Public Protection of Software

Amir Herzberg and Shlomit S. Pinter

Dept. of Electrical Engineering

Technion - Israel Institute of Technology

Haifa 32000, ISRAEL

Abstract

One of the overwhelming problems that software producers must contend with, is the unauthorized use and distribution of their products. Copyright laws concerning software are rarely enforced, thereby causing major losses to the software companies. Technical means of protecting software from illegal duplication are required, but the available means are imperfect. We present protocols that enables software protection, without causing overhead in distribution and maintenance. The protocols may be implemented by a conventional cryptosystem, such as the DES, or by a public key cryptosystem, such as the RSA. Both implementations are proved to satisfy required security criterions.

H.C. Williams (Ed.): Advances in Cryptology - CRYPTO '85, LNCS 218, pp. 158-179, 1986.

1. Introduction

Great losses to software producers are currently incurred due to the ease of copying most computer programs. It is common practice for one user to buy a software product, and without the producer's consent to give or sell it to other installations. The economic value of software protection resulted in many products that supplied means to protect software. It is shown in [HK84] that many commercially available means suffer from some of the following deficiencies:

1. Insufficient protection.
2. Impaired backup capability (for the innocent user).
3. Narrow range of applicable systems (i.e., methods that protect only firmware).
4. Obstacles for distribution and maintenance, of the computers and the software.
5. Excessive overhead in total costs or in execution time.

This paper describes and proves the security of a software protection system that does not suffer from the deficiencies indicated. A preliminary version of PPS (Public Protection of Software) has been presented, with other software protection methods, in [HK84]. In contrast with the deficiencies outlined above, PPS provides:

1. Provable, hence reliable, protection.
2. Undisturbed backup capability.
3. Applicable on virtually all systems.
4. Simple, undisturbing protocols for distribution and maintenance.
5. Reasonable overhead in total costs and execution speed.

PPS requires modifications to the architecture of the processor. Therefore, it can only be implemented by CPU manufacturers. In a recent paper [AM84], another software protection method (henceforth referred to as AM) was presented that requires similar modifications to the internals of the processor. PPS differs

mainly in the protocols used. The PPS protocols require less communication between the parties, and minimal intervention of the key generating body (denoted Z) and the software producer. For example, communication between the software producer and the system integrator before the protection of each product is not required. This communication is essential in AM. In addition, PPS provides protocols for replacing malfunctioning CPUs and indirect software distribution (via a dealer). AM does not provide protocols for those functions. A detailed comparison of PPS and AM may be found in Section 2.1.

PPS is the combination of three protocols, two for the distribution of software and one for replacement of malfunctioning CPUs. PPS may be implemented either by public key cryptosystems or by conventional cryptosystems. Section 2 discusses the protection supplied by PPS. In Section 3 we describe how PPS may be implemented by public key cryptosystems (PPS/PK). Section 4 a formal model for discussing the security of PPS is presented. The security of the public key cryptosystem implementation is then proved. This implementation is straightforward, but the conventional cryptosystem implementation (PPS/C) presented in Section 5 seem to be much more realistic. Section 6 gives the final conclusions.

The protection provided by PPS

PPS attempts to render unprofitable the effort required to copy protected software. PPS relies upon mechanisms embedded in the CPU, therefore PPS cannot prevent the CPU producer from making secret trapdoors in the CPU that will enable software duplication. PPS requires a key-producing body, which installs the initial keys in the CPU and enables replacement of failing CPUs. This body may be the CPU producer, and it is represented by Z or the *center* in this paper. PPS enables Z to distribute the keys in such a manner that prevents other bodies from creating valid keys. If the system's OEM is Z, this feature might help to prevent the creation of "clones" (compatible computers by other OEMs).

