Tamper Resistant Software: An Implementation

David Aucsmith, Intel Architecture Labs
Gary Graunke, Intel Architecture Labs

Tamper Resistant Software: An Implementation

David Aucsmith, Intel Architecture Labs
Gary Graunke, Intel Architecture Labs

Abstract

This paper describes a technology for the construction of tamper resistant software. It presents a threat model and design principles for combating a defined subset of the available threat. The paper then presents an architecture and implementation of tamper resistant software based on the principles described.

The architecture consists of segment of code, called an Integrity Verification Kernel, which is self-modifying, self-decrypting, and installation unique. This code segment communicates with other such code segments to create an Interlocking Trust model.

The paper concludes with speculation of additional uses of the developed technology and an evaluation of the technologies effectiveness.

Introduction

One of the principle characteristics of the PC is that it is an open, accessible architecture. Both hardware and software can be accessed for observation and modification. Arguably, this openness has lead to the PC's market success. This same openness means that the PC is a fundamentally insecure platform. Observation or modification can be performed by either a malevolent user or a malicious program. Yet, there are classes of operations that must be performed securely on the fundamentally insecure PC platform. These are applications where the basic integrity of the operation must be assumed, or at least verified, to be reliable such as financial transactions, unattended authorization and content management. What is required is a method which will allow the fundamentally insecure, open PC to execute software which cannot be observed or modified.

This paper presents the notion of *tamper resistant software*. Tamper resistant software is software which is resistant to observation and modification. It can be trusted, within certain bounds, to operate as intended even in the presence of a malicious attack.

Our approach has been to classify attacks into three categories and then to develop a series of software design principles that allow a scaled response to those threats. This approach has been implemented as a set of tools that produce tamper resistant *Integrity Verification Kernels (IVKs)* which can be inserted into software to verify the integrity of critical operations.

This paper describes the threat model, design principles, architecture and implementation of the IVK technology.

Threat Model

Malicious observation and manipulation of the PC can be classified into three categories, based on the origin of the threat. The origin of the threat is expresses in terms of the security perimeter that has been breached in order to effect the malicious act which translates generally to who the perpetrator is, outsider or insider.

- *Category I* - In this category, the malicious threat originates outside of the PC. The perpetrator must breach communications access controls but must still operate under the constraints of the communications protocols. This is the standard "hacker attack." The perpetrator is an outsider trying to get in.
- *Category II* - The Category II malicious attack originates as software running on the platform. The perpetrator has been able to introduce malicious code into the platform and the operating system has executed it. The attack has moved inside the communications perimeter but is still bounded by the operating system and BIOS. That is, it must still utilize the operating systems and BIOS interfaces. This is the common virus or Trojan horse attack. The perpetrator is an outsider who had access an one time to a system.
- *Category III* - In Category III attacks, the perpetrator has complete control of the platform and may substitute hardware or system software and may observe any communications channel (such as using a bus analyzer) that they wish. This attack faces no security perimeter and is limited only by technical expertise and financial resources. The owner of the system is the perpetrator.

Category I attacks do not require the use of tamper resistant software, rather, require correctly designed and implemented protocols and proper administration. As, by definition, the perpetrator has no direct access to the platform's hardware or software, Category I attacks are better defended by robust access control mechanisms. Frequently, the goal of a Category I attack is to mount a Category II attack.

Category II attacks are caused by the introduction of malicious software into the platform. The malicious software may have been introduced with or without the user's consent and may be explicitly or implicitly malicious. Examples of such software include viruses and Trojan horses as well as software used to discover secrets stored in other software on behalf of other parties (such as another user's access control information).

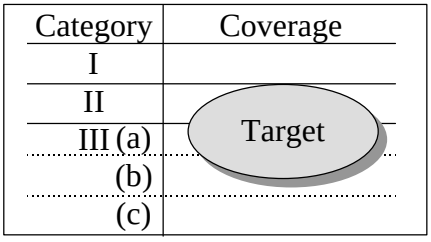

Figure 1: *Threat Model and Tamper Resistance*

An important characterization of Category II attacks is that they tend to attack classes of software. Viruses are a good example of a *class attack*. Viruses must assume certain coding characteristics to be constant among its target population such as the format of the execution image. Other examples would include a Trojan horse program that searches a particular financial application in order to purloin credit card numbers because it knows where within that application such numbers are stored. It is the consistency of software across platforms that enables Category II attacks.

