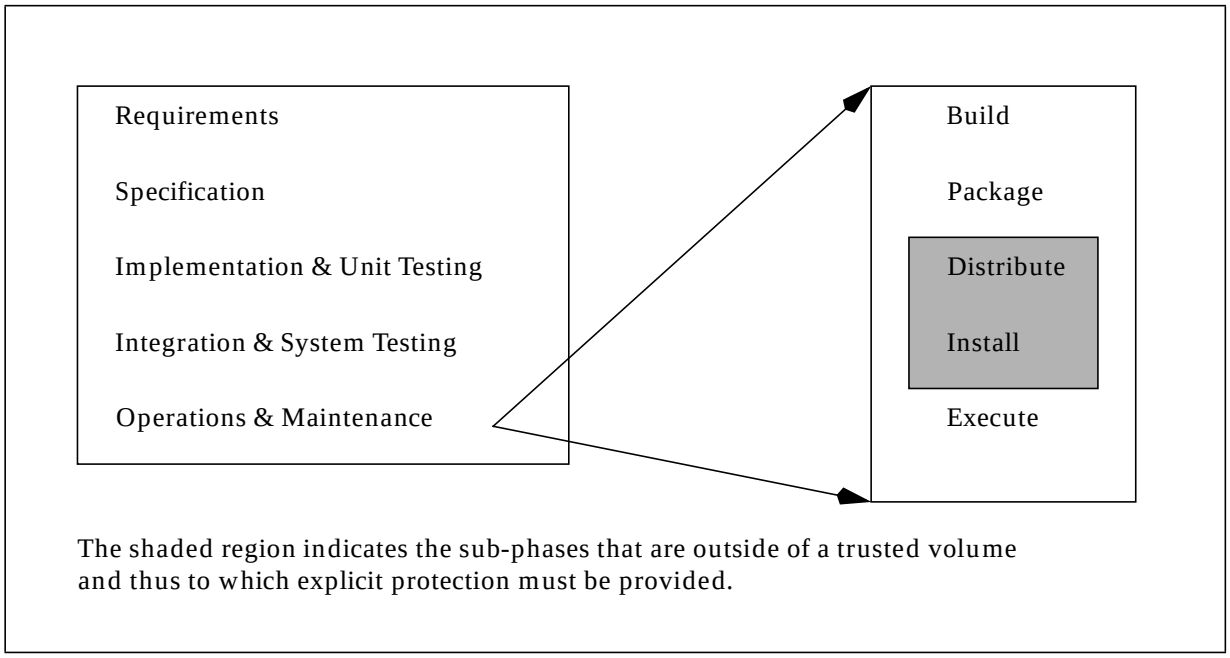


Figure 2 Software Lifecycle Phases

The protection over the Operations & Maintenance phase of the lifecycle, as shown in Figure 2, translates into a working model of protection, using a "Protection Engine," as shown in Figure 3. (The left edge of the left box of Figure 3 is omitted to suggest that the trusted volume

extends to the previous lifecycle phases.)

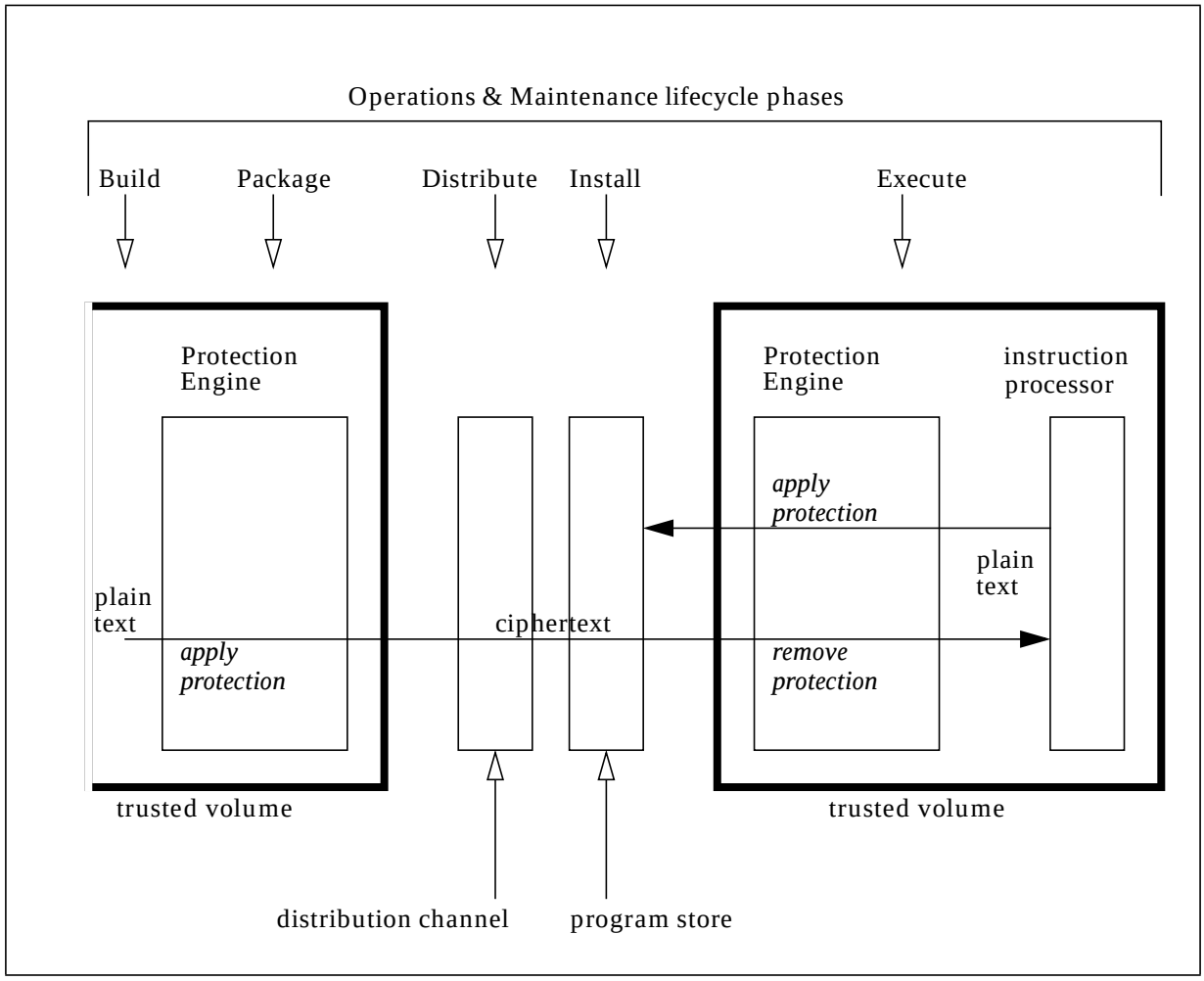


Figure 3 Working Model of Protection

The schemes we present below are examples of the Protection Engine shown in Figure 3.