Intuitively, PPS provides three levels of protection. The first level is against simple piracy attacks. Such attacks use legal procedures and attempt to duplicate

software by some unforeseen manipulation of those procedures. The second level is against more determined attacks, that include the faking of a CPU failure. Due to obvious reasons, a new CPU, that runs all the software bought for the failing CPU, should be provided quickly. It is obvious that if the CPU did not really fail, and is not returned, the attackers will have two CPUs that run the same software. While this hazard should be protected against by an appropriate procedure, PPS ensures that no further gain may be achieved by faking a CPU failure. PPS's third level of protection is against attackers that physically violate the CPU's enclosure, and discover (literally!) the keys held within. This approach is quite extreme, but it has been argued that such attacks may be attempted by parties that desire to cause distrust in the center or in the CPU. Only when implemented by a public key cryptosystem, PPS provides some protection against this attack. After violating the integrity of the CPU, the attackers will only be able to decypher protected code encrypted for the violated CPU.

A possible modification of PPS is transferring a key (*execution key*) instead of the actual program [AM84,HK84]. The program is then transferred and encyphered by that key. The CPU operates the program using the execution key. The security analysis of such a modification would not change compared to that of PPS (given in Section 4). As described in the references, this modification might improve greatly the performance of the CPU.

1.1. PPS vs AM

1. The influence of PPS on the architecture of the CPU is the same as the influence detailed in [AM84], and is not discussed here.
2. Both methods provide sufficient protection, undisturbed backup capability, wide range of applicable systems, and reasonable overhead in total costs and execution time.
3. PPS may be implemented either by using public-key cryptosystems or by using conventional cryptosystems, while AM requires public-key

cryptosystems. The implementation of public-key systems is much harder.

4. PPS does not require communication between the software producer and the customer during the purchase of the software. Rather, an untrusted dealer may sell the software, with no need for immediate communication with the software producer (see Section 3.2). This communication is essential in AM, and may present quite an obstacle in software distribution.

5. PPS does not require communication between the software producer and the system integrator before the protection of each product. This communication is essential in AM and presents another obstacle in software distribution. Also, the added transmissions may be tapped and altered, and the security is endangered.

6. PPS provides a protocol that enables the replacement of a malfunctioning CPU by untrusted servicemen, without requiring the physical transfer of a new CPU from the producer. AM requires the physical transfer of a new CPU.

7. The motives of all the parties involved in the usage of the protection method (CPU producers, system integrators, software producers, etc.) are similar in both methods. Those motives are discussed in depth in [AM84]. We will not repeat these arguments.

8. AM allows the system's OEM (Original Equipment Manufacturer) to require a fee from software producers for each usage of the system to protect software. By a simple variant to PPS the same result may be achieved. We will not discuss this here.

2. Implementation of PPS with Public-Key Cryptosystem (PPS/PK)

The implementation of PPS requires encrypting functions inside the CPU. The encryption may be done by a public-key cryptosystem (PKCS), such as [RSA78], or by ordinary encryption methods, such as [DES77]. In this section we will describe the implementation by a PKCS, denoted PPS/PK. This implementation is more straightforward; however, since no implementation of a PKCS seems both secure

and quick, the implementation by conventional cryptosystems seem to be more reasonable. The concept of PKCSs has been first suggested in [DH76], and several implementations - as well as numerous applications - have been published since then [M83].

A PKCS based on a set of pairs of functions $\{<E_i,D_i>\}$ such that

C1. $D_i E_i = E_i D_i = 1$

C2. Knowing E(M) and E, but not D, does not reveal anything about M.

C3. Knowing D(M) and M does not reveal D.

We use E to denote the encrypting function (or key), and D or E^{-1} for the Decyphering function (or key).

With each computer unit u, associate a pair of keys $<E_u,D_u>$, and with Z associate a pair of keys, $<E_z,D_z>$. Every computer unit C_u contains the following information:

1. D_u - The decyphering (secret) key of C_u
2. E_z - The encrypting key of Z
3. $D_z E_u$ - The encrypting key of C_u, signed by Z.

For indirect distribution via a software dealer U, another key is required in the dealer's computer - $F_u(i)$:

4. $F_u(i)$ - The software producer sells his or her software to the dealer with this key. The key is changed between sales.