In an absolute sense, Category III attacks are impossible to prevent on the PC. Any defense against a Category III attack must, at best, merely raise a technological bar to a height sufficient to deter a perpetrator by providing a poor return on their investment. That investment might be measured in terms of the tools necessary, and skills required, to observe and subsequently modify the software's behavior. The technological bar, from low to high, would be:

a) No special analysis tools required. These include standard debuggers and system diagnostic tools.

b) Specialized software analysis tools. Tools here include specialized debuggers such as SoftIce and software breakpoint-based analysis tools

c) Specialized hardware analysis tools. These tools include processor emulators and bus logic analyzers.

Our goal for tamper resistant software is to defend against Category II attacks and Category III attacks up to the level of specialized hardware analysis tools. We believe that this provides a reasonable compromise. It is an axiomatic that threat follows value, thus, this level of tamper resistance is adequate for low to medium value applications and high value applications where the user is unlikely to be a willing perpetrator (such as applications involving the user's personal property).

Principles

It is our premise that for software to be tamper resistant it must be immune from observation and modification (within the bounds stated earlier). This requirement implies that the software contains a secret component. Operation on the secret component is the basis for the trust that the application has not been tampered with. Were it not for the secret component, a perpetrator could substitute any software of their choosing for the correct software.

It is the existence of this secret component that compels the user to use that specific software for that specific function rather that some other software. For example, the secret may be a cryptographic key used to encrypt a random challenge in an authentication protocol. Possession of the cryptographic key creates the trust that the software is legitimate.

As another example, consider the need to guarantee that the software has completed a predetermined set of steps. If each step contributed some information to the formation of a shared secret then the presentation of that secret would provide proof that the steps have been executed correctly.

The design principles that we have developed are based on the need to hide a secret in software and insure that the recovery or alteration of that secret is difficult. Four principles were developed:

- *Disperse secrets in both time and space* The secret should never exist in a single memory structure, where it could be retrieved by scanning active

memory. Additionally, the secret should never be processed in any single operation, where anyone monitoring code execution could easily deduce it.

- *Obfuscation of interleaved operations* The complete task to be performed by the software should be interleaved so that a little bit of each part of a task is performed in successive iterations or rounds of the executing code. The goal is to achieve software atomicity, i.e., an "all or none" execution of the software. Such interleaving could be done in a multi-processing environment with cooperating threads. Additionally, the actual execution should be obfuscated to prevent easy discovery of the interleaved component results. Such obfuscation could be accomplished by self-decrypting and self-modifying code.
- *Installation unique code* In order to prevent class attacks, each instance of the software should contain unique elements. This uniqueness could be added at program installation in the form of different code sequences or encryption keys.
- *Interlocking trust* The correct performance of a code sequence should be mutually dependent on the correct performance of many other code sequences.

None of these principles alone will guarantee tamper resistance. Tamper resistance is built from many applications of these ideas aggregated into a single software entity. We have applied these principles in the construction of *Integrity Verification Kernels* (*IVKs*) which are small, tamper resistant sections of code that perform critical functions.

Architecture

The tamper resistant software architecture consists of two parts:

1. *Integrity Verification Kernels* These kernels are small code segments that have been "armored" using the pervasively mentioned principles so that they are not easily tampered with. They can be used alone, to insure that their tasks are executed correctly, or they can be used in conjunction with other software, where they provide the assurance that the other software has executed correctly. That is, they can be used as verification engines.
2. *Interlocking Trust Mechanism* This mechanism uses the inherent strength of the IVK in a robust protocol so that IVKs may check other IVKs. This mutual checking greatly increases the tamper resistance of the system as a whole.

Both of these parts are described in more detail in the following sections.