# 3 Program Integrity

There are two basic approaches for program integrity under $FE_B$: stateless and stateful. We outline schemes using each, both of which use cryptographic protection. The first scheme is completely stateless. The second scheme is predominately stateful.

## 3.1 Stateless

The scheme we outline here for stateless program integrity uses a simple, even a "toy" protection algorithm. The heart of the algorithm is a set of 4-bit-to-4-bit transformations or mappings, the keys for which are kept secret from the adversary. We use three such secret mappings to provide sequence integrity, instruction integrity, and program privacy, respectively.

We will present the general concept of our mappings, then each of the three mappings, and then show how these can be put together to provide program integrity.

### 3.1.1 Stateless Mappings

The mappings we use in the scheme presented below is each a table lookup of a 4 bit value yielding another 4 bit value. This is one of the simplest transformations to implement in hardware or software. Using such a mapping, each of the two 4-bit "nibbles" of an 8-bit byte could be done in parallel. Four different mappings, denoted using functional notation, of 4 bits to 4 bits are shown in Table 1.

Table 1 Four 4-bit-to-4-bit Mappings (in hexadecimal)

| Input | Output | | | |
|---|---|---|---|---|
| | Invertible | | | Non-invertible |
| | $g_r()^a$ | $g_r^{-1}()$ | $g_t() = g_t^{-1}()^b$ | $g_u()$ |
| 0 | 4 | A | 6 | 3 |
| 1 | 1 | 1 | D | 3 |
| 2 | D | 8 | 3 | 2 |
| 3 | C | F | 2 | 8 |
| 4 | 6 | 0 | B | A |
| 5 | B | 7 | 8 | 9 |
| 6 | 9 | 4 | 0 | 0 |
| 7 | 5 | D | F | F |
| 8 | 2 | E | 5 | B |
| 9 | A | 6 | E | E |
| A | 0 | 9 | C | 1 |
| B | F | 5 | 4 | D |
| C | E | 3 | A | 5 |

Table 1 Four 4-bit-to-4-bit Mappings (in hexadecimal)

| Input | Output | | | |
|---|---|---|---|---|
| | Invertible | | | Non-invertible |
| | $g_r()$[a] | $g_r^{-1}()$ | $g_t() = g_t^{-1}()$[b] | $g_u()$ |
| D | 7 | 2 | 1 | 6 |
| E | 8 | C | 9 | C |
| F | 3 | B | 7 | 4 |

a. The names of these mappings follow the form of "$name_{key}()$" and, for the inverse, "$name_{key}^{-1}()$." There is no significance to the name of the keys r, t, or u.

b. This mapping happens to be its own inverse.

The transformations $g_r()$, $g_r^{-1}()$, and $g_t()$ in Table 1 are "invertible" because there exists a key that will invert the transformation: $g_r^{-1}(g_r(i)) = i$, $g_r(g_r^{-1}(i)) = i$, and $g_t(g_t(i)) = i$ for $0 \le i \le F$. The transformation $g_u()$ in Table 1 is "non-invertible" because there is no key that will invert each transformation because $g(g(i)) \neq i$ for at least one i, $0 \le i \le F$.

Unfortunately the term "non-invertible" is customarily also used to refer to any transformation, both invertible and non-invertible, as though the term in this case were really "not necessarily invertible" or simply "any key." We use the term in the same way in the text that follows.

The "key" describing $g_r()$ can be expressed as a 64-bit hexadecimal number, 0x41DC6B952A0FE783, since $g_r(0) = 4$, $g_r(1) = 1$, and so on. The keys for the other transformations can be expressed similarly.

Using the hexadecimal alphabet, an invertible mapping uses all 16 of the hexadecimal digits—0123456789ABCDEF—exactly once. There are 16! (approximately $2^{44}$) mappings of this kind. A non-invertible mapping includes all invertible ones and uses any hexadecimal digit any number of times. There are $16^{16}$ (approximately $2^{64}$) mappings of this second kind. In general terms, if there are n possible digits[5], then there are n! invertible mappings and $n^n$ non-invertible mappings. In general a non-invertible key is preferred because of the larger key space.

We use three, secret mappings and denote them as $g_i()$, $g_s()$, and $g_p()$[6]. The first mapping, $g_i()$, provides instruction integrity. The second mapping, $g_s()$, provides sequence integrity. Since this approach is stateless the mappings operate in Electronic Code Book (ECB) mode, meaning that repeated plaintext blocks result in repeated ciphertext blocks. In order to preclude the adversary from mounting replay attacks, privacy is needed. The third mapping, $g_p()$, provides privacy.

### 3.1.2 Stateless Instruction Integrity

Instruction integrity, using our definition from Section 1, requires the "detection of changes in

5. With hexadecimal there are 16 possible digits.

6. We use the name "g" for these mappings only to avoid confusion with "FE" and the non-linear function "h" that appears in Figure 8.

the bits within an instruction or block of instructions." When we apply protection in the sample scheme that we present here we generate for each plaintext program block an integrity block (this would be in the Protection Engine on the left of Figure 3). When it comes time to check the integrity of a program block, as part of the removal of protection (this would be in the Protection Engine on the right of Figure 3), we generate a second integrity block using the plaintext program block, and then we compare the two integrity blocks. Since we do not try to recover information here but rather just compare it with information that we reproduce, we can use a non-invertible mapping. Our scheme for instruction integrity is shown in Figure 4.