The keys D_u, $F_u(i)$, and E_z, are kept hidden inside the CPU itself. They may not be accessed by the CPU instructions, except the special instructions that implement PPS. The signature of Z, denoted D_z, is even more secret: it is not kept in the CPU at all. On the contrary, E_u may be used quite easily (and is not a secret).

The cryptographic utilities required for PPS are only trapdoor functions. Actually, we only require $D_u E_u = 1$ for every computer u, and $E_z D_z = 1$.

The cryptosystem may be commutative, i.e. $E_a E_b = E_b E_a$. Several PKCSs have this property, including [RSA78], and some protocols are not secure with commuting cryptosystems. If other properties are known for the cryptosystem, analysis as in Section 4 should be done.

2.1. Direct software distribution protocol (PPS/PK)

The protocol that a user U with computer C_u should follow in order to buy PPS/PK protected software from its producer P is the direct distribution protocol outlined below. Note that information should pass only once from the user to the producer and vice versa. The notation used for a user U sending a message M to his computer C_u or to another party B is : (U,M,C_u) or (U,M,B) respectively.

D1. $(U,D_z E_u,P)$ - The user U sends to P the encryption key E_u signed by Z.

D2. $(P,(D_z E_u,PGM),C_p)$ - The producer P enters the encryption key of the customer's computer signed by Z and the program to be distributed, PGM, into his computer.

D3. $(C_p,[E_z D_z E_u]PGM,P)$ - The encryption procedure E_z is known to all computers, but hidden from the users.

D4. $(P,E_u PGM,U)$ - The user receives the software package.

D5. $(U,E_u PGM,C_u)$ - Loading the program.

D6. $(C_u,O(D_u E_u PGM),U)$ - The computer C_u (but not U) knows D_u. While executing, the code PGM is hidden inside the processor. The operation (run) of software P by a computer is $O(P)$.

It is assumed that knowing O(PGM) does not enlighten the intruder about PGM.

2.2. Indirect software distribution protocol (PPS/PK).

Usually software is not sold directly from the producer to the customer, but rather it is sold via a third party, the software dealer. Even telephone connection with the producer should, in these cases, be avoided. The direct software

distribution protocol, described in Section 3.2, is not suitable here, since the producer may rarely rely on the honesty of all the dealers. PPS provides a special protocol for indirect software distribution. This protocol requires one extra key hidden inside the dealers' CPU. The extra key is changing in each execution of the protocol. This temporal key, $F_u(i)$, (of a dealer U) is assumed to be the key of a conventional cryptosystem (although it could be implemented with a PKCS as well). The protocol is divided into two phases. In the first phase, the dealer L buys token programs from the producer. The tokens are converted to useful programs by the dealer's computer, C_L, in the second phase. Each token produces no more than one useful program, encyphered with the key of some buyer's computer. The initial key $F_i(0)$ is known only to the software producer. For example, $F_i(0)$ may be initiated in C_L by the producer before the computer C_L is given to the dealer.

The distribution protocol is outlined below. The first phase I1 is done for each token i to be used. Note that information should pass only once in each direction.

I1. $(P, F_i(i)[PGM, F_i(i+1)], L)$ - The producer P gives the dealer L a token i. This is the first phase of the protocol, and it may be done independently of the other phases.

I2. $(U, D_z E_u, L)$ - The user U sends his key to the dealer.

I3. $(L, (F_i(i)[PGM, F_i(i+1)], D_z E_u), C_L)$ - The computer C_L now contains key D_i that corresponds to token i.

I4. $(C_L, E_u PGM, L)$ - In the same time, C_L changes from key $F_i(i)$ to the new key $F_i(i+1)$. The new key that is given in the token!

I5. $(L, E_u PGM, U)$ - From this step on, the protocol is the same as the direct distribution protocol. The user receives the software package.

I6. $(U, E_u PGM, C_u)$ - Loading the program.