Integrity Verification Kernel

The IVK is a small, armored segment of code which is designed to be included in a larger program and performs the following two functions:

- Verifies the integrity of code segments or programs
- Communicates with other IVKs

To accomplish these functions securely, an IVK utilizes five defenses:

1. *Interleaved tasks* An IVK may also perform other functions as required but all functions will be interleaved so that no function is complete until they are all complete. Thus, for tasks *A*, *B*, and *C* where *a*, *b*, and *c* are small parts of tasks *A*, *B*, and *C* respectively, the IVK executes *abcabcabcabc* rather than *aaaabbbbcccc*. This is done to prevent a perpetrator from having the IVK complete one of its functions, such as performing the integrity verification of the program, without performing another function, such as verifying the correct functioning of another IVK.
2. *Distributed secrets* The IVK must contain at least one secret (or the IVK could be bypassed by any code written to respond in a pre-determined way). In general, one of these secrets will be a public key used to verify digital signatures.[1] This public key would be used to verify the integrity of the program and to answer the challenge part of the *Integrity Verification Protocol*. In accordance with one of the previously mentioned principles, secrets are broken into smaller pieces and the pieces distributed throughout the IVK. In the case of public keys, they are broken into their Montgomery components (i.e., power series).
3. *Obfuscated code* The IVK is encrypted and is self-modifying so that it decrypts in place as it is executed. The cipher used ensures that, as sections of the code become decrypted, other sections become encrypted and memory locations are reused for different op-codes at different times.
4. *Installation unique modifications* Each IVK is constructed at installation time in such a way that even for a given program, each instance of the

program contains different IVKs. This way, a perpetrator may analyze any given program but will not be able to predict what the IVK on a particular target platform will look like, making class attacks very unlikely. The uniqueness is a property of installation specific code segments and cryptographic keys.

5. *Non-deterministic behavior* Where possible, the IVK utilizes the multi-threading capability of the platform to generate confusion (for an attacker) as to the correct thread to follow.

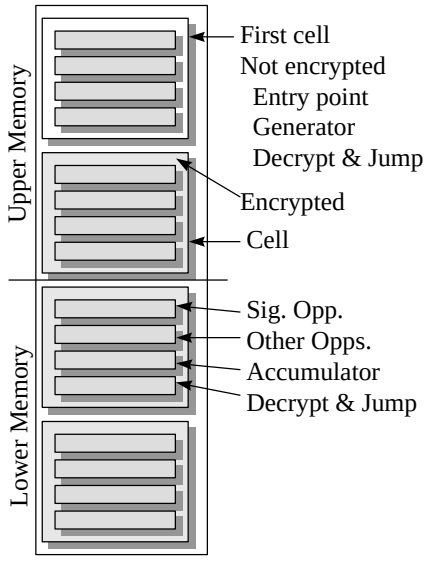

Figure 2: *Structure Of An IVK*

The structure of an IVK is illustrated in figure 2. The IVK is divided into 2^N equal size cells (or code segments) where N > 2. Each cell contains blocks of execution code and, with the exception of the first cell, are encrypted. Cells are not executed linearly but in a pseudo-random order determined by a key. The cell is thus the smallest level of granularity which is ever exposed unencrypted. Cells are exposed by decryption one at a time.

The first cell contains the IVKs entry point. The entry point accepts parameters and begins execution of the IVK. Once control has been transferred to the IVK, one of the parameters is passed to the *generator function*. The generator function uses one of the parameters as the key seed vector, *KV*, to a pseudo-random number function and XORs the generated pseudo-random number string, *KG*, with all of the IVK's remaining cells in a pseudo-one-time-pad manner. So that for an IVK, *M*, which is composed of m bytes

$$m_0 \le m_i \le m_B$$

where there are a total of *B* bytes in *M* and KG is composed of bytes kg_i, then

```
For i = cell_size to B
    m[i] = m[i] XOR KG[i]
```

This sets up the initial state of the IVK. All further states of the IVK are a function of all of the preceding states. Thus, the initial state defines the possible values of future states.

Once the initial state is set up, the *decrypt and jump function* is executed. The decrypt and jump function XORs every cell in upper memory with a partner cell in lower memory and with a substitution key value. An upper memory cell's lower memory partner is chosen according to the value of a transposition key. Thus the value of each lower memory cell is a function of a keyed substitution-transposition derived from cells in upper memory.