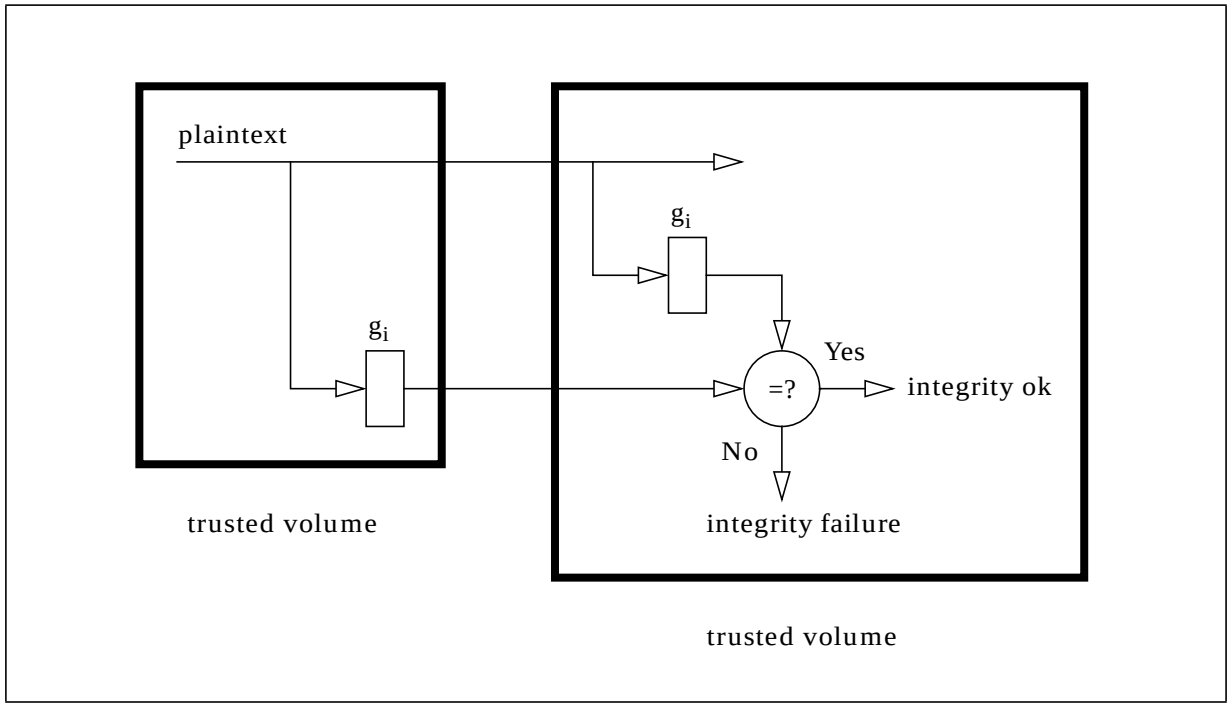


Figure 4 Instruction Integrity

### 3.1.3 Stateless Sequence Integrity

Sequence integrity, again using our definition from Section 1, requires the "detection of changes in the locations of instructions within a program." In our sample scheme we transform the address using a non-invertible mapping, then exclusive-or that with the plaintext program

block, as shown in Figure 5.

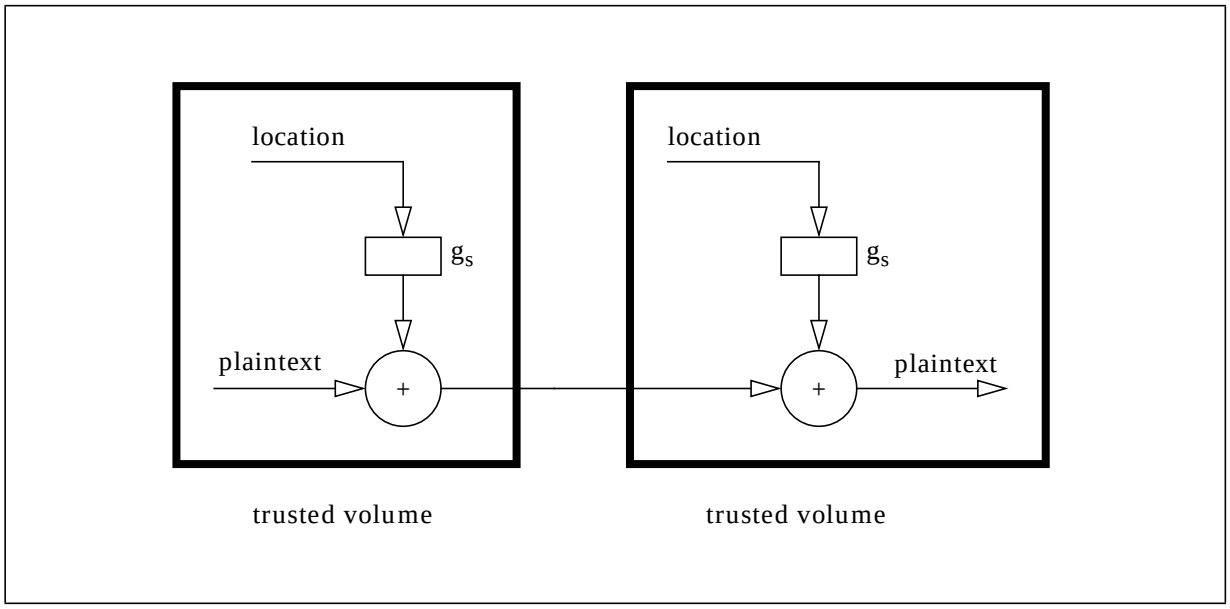


Figure 5 Sequence Integrity

### 3.1.4 Stateless Program Integrity

The third and last mapping provides privacy. For this we use an invertible mapping, since we need to be able to recover information. This mapping is shown in Figure 6. The line outside of the trusted volumes in Figure 6 is denoted “ciphertext,” unlike Figure 4 (the bottom line) and Figure 5, because the plaintext can be recovered from the ciphertext, unlike Figure 4, and the intent is privacy, unlike Figure 5.