I7. $(C_u, O(D_u E_u PGM), U)$ - The computer C_u (but not U) knows D_u. While executing, the code PGM is hidden inside the processor. Several $F_i(i)$ mechanisms may be implemented in the same processor. Also, processors dedicated to users need not have $F_i(i)$ at all.

2.3. The replacement protocol (PPS/PK).

If the CPU of a user malfunctions, a new CPU must be provided. An essential property of the new CPU is being completely compatible: every software run on the old CPU should also run on the new one. To enable the new CPU to run PPS/PK protected software, it must have the same keys as the old one. A similar requirement ensues from upgrades to the CPU, when CPU replacement is required.

The new CPU must be made available as soon as possible. It should be possible for several service centers to make available a CPU to replace any malfunctioning CPU in their territory. Obviously one cannot permit such service centers to produce CPUs and determine their keys at will. We present a solution in which deceptions are likely to be discovered or prevented, and even if deception is committed by the service center, no more than one illegal CPU will be obtained. Those results are formally proved in Section 4.3.

The solution we suggest to this problem requires the remote help of Z. However, this help is only remote (by communication), and does not require physical interaction with Z, as in [AM84]. The protection will not fail even if the communication is tapped or altered.

Every CPU replacement will require Z's intervention. After the CPU has been replaced, Z must verify that a replacement has in fact taken place (for example, by receiving the malfunctioning CPU and verify it's identity). The service center S uses the remote help of Z to convert a spare computer C_s (with keys E_s and D_s) into a replacement for C_u. After the successful completion of the protocol, C_s will have keys E_u and D_u. The replacement protocol is outlined below.

R1. $(U, D_z E_u, S)$ - User U requires replacement CPU from S.

R2. $(S, (D_z E_u, D_z E_s), Z)$ - The Serviceperson asks Z for a transformation key that will change the key of the spare CPU C_s from E_s, D_s to E_u, D_u.

R3. $(Z, E_s(D_u; replace), S)$ - By the creation tables, Z finds for E_u the corresponding D_u. Then Z encrypts D_u - concatenated with a predefined string - by E_s, and sends it to S.

R4. $(S, E_s(D_u; replace), C_s)$ - Installation of new key in C_s. The key D_u will be installed only if it is concatenated with the correct string. The public key $D_s E_u$ is installed too.

R5. The CPUs may be replaced. The replaced CPU ought to be returned to Z and its number verified.

3. A Formal Analysis of PPS/PK

The presentation of any nontrivial security protocol or system would not be complete without a formal representation of the assumptions and formal proof of security. Therefore, we prove that, under acceptable assumptions, PPS/PK is secure. This is done using the Transaction System Model [HP85]. We proceed by describing the essence of the model and the correspondence between the model and PPS/PK. The model as described below is a simplified version of the transaction model for systems in which the timing is irrelevant to the security. Merritt [M83] also presented a formal model for analyzing the security of protocols.

The formalization of cryptographic protocols enables a precise inspection of the arguments of security. In the case of PPS, the reader is encouraged to inspect if the formal model is truly derived from the assumptions and protocols, and if the proofs of the security of the model are valid.

3.1. The essence of the Transaction Model.

A *Transaction System* (TS) is a partial algebra, defined by a domain and a set of relations on that domain. The domain of a TS is considered as the set of all the possible states of some information system. A state is defined by a set of variables. One of the variables is the set of all the messages transmitted so far. The set of messages transmitted is known to the attackers, since they have complete control over the communication lines. A *state* S is a set of values of all the variables. The relations on the domain represent the possible inferences available for the attacker. The relations are grouped into meaningful sets, called *Transactions*.

Each transaction is a set of ordered pairs of states. A *Transaction System* $TS=(T,S)$ is defined by a set of transactions T on a set of states S.

The definition of a TS does not yet ensure that the TS represents the real world correctly. A TS would be *correct* if all the possible inferences for the attacker from a given state, and no impossible inferences, may be obtained by executions of transactions from that state. For example, inferences include the innocent activities of other participants, usage of properities of functions used, etc.