The results of the decrypt component of the decrypt and jump function is that at least one cell in lower memory consists of valid op-codes (non-encrypted), referred to as *plaintext*. The jump component then jumps to that one plaintext cell. Once the plaintext cell has been executed, it's decrypt and jump function XORs every cell in lower memory with a partner cell in upper memory and with a substitution key value. Thus, obliterating previous plaintext cells in upper memory and exposing at least one new plaintext cell in upper memory and jumping to it.

The decrypt component is as follows: Given *C* cells in the IVK where m_i denotes a cell in *C* such that

$$m_0 \le m_i \le m_C$$

and

$$m_i \in C \text{ and } C = 2^N$$

There are two encryption keys, *KP*, the permutation key, and, *KS*, the substitution key. Where *KP*[*i*] is the *i*th element of *q* total elements and *KS*[*i*]] is the *i*th element of *t* total elements respectively. Both keys, *KP* and *KS*, are random bit strings.

Next, we define the *partner function*, *P*, as

$$P(i, j) = (KP[\, j \bmod q] \vee 2^{N-1}) \oplus i$$

where *i* is an index of cells such that

$$0 \le i \le 2^{N-1} - 1$$

and *j* is monotonically increasing from 0

$$0 \le j \le \infty$$

Then the decrypt function is

```
For i = 0 to 2^N-1
  m[P(i,j)] = m[P(i,j)] XOR m[i]
    XOR KS[j mod t]
  j = j+ 1
```

for odd rounds, and

```
For i = 0 to 2^N-1
  m[i] = m[i] XOR m[P(I,j)]
    XOR KS[j mod t]
  j = j+ 1
```

for even rounds. One round being one execution of a decrypt and jump function such that even rounds are executed by plaintext cells in upper memory and odd rounds are executed by plaintext cells in lower memory.

Note that *KS* and *KP* do not explicitly exist, rather; the values *KS*[*i*] and *KP*[*i*] are derived at IVK creation and are hard-coded.

The decryption function has many interesting properties. By controlling specific values in *KP* and *KS*, arbitrary cycles can be introduced. These cycles can be used to process function calls or loop structures in the original code. This property is an artifact of XOR exchange. To exchange two values *a* and *b* with out using temporary storage the following sequence can be exercised:

$$a = a \oplus b$$

$$b = a \oplus b$$

$$a = a \oplus b$$

Performing the sequence again will return the original values of *a* and *b* thus creating a cycle.

The other property of interest is that any arbitrary end state can be produced such that a different *KG* could be needed for the IVK to run again.

Returning back to figure 2, once a decrypt and jump function has jumped to a new cell, the new cell begins executing the interleaved tasks in that cell. The cell first process some part of a digital signature. For example, one round of the hash computation or one modular exponentiation of a Montgomery component.

Then the cell performs one part of some additional function such as checking to see if the process is being debugged. Any task can be performed as long as it can be interleaved among many cells.

Next the cell executes the accumulator function. This function computes one round of a hash function where the current cell's value is added into the accumulating product. This accumulating product can be checked by any cell to insure that all previous cells were executed correctly and in the correct order. Given cell *m*, hash function *H*, and the value of *H* at *i* as h_i then

$$h_i = H(h_{i-1}, m_i)$$

After the accumulator function has run, the cell executes it's decrypt and jump function and a new cell is uncovered and begins execution.

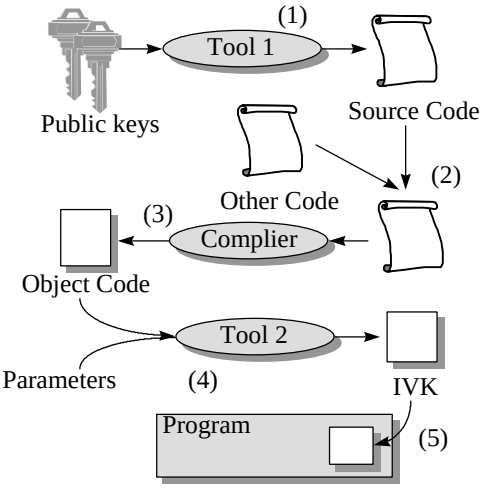

Figure 3: *IVK Creation*

Integrity Verification Kernel Creation

IVKs are constructed using two specialized tools. Tool 1, which is run by the creator of the program in which the IVK is to be embedded and tool 2, which is run at installation time by the install script. Figure 3 illustrates the steps in creating an IVK. In step 1, public keys are feed into Tool 1. Tool 1 computes the Montgomery components of the public keys and generates "C" source code for performing digital signatures using those components.