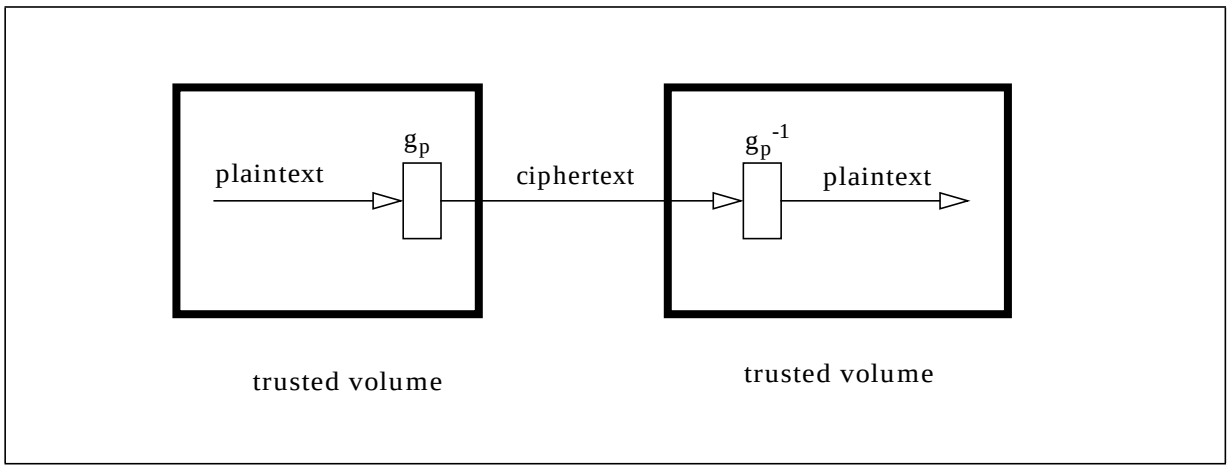


Figure 6 Program Privacy

Combining our schemes for instruction integrity, sequence integrity, and program privacy provides us with program integrity, as shown in Figure 7. Note that we apply the program

privacy transformation, $g_p()$, to the concatenation of the plaintext and the output of the instruction integrity transformation, $g_i()$, before it leaves the trusted volume, and then we show it split apart after it subsequently enters a trusted volume.

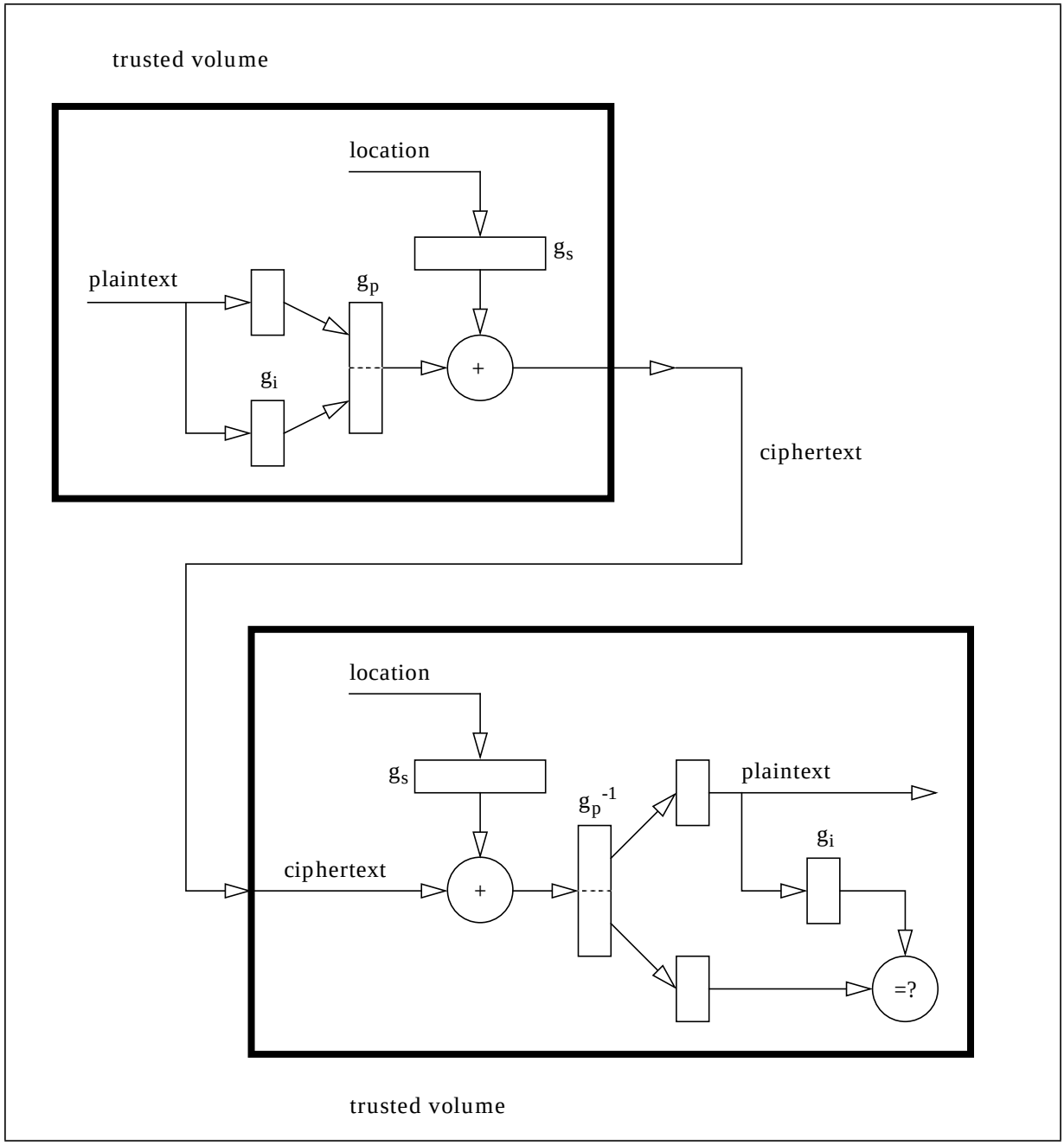


Figure 7 Stateless $FE_B$

The scheme shown in Figure 7 is one of several variations. For example, we could change the order of $g_i()$ and $g_p()$. The scheme we have presented is intended to be for illustration and not as an indication of the most robust or efficient scheme.

## 3.2 Stateful

A stateful scheme for program integrity is at once simpler and harder than stateless. The use of state creates by itself an association between one block and the next, thereby providing sequence integrity. And the mode of encryption also provides instruction integrity. There is no need to add additional privacy, as there is with stateless program integrity, because each instruction is already encrypted. This is the simpler part. The presence of jumps in the code and the program-specific pattern of data accesses pose a problem for a stateful scheme. This is the harder part. We present here an outline of a scheme that exploits the advantages listed above and overcomes the disadvantage, at least on a simple, idealized machine.

We first present an example of stateful encryption, namely Plaintext Block Chaining (PBC) mode. We then show how to overcome the problem with jumps. We then outline our scheme.