A pair of states (S_i,S_{i+1}) of a TS is an *ordered pair*, with S_i termed *Tail* and S_{i+1} termed *Head*, if S_{i+1} is the result of applying some transaction of TS on S_i. A sequence of states $S_0,S_1,\ldots$ is a *history*, starting from S_0, if for all $i\geq 0,(S_i,S_{i+1})$ is an ordered pair. The *length* of a history is the number of states in the sequence. A state S_i is *i-reachable* from state S_0 if there exists a history H of length $i+1$ which starts at S_0 and ends at S_i, and no shorter history exists from S_0 to S_i. If there exists an i such that state S_i is i-reachable from state S_0, then S_i is *reachable* from S_0. If a state S_k is not reachable from state S_j, we say that S_j is *harmless* for S_k. A set of states is reachable if any of the states in the set is reachable. Similarly, we define the harmless property for a set of states.

We state without proof some elementary and intuitive results. The proofs are simple, and are given in [HP85].

Lemma 4.1 proves the transitivity of the reachability property.

LEMMA 3.1. *If a state S_i is i-reachable from S_0, then every state S_j, j-reachable from S_i, is $(j+i-1)$-reachable from S_0.*

Theorem 4.1 proves that the results obtained will hold for more restricted cryptosystems, for example - without commutativity between cryptographic operators.

THEOREM 3.2. *Let S be a set of states harmless for a set of states D_s in TS, then in every TS' s.t. Transactions$(TS')\subseteq$Transactions(TS), state S is harmless for D_s.*

3.2. PPS/PK as a TS.

The protocols detailed in Section 3 for PPS/PK execution correspond to the following TS called PPS/PK, under the assumptions listed below:

A. Information hidden inside a processor cannot be read.

B. Resurrecting the software by observing the ports outside the CPU is infeasible.

C. The cryptosystems used are secure. The security requirements have been detailed in Section 3.

D. The producer verifies faultlessly the identity of the user that sent the payment, and always delivers the software. The payment could have been implemented in the protocol, but it seemed unnecessary.

E. No information leaks from Z (except by the replacement protocol).

F. All the keys are cryptographically independent - no key may be obtained by known manipulations of other keys. The notion of cryptographic independence is formally defined in [HP85].

For proving the safety of PPS/PK we need consider only one producer of software, P. All the attackers may, however, use the protocol as if they are producers For the analysis, assume that all the users are attackers (since the attackers can impose as honest users). The variables of PPS/PK are: X is the total expenses of the attackers, for every user u, K_u is the decyphering key of his computer C_u. Initially K_u contains D_u. During a CPU exchange, a K key of a spare computer is changed to the D key of the failing computer. For every dealer L, Q_i has the same rule as the K key for the the temporal key $F_i(i)$. The set M of all the messages transmitted so far, which corresponds to the information held by the attackers.

The only source of information in PPS is the defined transactions. Therefore if PPS is in any given state, then that state is reachable from some initial state in which no messages were sent. The transactions of PPS/PK for computers C_u and C_w are listed in Table 1. In the table, P denotes a program to be sold by some

software producer for the sum of money - *cost* (by T12, T13 and T14). An application of operator a on string b is denoted $a(b)$. We omitted the brackets where there was no danger of confusion.

The TS model is a worst case analysis of the system. Therefore, data and keys are interchangeable (a key may be used as data and vice versa). Also, knowing the key of a cryptofunction is equivalent to knowing that cryptofunction. Therefore any string or key may be 'applied' to any string or key. This application may be done implicitly in some of the transactions, or directly by the attacker (by T7). When a transaction is explicitly used in one of the protocols, we note the step in the protocol. For example, T9 is used in D5 (step D5 of the distribution protocol).