In step 2, the generated "C" code is interleaved with other, pre-written "C" code, including standard

preambles and footers to produce the source code for the IVK. The source code for the IVK is then compiled in step 3 to generate object code for the IVK.

The object code for the IVK is processed in step 4, along with other parameters, at installation time, to generate a vector of encrypted bytes. The encrypted bytes have a defined entry point which is not encrypted. The encrypted bytes are the IVK. They are then inserted into the target program in step 5 into a location left "empty" when the original program was created.

We will now look at each of these steps in more detail.

Step 1 In step 1, one or more public keys are fed into tool 1 to generate "C" code that produces digital signatures using the Montgomery components of the public keys. The source code which is output contains the "unrolled," optimized code for computing a cryptographic hash followed by the modular exponentiation. The public keys are hard coded into the source code as part of the mathematical operations.

Step 2 In step 2, the source code generated by tool 1 is combined with standard pre-written source code. The pre-written code includes the IVK's entry code, generator code, accumulator code and other code for tamper detection. This step is a primarily manual step. The code for the decrypt and jump function is not added during this step. It is added by tool 2 in step 4.

Step 3 The combined source code is compiled by a standard compiler in step 3 to produce relocatable object code.

Step 4 Step 4 is performed at installation time. The relocatable object code is processed by tool 2. Tool 2 has the following four phases:

I. *Peephole Randomization* In this phase, a peephole randomizer passes over the object code and replaces code patterns with random equivalent patterns chosen from a dictionary of such patterns.

II. *Branch Flow Analysis* In this phase, a branch flow analyzer passes over the object code and rearranges and groups the code into small linear code segments.

III. *Cell Creation* In this phase, tool 2 determines the number of cells to be used, allocates code segments to the cells, adds the accumulator and the decrypt and jump functions to each cell, adds random padding where needed, and finally, fixes-up all the address in the code.

IV. *Obfuscation Engine* This, the last phase of tool 2, generates the three random keys, KP, KS, and KG, computes the visibility schedule for each of the cells, computes the initial start state and encrypts the IVK.

Step 5 In the last step of IVK creation the encrypted IVK generated by tool 2 is copied into a reserved area of the program. The IVK is then ready to be invoked.

Interlocking Trust

Although the IVK can be made generally tamper resistant, its ability to defend itself is greatly enhanced by using an Interlocking Trust mechanism with other IVKs. The Interlocking Trust mechanism consists of three parts: Integrity Verification Kernels (described earlier), an Integrity Verification Protocol and a System Integrity Program. These three components work together as illustrated in figure 4 to create an Interlocking Trust mechanism.

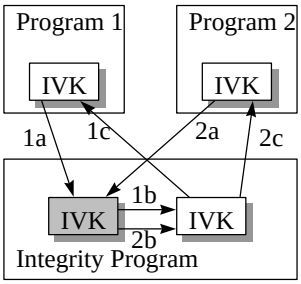

Figure 4: *Interlocking Trust Overview*

As illustrated in figure 4, the objective of the mechanism is to protect programs 1 and 2 from manipulation. As with any non-trivial program, programs 1 and 2 are too complex to be adequately protected. In our implementation we create a small, verifiable, defensible Integrity Verification Kernel. An IVK is included as a part of both programs 1 and 2 as well as part of a system Integrity Program which is accessible to all programs.

Each IVK is responsible for the integrity of the program in which it is embedded. Integrity is verified by calculating the digital signature of the program in which they are contained and comparing the calculated signature value with the correct, hard-coded value, stored in an IVK itself. If desired, the IVK could verify the integrity of any other software component, in addition to the program in which it is contained.

A program may have more than one IVK and may execute them at any time. As long as the IVK is actually executed and is executed correctly, the program cannot be modified without detection. Thus, the vulnerability rests on IVK's correct execution.