### 3.2.1 An Example Stateful Mode: Plaintext Block Chaining

A stateful encryption mode uses a previous state as input to the current state. One example of this mode is Plaintext Block Chaining (PBC), as shown in Figure 8. PBC makes the decryption of each cyphertext word dependent on the proper processing of *all* prior cyphertext words. Changing a single cyphertext word causes all subsequent cyphertext words to become garbled upon decryption.[7]

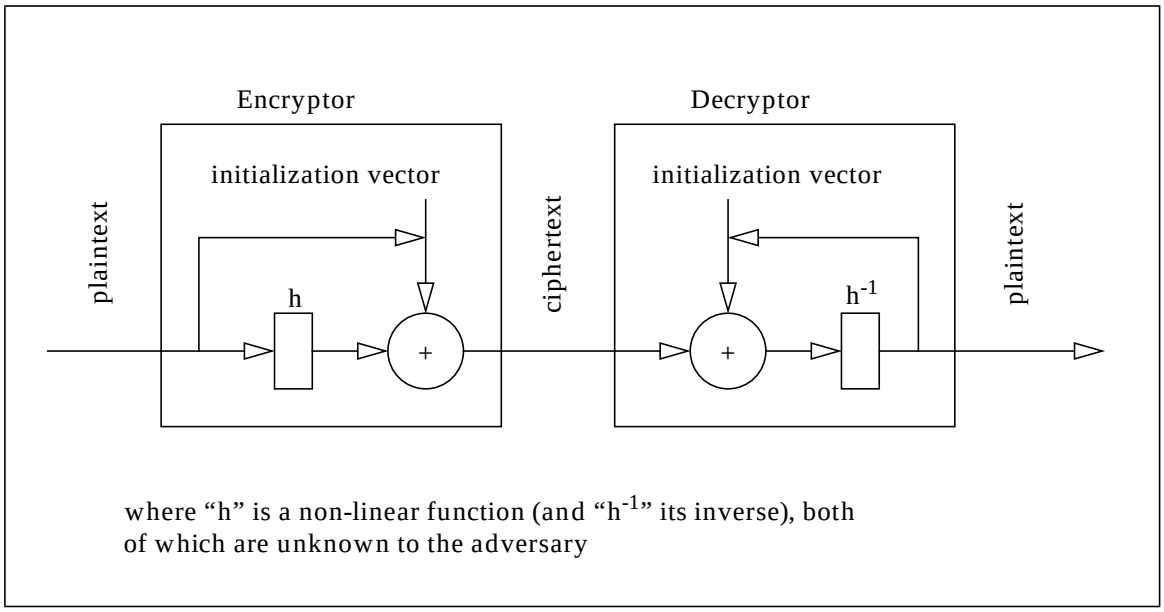


Figure 8 Plaintext Block Chaining (PBC)

7. Because of this "infinite error extension," Plaintext Block Chaining, has never been used for encryption in a communication system that may suffer communication channel errors. However, this "error magnification" characteristic is exactly the type of behavior we want for program integrity.

### 3.2.2 Stateful Program Integrity

As noted above, the problem with stateful encryption for our program integrity is that it loses synchronization each time a jump is made. A scheme[8] to this problem is to direct the Protection Engine to change the initialization vector (IV) whenever it processes a jump instruction, conditional or unconditional. The Protection Engine can detect jumps by decoding each instruction as the instruction makes it way in to the CPU, after the Protection Engine has removed protection. When the current instruction is a jump, then the Protection Engine raises a flag. If that flag is raised when the CPU issues an instruction fetch, then the Protection Engine fetches and loads into its decryptor the IV corresponding to the address of the fetch, then it resets the flag. When the new IV is in place, the instruction that the CPU has requested can be processed. Note that this approach does not limit the number of jumps for which any given address is the target.

This scheme requires that protection be applied as follows (i.e., this is how protection would be applied to a binary file, how the "Protection Engine" on the left of Figure 3 would operate):

- The binary file is first broken into basic blocks.[9]
- An IV is generated for the first instruction of each basic block.
- Each basic block is encrypted, as in Figure 8, using its respective IV.
- The IVs are stored in a separate file, kept secret from the adversary, which is a "dictionary" consisting of <address, IV> pairs.

When the protected file is to be executed, the Protection Engine must also be supplied with the corresponding IV file.

An alternative arrangement may be to compute the IVs dynamically. For example, an IV could be a function of (a) the address, (b) the plaintext instruction, and (c) a base IV that is applicable for all the IVs in an instance of a file. This approach would not require separate storage for IVs. We assume that the base and computed IVs, as well as the plaintext instructions, are all kept secret from the adversary.

Although this approach may work on a simple, idealized machine it is not clear how it would fare on a complex, real machine.

Stateful schemes, such as the one we are describing, rely upon a sparsely populated instruction set, so that the probability will be high that the decryption of a tampered instruction will raise an instruction fault. However, even with a sparsely populated instruction set there is still a non-zero probability that a tampered instruction will decrypt to a legal instruction. If an application

---

8. Attributed to Tom Tarman. See also [26].

9. A basic block is a sequence of instructions such that if the first instruction in the sequence executes, then the entire sequence is guaranteed to execute (i.e., there are no intervening jumps). Basic blocks are assumed to be the longest possible such sequences. Note that a compiled program consists of a sequence of basic blocks, some or all of which may be empty, with one jump instruction between each block. Note also that the instructions after a conditional jump—i.e., the code to which control flows if the jump is not taken—constitute a separate block.

requires the detection of even a single tampered instruction, then redundancy, perhaps similar to that shown in Figure 4, would need to be added.

We do not see a tractable way to apply stateful program integrity to data fetches unless there is an input-independent pattern of the addresses for the sequence of fetches. Stateless program integrity seems to makes more sense here, as outlined in Section 3.1. We presume that the Protection Engine would be able to distinguish between instruction fetches and data fetches.

# 4 Key Management

Protecting a code requires that the Protection Engine in the executing platform (see Figure 3) receive at least one file that contains keys. In the sample scheme we present below we show one way that the keys (and the protected file) could arrive on the "End User's" machine. We presume the following players: Hardware Manufacturers, Software Manufacturers, End Users, and a distribution channel, as shown in Figure 9.