The transactions basically represent the capabilities of the attackers. If an attacker manages to use some transaction with proper input, the table shows the output and the change in the system. The results of a transaction are added messages ("output") or a change for the variables X, Q or K. In the table, before any transaction is used, assume $K_u = D_u$ and $Q_u = F_u(i)$.

Some of the transactions will not be available in certain implementations. For example, the transactions that present the commutativity of the PKCS will not be present with a non-commuting PKCS. But, from Theorem 4.1 the security properties that were proved, hold as well without those transactions. Transaction T18, physically violating the CPU integrity, would not be considered part of PPS/PK. The TS that includes all the transactions, including T18, denoted as PPS/PKV, would be referred to only in the last theorem.

Notes: see the description of PPS and verify that all the steps in the protocols are performed by those transactions. We do not differentiate between operators and strings. When a string should be used as an operator, we use it as a key for the cryptographic operator.

A special kind of attack may be performed by an attacker which is also serviceperson. Such an attacker might accepts replacement for a CPU from Z without returning the original CPU. This attack causes expanses to the attacker (including

risk); those expense are by denoted R. Theorem 4.6 shows that after using T17, there is no way to get more then two CPUs that use the same key (that originally belonged only to one of them). This ensures also that if the CPU have been replaced properly, the attackers will have only one CPU with the old key, and therefore with no gain.

Another extreme attack is physically violating the enclosure of the CPU, to find the keys hidden within (T18). The expense of this attack is denoted by V. Theorem 4.8 shows that when PPS is implemented by PKCS, even if T18 is used, the attacker must still use T12 with $E_z(E_u)$, where u is the identity of the attacker's computer, to obtain the uncyphered program P. This result enables enforcement of auditing means against such attacks.

3.3. Proofs of PPS/PK security.

The next lemma shows that no attacker can forge the signature of Z. The discussion in this section refers always to PPS/PK, except where stated otherwise.

LEMMA 3.3. *If $S=(M,X,K)$ is reachable from $S_0=(null,X_0,K_0)$, where $D_z a \in M$, then there exists computer C_u and b such that $a=E_u b$.*

Proof. Only T10 produces a message that includes D_z, therefore $D_z E_u$ must have been manipulated to produce $D_z a$. To remove E_u only T9 or T1 can be used. But the only result of T9 is operated by 0 and there is no transaction that removes 0. The use of T1 to remove E_u requires an input string that includes D_u but not E_u. But no transaction produces such a string. Therefore $D_z a$ cannot be produced unless $a=E_u b$. ■

Theorem 4.4 shows that the attacker cannot reproduce the decyphered code P, given the encrypted program by T12 or T13. The producer's computer uses E_z on the input string x sent by the user, to produce the encryption for the program P. This is given by $[E_z x]P$ for any string x. Reproducing the encrypted program implies $P \in M$.

THEOREM 3.4 *If $S_1=(M_1,X,K)$ is a harmless state for $W=W_a \cup W_b$, where $W_a=\{(M,X,K)\mid P\in M\}$ and $W_b=\{(M,X,K)\mid \exists\ D_u$ such that: $D_u\in M\}$, then $S_2=(M_1\cup[E_x x]P,X,K)$ is harmless for W.*

Proof. By contradiction, assume W is reachable from S_2. Since S_1 is harmless for W, then $m_2=[E_x x]P$ has been used to reach W. The only transaction, when D_u is unreachable, that removes E_x is T10, where $x=D_x E_u$. Therefore it remains to show that $S_3=(M_1\cup E_u P,X,K)$ is harmless for W. However, there is no transaction that removes E_u when D_u is unreachable. Thus, both W_a and W_b are unreachable, since both require the removal of E_u and D_x. ■

We have shown the original code cannot be obtained. Now we prove that the code cannot be 'adjusted' to another computer, i.e. no manipulation to the encrypted code produces code encrypted by a key of a different CPU. The idea of the theorem is that if an attacker can't get a program without paying, then he can't get two programs without paying twice the price of the program.