Each IVK is constructed according to the previously enumerated principles to make them tamper resistant. They are self decrypting and installation unique. All of their functions are interleaved and self-verifying. These mechanisms are described in detail in the next section of this paper. But, to eliminate a single point of attack, IVKs are interlocked so that a failure or by-pass of any one IVK will be detected by another.

The architecture assumes that there is a System Integrity Program running on the PC that is available to all programs. This System Integrity Program contains a special IVK called the Entry Integrity Verification Kernel (eIVK), shown with a "hatched" background in figure 4, and one or more other IVKs. IVKs within the System Integrity Program verify the integrity of the System Integrity Program and any IVKs embedded in the System Integrity Program.

Upon installation of the System Integrity Program, one or more IVKs are created that are installation unique. The eIVK has a published, external interface that can be called by any other IVK using the Integrity Verification Protocol.

Using figure 4 to illustrate, program 1 executes an embedded IVK. The IVK verifies the integrity of program 1 and then calls (label *1a* in figure 4) the eIVK. The eIVK then verifies the integrity of the System Integrity Program and all contained IVKs and calls another IVK in the System Integrity Program (label *1b* in figure 4). The additional IVK then verifies the integrity of the System Integrity Program and all contained IVKs and calls the original IVK in program 1 (label *1c* in figure 4).

To tamper with program 1, the perpetrator would need to tamper with both program 1 and the System Integrity Program. However, as can be seen in figure 4, any other program (program 2 in figure 4) will also be using the IVKs in the System Integrity Program. Thus, to tamper with any program all programs would have to be compromised.

Integrity Verification Protocol

The integrity verification protocol is used to establish a distributed trust environment. It is the protocol by which IVKs request mutual verification as described in figure 4.

As is shown in figure 4, the protocol is a three party communication between:

1. An IVK embedded in an application (which could be the System Integrity Program)
2. the eIVK, and
3. an IVK embedded in the System Integrity Program.

The protocol provides a challenge/response authentication between the IVK elements and information (such as the success or failure of the verification action) passed in such a way as to be bound to the authentication.

Table 1 gives the definitions of the objects used in the protocol. In the table, subscripts refer to the software module where the value originates. The subscript values may be either *A*, *E*, or *S* for the application module, the eIVK module or the Software Integrity Program respectively.

Object	Definition
K_X^{-1}	Private key of IVK embedded in module X.
K_X^{1}	Public key of IVK embedded in module X.
A_X	Address of the entry point of the IVK embedded in module X.
H_X	Hash value computed over the code of module X.
R_X	Random value derived by module X.
F_X	Flag value (success or failure) of operation originating from X. The flag is 9 bits where each bit is the result of an operation and where a 0-bit is a success and a 1-bit is a failure.

Table 1: *Protocol Components*

The protocol is a simple signed message protocol with random values added to provide protection against replay. Table 2 lists the information known by, and present in each of the parties *a priori.* Table 2 uses the following notation:

- *M* encrypted with public key of *X*.

$$K_X^1[M]$$

- *M* concatenated with *N*.

$$M|N$$

- *X* sends *M* to *Y*

$$X \rightarrow Y: M$$

- Bit 0 of F is equal to the result of a test of *M* = *N*.

$$F_0 = (M = N)$$

Party	Information
Application (A)	K_A^1 public key of A K_E^1 public key of E $K_A^{-1}[H_A]$ signature of A under A $K_E^1[H_A]$ signature of A under E A_E address of E A_A address of A
eIVK (E)	K_E^{-1} private key of E $K_E^1[H_E]$ signature of E under E A_S address of S
System Integrity Program (S)	K_S^1 public key of S K_E^1 public key of E $K_S^{-1}[H_S]$ signature of S under S $K_S^{-1}[H_E]$ signature of E under S $K_E^1[H_S]$ signature of S under E A_E address of E

Table 2: *Initial state knowledge of participates*

The protocol can be described in the following steps:

1. *A* verifies signature of *A* and reports any failure

$$F_0 = (H_A = K_A^1[K_A^{-1}[H_A]])$$

2. *A* sends signature, flag, and random number to *E*

$$A \rightarrow E: K_E^1[K_E^1[H_A]|F|A_A|R_A]$$

3. *E* verifies signature of *E* and reports any failure

$$F_1 = (H_E = K_E^{-1}[K_E^1[H_E]])$$

4. *E* verifies signature of *A* and reports any failure

$$F_2 = (H_A = K_E^{-1}[K_E^1[H_A]])$$

5. *E* sends signature and address of *A*, flag and random number to *S*

$$E \rightarrow S: K_E^{-1}[H_A|A_A|F|R_E]$$

6. *S* verifies signature of *S* and reports any failure

$$F_3 = (H_S = K_S^1[K_S^{-1}[H_S]])$$

7. *S* verifies signature of *E* and reports any failure

$$F_4 = (H_E = K_S^1[K_S^{-1}[H_E]])$$

8. *S* verifies signature of *A* and reports any failure

$$F_5 = (H_A = K_E^1[H_A])$$

9. *S* sends *E* signature, flag and random number

$$S \rightarrow E: K_E^1[K_E^1[H_S]|F|R_E]$$

10. *E* verifies that received random number is the same

$$F_6 = (R_E = K_E^{-1}[K_E^1[R_E]])$$

11. *E* verifies signature of *S* and reports any failure

$$F_7 = (H_S = K_E^{-1}[K_E^1[H_S]])$$

12. *E* sends flag and random number to *A*

$$E \rightarrow A: K_E^{-1}[F|R_A]$$

13. *A* verifies that received random number is the same

$$F_8 = (R_A = K_E^1[R_A])$$

14. *A* verifies that flag equal 0 and reports any failure

$$(F = 0)$$

The above protocol is not particularly efficient but is reasonably robust. It requires no prior knowledge of the application from the other parties is the case given the installation specific uniqueness.

System Integrity Program

The System Integrity Program is a program module that constantly monitors the integrity of the security components of the computer system. While the System Integrity Program monitors the integrity of the security components, it depends on the eIVK to monitor its own integrity.

The System Integrity Program is created at installation time so that both the eIVK and the embedded IVK can be made unique. The System Integrity Program insures that the eIVK has a known entry point an a known public key.

Technology Extensions

The current design of the IVK could be expanded in numerous ways to make it more effective. These technology extensions fall into two broad categories: *active defense* and *hardware assisted protection*.

Active defense is code, added to the IVK, that detects attempts to observe or modify the execution of the IVK and then fights back. Detection possibilities include scanning running code for "signatures," much the way viruses are detected. Signatures of debuggers or emulators can be scanned. Additionally, one may look for specific interrupt vectors which are used by particular programs. Once detected, the IVK may disconnect interrupts used by the debugger or modify the target's code so that it fails.

Hardware assisted protection uses some characteristic of the hardware to assist the software tamper resistance. Several techniques are currently under investigation. The first utilizes the processor's execution counter to measure the time used by the process. Once this is done then the secret, a private key, is not actually stored in the IVK using tool 1, as described earlier, but, rather, the statistical timing characteristics of operating on the secret are stored instead. The IVK then guesses at the correct key and utilizes the timing characteristics in an auto-correlation process to derive the secret. Any attempt to manipulate the code while the auto-correlation is executing would keep the IVK from deriving the correct secret. This method is a use of Kocher's cryptanalysis of fixed exponent systems.[2]

Additional hardware assistance can be derived by locking the execution of the IVK in the processors on-chip cache.

Conclusion

This paper presented an implementation of tamper resistant software. The true resistance of which can only be judged empirically. As such, only time will confirm whether the principles described herein are valid and the implementation sufficient.

Acknowledgment

The authors would like to thank the many people who's comments, criticism, and abuse have keep use refining our approach.

References

[1] Rivest, R., Shamir, A., and Adleman, L. A Method for Obtaining Digital Signatures and Public-Key Crypotsystems. *Communications of the ACM*, vol. 21, issue 2, Feb. 1978, pp. 120-126.

[2] Kocher, P. Cryptanalysis of Diffie-Hellman, RSA, DSS, and Other Systems Using Timing Attacks. *Private Extended Abstract*, 7 December 1995.