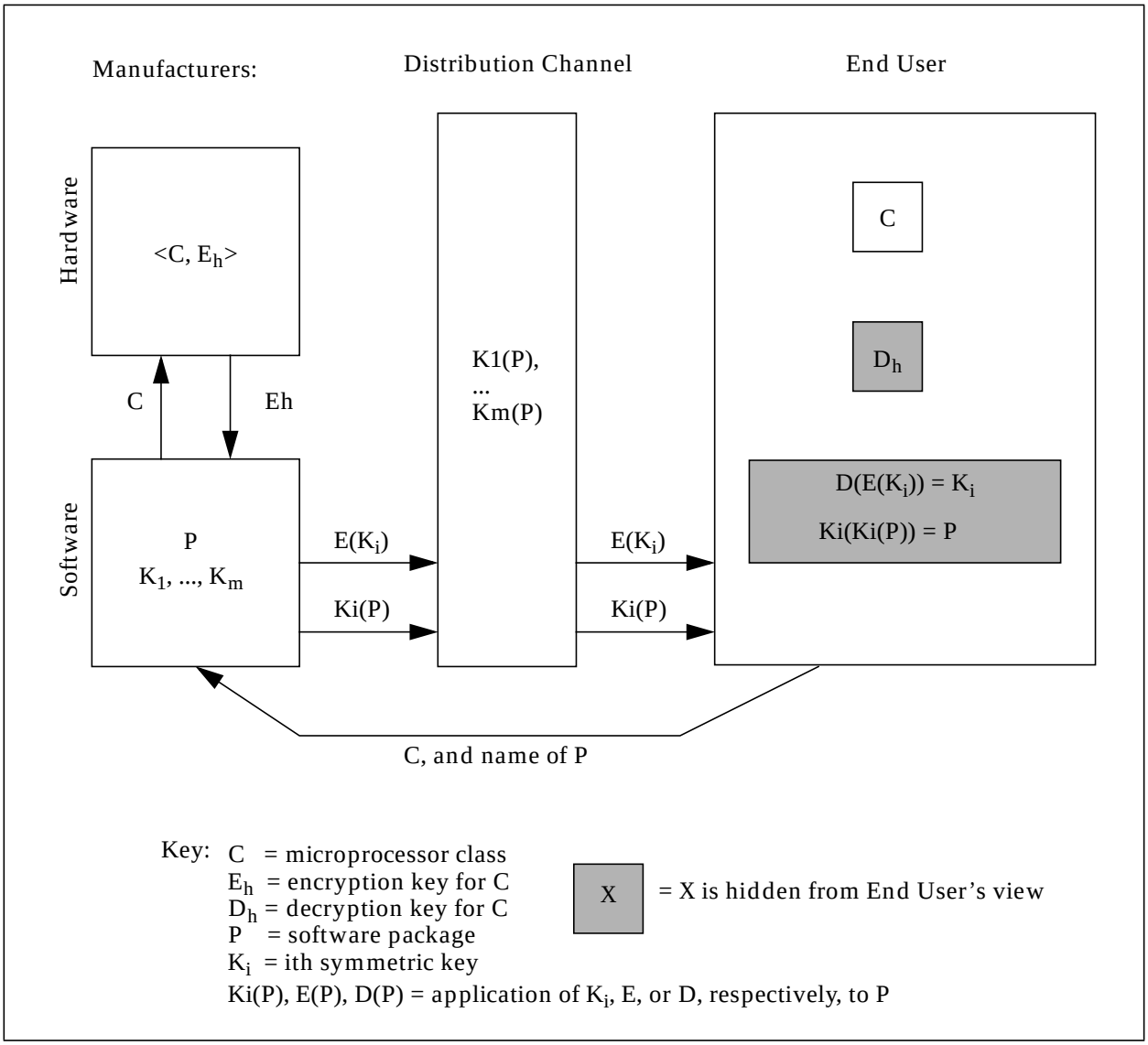


Figure 9 Key Management Model

We presume that Hardware Manufacturers produce microprocessors in different classes (i.e., groups), and that each class has a unique public/private key pair, which we will call $E_h$ and $D_h$ respectively, that are generated by the manufacturer. We presume that the private key is inserted in the microprocessors at the time of manufacture and that the key is never visible to

the End User or the Hardware Manufacturer. The Hardware Manufacturer will have to balance two forces, the one pushing for a large number of microprocessors in each class, making a single, encrypted program executable on a large number of microprocessors, and the other pushing for many classes, limiting the effects of the compromise of a single private key. We presume that the Software Manufacturer has a program, P, that the End User would like to execute.

Notation: We let

"$K_i$" denote the $i^{th}$ symmetric key;

"Ki(P)" denote the encryption or decryption of P via the symmetric key $K_i$;

"E(P)" denote the encryption of P via public key E; and

"E(P)" denote the decryption of P via private key D.

When the End User requests a copy of P, the protocol can begin:

1. The End User sends to the Software Manufacturer the class, C, of his microprocessor and the name of the program, P, he wants. (If the program is already available in some distribution channel, as noted in step 4 below, then the End User would just consult that distribution channel and not need to contact the Software Manufacturer directly.)

2. The Software Manufacturer sends C to the hardware manufacturer.

3. The Hardware Manufacturer does a table lookup and sends the public key, $E_h$, for C to the Software Manufacturer, or notifies of a key compromise.

4. The Software Manufacturer then selects a symmetric key, $K_i$, to encrypt P. The Software Manufacturer encrypts P, producing Ki(P). The Software Manufacturer then uses the End User's public key, $E_h$, to encrypt $K_i$, producing $E(K_i)$. The Software Manufacturer puts both Ki(P) and $E(K_i)$ in some place accessible to the End User. (The Software Manufacturer could do all of these steps before step 1 and provide the End User a table by which he could find Ki(P) and $E(K_i)$ for his C and a particular P.)

5. The End User gets both Ki(P) and $E(K_i)$.

6. The End User instructs his microprocessor to run $D(E(K_i))$ to produce $K_i$.

7. The End User instructs his microprocessor to run Ki(Ki(P)) to produce P. (Note that we presume that $K_i$ and P, both in plaintext, are hidden from the End User's view. Note also that this scheme, as we have presented it, is independent of the granularity at which P is protected: it could be protected as an entire unit, or it could be protected using smaller units, even as small as each instruction or each byte.)

The Software Manufacturer is presumed to have more than one program, P, that it has produced, so we could have $P_1$, ..., $P_n$. And there is more than one class of chip, C, so we could also have $C_1$, ..., $C_q$. We have specified the multiple symmetric encryption keys in the diagram to emphasize that we would expect the Software Manufacturer to create multiple copies of P

symmetrically encrypted with a different key to facilitate distribution of P.

# 5 Related Work

In this section we show the context of FE by comparing and contrasting it to related work.