THEOREM 3.5. *If $S\in\{(M,Y,K)\mid Y<cost\}$ is a harmless state for some set of states U_1 defined below, then it is also harmless for U_2. Where: $U_1=\{(M,X,K)\mid (X<cost)\&(E_u P\in M)\&(K_i=D_u\neq null)\}$ and $U_2=\{(M,X,K)\mid ((X<min(2*cost,R))\&(E_u P,E_w P\in M)\&(j\neq i)(K_i=D_u\neq null)(K_j=D_w\neq null)\}$*

Proof. If $E_u P\in M$, T12 or T13 must have been used. By Theorem 4.4, P is not in M. If T13 have been used to reach $E_u P\in M$ from S, then T14 must have been used before since it is the only transaction that produces $F_u(i)[P,F_u(i+1)]$. But if T14 occurred, it must have been in a history reachable from S, since $Y<cost$. In order to prove that U_2 is not reachable from S, we notice that T12 and T14 cannot be used twice. Also, from the arguments above, T13 cannot be used again. Therefore $E_w P$ cannot be produced by T12 or T13, and since no transaction that removes E_u it remains to show that no two computers can have the same key. That is for every two computers i,j where $i\neq j$, $K_i=K_j=D_1\neq null$. In order to get a second key transactions T17 or T16 must be used. Since X<R in U_2, only T16 can be used, but the application of T16 change K_w to null. ■

If the decyphered code is unreachable, as we showed in Theorem 4.4, and we cannot encrypt the code for another CPU, according to Theorem 4.5, there still remains an alternative: to generate several computers with the same keys. Then the attacker shall have to pay only for one copy, and actually obtain several copies.

This attack cannot be prevented completely, since we must permit replacement of CPUs (see Section 3.3). Indeed the same problem exists in the other software protection methods, and the solutions available are usually rather unsatisfactory [HK84].

It is now proved that all the CPUs with the same keys, except one, should be returned to Z. Therefore the effect of these attacks is minimal. Given two computers with different keys, T17 must be used in order to make the keys of both computers equal and meaningful. Meaningful keys are keys that decypher programs distributed by T12 or T13.

THEOREM 3.6. *Let* $S_0=(M_0,X_0,K_0)$ *be a state such that* $M_0=\phi$ *and* $X_0=0$ *and all the keys are cryptographically independent. Then S is harmless, for* $U=\{(M,X,K)\mid(j\neq i)(K_i=K_j=a^{-1}\neq null)\&(D_a a\in M)\&X<R\}$.

Proof. Since $X<R$ in U, T17 cannot be used. The only transaction that changes keys is T15; but in order to use it, T16 must be employed. But if T16 has been used to produce $E_u(D_w;replace)$, where $K_i=D_u$ and $K_j=D_w$ before T16, then $K_j=null$ after T16, and since T15 may be used only for C_i, S is still harmless for U. ∎

We state somewhat unformally and without proof the following theorem, which finds the expenses of the attacker for obtaining n computers with identical keys.

THEOREM 3.7. *If S is harmless for* $U=\{(M,X,K)\mid(X<R)\&(i\neq j)(K_i=K_j\neq null)\}$ *then it is harmless for* $W=\{(M,X,K)\mid(X<R*log_2(q))\&(\mid I\mid=q)\&((i,j\in I)=>(K_i=K_j\neq null))\}$.

The next result is, perhaps, of minor importance . We prove that even if T18 is used, and all the keys in a CPU are revealed, the attackers cannot forge the signature of Z. Thus the attackers still have to order software by sending the correct

public key. This result holds only when PPS is implemented using PKCS. We denote PPS/PKV to be PPS/PK with the addition of T18. Let V be the price for violating the integrity of the CPU.