FE is a type of software protection. An implementation of FE requires more than just software, unlike the "Evaluating Encrypted Functions" approach that Sander & Tschudin ([20], [21]) are researching (see also ([4], [8], [12], [14], [18], [22], and [23]; see [15] for a review of the area, and [1], [3], and [10] for related ideas). In particular it requires specialized hardware, unlike, for example, Fischmeister's "Supervisor-Worker Framework" [11]. It does not require a co-processor, such as Wilhelm's CryPO scheme ([24], [25]) or Yee's "Secure Coprocessor" [27], where the former is designed for mobile code and the latter is not. An implementation of FE could execute more than one piece of software, unlike Best's "Crytpo-Microprocessor" [5]. A copy of the key that an FE implementation uses exists independent of the hardware. That is, there is a key "off-chip," unlike the Secure Microprocessor Chip from Dallas Semiconductor [9]. (The Secure Microprocessor Chip stores sequential instructions in random locations, a feature that we do not consider to be required for FE.) And finally, FE provides integrity, unlike Albert & Morse's scheme [2] (see also [13] and [17]). This area is shown in the form of a binary tree in Figure 10, each node of which is numbered for ease of reference.

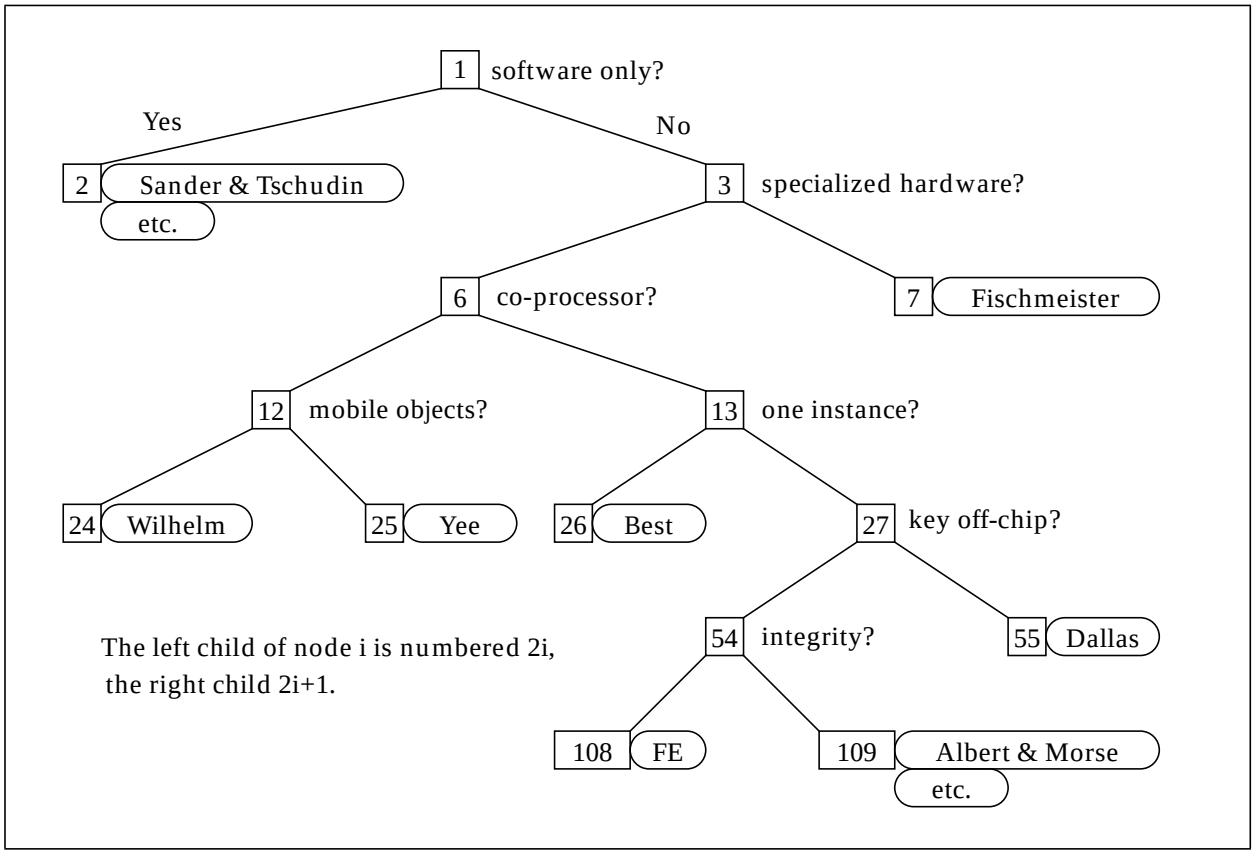


Figure 10 Software Protection

# 6 Conclusions & Future Work

In this report we have defined Faithful Execution and presented sample schemes for its implementation. A companion report, "Prototyping Faithful Execution in a Java Virtual Machine" SAND 2003-2327, describes the implementation of a simple prototype based on a selection of these schemes. We intend for the prototype, which we have named the Sandia Faithfully Executing Java Virtual Machine ($JVM_{fe}$), to enable exploration of more sophisticated techniques as well as tackling increasingly practical problems. Our goal is the implementation of $FE_B$ schemes in hardware with a view to the practical use of FE.

Between where we now stand and our goal there are a number of issues, some of which we list here and which we hope to address via our prototype (the order of the list has no significance):

1. Stateful protection:

   What is the minimum redundant information we need to provide sequence integrity when using stateful protection?

   Can we combine schemes for instruction and sequence integrity to reinforce each other, and thus require less redundant information?

2. Stateless protection:

   How much can we optimize the synchronization penalty at branch targets?

   How effective is loop unrolling?

   Is there an efficient application of stateless protection to data areas?

3. Mode of operation:

   Is a hybrid reasonable, say stateful protection for basic blocks and stateless protection between blocks?

4. Key management:

   What are the alternatives to our scheme, as presented in Figure 9?

5. Context agility:

   What approach optimizes the ease with which we can fetch from and store to different memory areas?

   How do we best implement different levels of integrity and switch between them?

   How do we best incorporate system code that is not protected?

6. Processor instruction set / architecture:

What are the issues in applying FE on differently sized blocks, such as on a variable-length instruction basis as opposed to a byte basis?

Can we efficiently pipeline stateful protection?

What is the optimal placement of the FE Protection Engine with respect to cache?

# References


[1] M. Abadi & J. Feigenbaum, “Secure circuit evaluation.” Journal of Cryptology, vol. 2, no. 1, pp. 1-12, 1990.

[2] Douglas J. Albert, Stephen P. Morse, “Combating Software Piracy by Encryption and Key Management.” Computer, pp. 68-73, April 1984.

[3] Mikhail J. Atallah, K. N. Pantazopoulos, John R. Rice, Eugene E. Spafford, “Secure Outsourcing of Scientific Computations.” Advances in Computers, Vol. 54, pp. 215-272. 2001.

[4] David Aucsmith, “Tamper Resistant Software: An Implementation.” Lecture Notes in Computer Science. No. 1174. pp. 316-333. 1996. 2 refs.

[5] Richard Best, “Preventing Software Piracy with Crypto-Microprocessors.” Proc. Compcon, Spring 1980, IEEE-CS Press, Los Alamitos, CA, pp. 466-469.

[6] B. Boehm, “A Spiral Model of Software Development and Enhancement,” IEEE Computer, Vol. 21, No. 5, pp. 61-72, May, 1988.

[7] Philip L. Campbell, Lyndon G. Pierson, and Edward L. Witzke, “Trusted Objects,” Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference, Phoenix, AZ, April 4-6, 2001. IEEE, Piscataway, NJ, 2001.

[8] Christian Collberg, Clark Thomborson, Douglas Low, “A Taxonomy of Obfuscating Transformations.” Technical Report 148, Department of Computer Science, University of Auckland. 1997. 36 pages. 33 refs.

[9] “DS5002FP Secure Microprocessor Chip.” Dallas Semiconductor. dated 052299. 29 pages.

[10] J. Feigenbaum, “Encrypting Problem Instances Or ..., Can you take advantage of someone without having to trust him?” Advances in Cryptology-CRYPTO ‘85, pp. 477-488.

[11] Sebastian Fischmeister, “Building Secure Mobile Agents: The Supervisor-Worker Framework.” Informationssteeme Institut, Abteilung Verteilte Systeme, Technische UniversitatWien. February 2000. 78 pages. 58 refs.

[12] Robert S. Gray, David Kotz, George Cybenko, Daniela Rus, “D’Agents: Security in a Multiple-Language, Mobile-Agent System.” Mobile Agents and Security. G. Vigna, editor. Lecture Notes in Computer Science, Volume 1419, pp. 154-87, 1998. Springer-Verlag , Berlin. 35 refs.

[13] Amir Herzberg, Shlomit S. Pinter, “Public Protection of Software.” Advances in Cryptology-CRYPTO ‘85, pp. 158-179. (7 refs.)

[14] Fritz Hohl, “Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts.” Mobile Agents and Security. G. Vigna, editor. Lecture Notes in Computer Science, Volume 1419, pp. 92-114, 1998. Springer-Verlag , Berlin. 13 refs.

[15] Gunter Karjoth, Joachim Posegga, “Mobile Agents and Telcos’ Nightmares.” Annales Des Telecommunications. V. 55, No. 7-8, pp. 388-400. 2000. 73 refs.

[16] S. Maguire, Debugging the Development Process. Microsoft Press. Redmond, WA. 1994. ISBN 1-55615-650-2.

[17] George B. Purdy, Gustavus J. Simmons, James A. Studier, “A Software Protection Scheme.” Proc. of the 1982 Symposium on Security and Privacy. April 26-8, 1982. Oakland, CA. pp. 99-103.

[18] James Riordan, Bruce Schneier, “Environmental Key Generation Towards Clueless Agents,” Mobile Agents and Security. G. Vigna, editor. Lecture Notes in Computer Science, Volume 1419, pp. 15-24, 1998. Springer-Verlag , Berlin. 8 refs.

[19] W. W. Royce, “Managing the Development of Large Software Systems: Concepts and Techniques,” in Proceedings IEEE WESCON, pp. 1-9. 1970.

[20] Tomas Sander, Christian F. Tschudin, “Towards Mobile Cryptography.” 1998 IEEE Symposium on Security and Privacy. May 3-6, 1998, Oakland, CA. pp. 215-24. 20 refs.

[21] Tomas Sander, Christian F. Tschudin, “On software protection via function hiding.” Lecture Notes in Computer Science, 1998, V1525, P111-123.

[22] E. Valdez and M. Yung, “Software DisEngineering: Program hiding architectures and experiments,” Lecture Notes in Computer Science, Vol. 1768, pp. 379-394, 2000.

[23] Giovanni Vigna, “Cryptographic Traces for Mobile Agents.” Mobile Agents and Security. G. Vigna, editor. Lecture Notes in Computer Science, Volume 1419, pp. 137-53, 1998. Springer-Verlag , Berlin. 32 refs.

[24] Uwe G. Wilhelm, “Cryptographically protected objects.” Proceedings of RenPar’9, Lausanne, Switzerland. 4 pages. 1997. 7 refs.

[25] Uwe G. Wilhelm, Sebastian Staamann, Levente Buttyán, “On the Problem of Trust in Mobile Agent Systems.” IEEE Symp. Network And Distributed System Security, 1998, San Diego, California. 11 pages. 28 refs.

[26] Edward L. Witzke, “Cryptographic Resynchronization for Faithful Execution,” Sandia Technical Advance (SD-7051), October 2001

[27] Bennet Yee, “Using Secure Coprocessors.” Ph.D. Dissertation. May 1994. Carnegie-Mellon University. 94 pages. 108 refs.

Distribution

| | | |
|---|---|---|
| 1 | 0161 | Patent and Licensing Office, 11500 |
| 1 | 0455 | R. S. Tamashiro, 6517 |
| 1 | 0510 | M. J. De Spain, 2116 |
| 1 | 0510 | G. L. Wickstrom, 2116 |
| 1 | 0784 | R. E. Trellue, 6501 |
| 1 | 0784 | M. J. Skroch, 6512 |
| 1 | 0785 | R. L. Hutchinson, 6516 |
| 1 | 0785 | P. L. Campbell, 6516 |
| 1 | 0785 | T. S. McDonald, 6514 |
| 1 | 0806 | J. M. Eldridge, 9336 |
| 10 | 0806 | L. G. Pierson, 9336 |
| 1 | 0806 | L. Stans, 9336 |
| 1 | 0806 | T. D. Tarman, 9336 |
| 1 | 0806 | E. L. Witzke, 9336 |
| 1 | 0874 | P. J. Robertson, 1751 |
| 2 | 0899 | Technical Library, 9616 |
| 1 | 1361 | K. W. Insch, 5323 |
| 1 | 9018 | Central Technical Files, 8945-1 |

External Distribution

| | |
|---|---|
| 1 | Brian Witten<br>DARPA<br>3701 N. Fairfax Drive<br>Arlington, Va 22203 |
| 2 | Gerald Hamilton<br>Schafer Corporation<br>3811 N. Fairfax Drive, Suite 400<br>Arlington, Va 22203 |