THEOREM 3.8. *In PPS/PKV, if S is harmless for $U_1=\{(M,X,K)|(D_z a \in M)\&(X<V)\}$ then it is harmless for $U_2=\{(M,X,K)|(D_z a \in M)\}$.*

Proof. There is no transaction ,including T18, that performs D_z on a given string. ■

4. PPS Implemented with Conventional Cryptosystem.

Implementing PPS by PKCS is quite natural, but also quite difficult. No chip available performs a PKCS, and the security of PKCS is still in doubt. Conventional cryptosystems are more mature. Several methods have been implemented in integrated circuits and are considered quite secure. Most known is [DES77].

The implementation of PPS by a conventional cryptosystem is based on emulating the required properties of PKCS by adding redundant information. Two features of PKCS are used in PPS:

1) Signatures - used to ensure that keys are not invented.
2) Secrecy - the program is encrypted by the distributor, yet he cannot decypher programs encrypted by other distributors.

4.1. PPS/C.

When using conventional cryptosystems, the signatures implemented with PKCS before, are now implemented by the processors. Each processor C_u contains three hidden keys:

1) K_z - The key of Z.
2) K_u - Computer's key
3) $F_u(i)$ - Temporal key for indirect software distribution

The emulation is performed by implementing E_x, D_x with conventional keys and the protocols are given in the following sections. Section 5.5 contains the corresponding transactions, which forms a TS denoted by PPS/C.

We assume the cryptosystems are secure , i.e. an attacker cannot determine m from $K_a(m)$, without knowing K_a. It is also impossible to find K_a from m and $K_a(m)$. Most cryptosystems are presumed to be secure in this manner. Note that we permit the encryption to be commutative, i.e. $K_a K_b(a)=K_b K_a(a)$.

4.2. Direct software distribution protocol (PPS/C).

The following is the protocol for direct distribution of software, from producer P to the user U. The words *key*, *prog* and *replace* are predefined strings used in the protocol. It is implicit that, whenever possible, honest participants in the protocol check for those strings in the input.

D1. $(U,K_x(K_u;key),P)$ - The user sends key K_u signed and hidden by K_x.

D2. $(P,K_x(K_u;key),PGM),C_p)$ - The producer enters both users' key and program into his computer...

D3. $(C_p,K_u(PGM;prog),P)$ - The encrypted program is given to the producer.

D4. $(P,K_u(PGM;prog),U)$ - The producer transfers the encrypted program to the user.

D5. $(U,K_u(PGM;prog),C_u)$ - The user gives his computer the encrypted program.

D6. $(C_u,O(PGM),U)$ - Th computer executes the program.

4.3. Indirect software distribution protocol (PPS/C).

The following is the protocol for indirect distribution of software, from producer P to a user U via a dealer L.

I1. $(P,F_u(i)[PGM,F_u(i+1)],L)$ - Producer P sells token i to dealer L. This step may be done (for several tokens) before the other steps of the

protocol.

I2. $(U, K_z(K_u;key), L)$ - User U sends his public key to the dealer.

I3. $(L, (K_z(K_u;key), F_u(i)[PGM, F_u(i+1)]), C_i)$ - The dealer uses token i.

I4. $(C_i, K_u(PGM;prog), L)$ - The encrypted program is given to the dealer. In the same time, C_i changes from $F_u(i)$ to $F_u(i+1)$.

I5. $(L, K_u(PGM;prog), U)$ - From this step - same as direct distribution.

I6. $(U, K_u(PGM;prog), C_u)$ - The user enters the program into his computer.

I7. $(C_u, O(PGM), U)$ - The computer executes the program.

4.4. CPU replacement protocol (PPS/C)

The following protocol in PPS/C is for the replacement of a users' CPU. The serviceperson S replaces C_u with C_z, by the help of Z.

R1. $(U, K_z(K_u;key), S)$ - User U sends his key to S.

R2. $(S, (K_z(K_u;key), K_z(K_s;key)), Z)$ - The serviceperson sends both keys to Z.

R3. $(Z, K_s(K_u;replace), S)$ - Note that Z, in PPS/C, does not have to keep track of the keys.

R4. $(S, K_s(K_u;replace), C_z)$ - New keys installation.

R5. The CPUs are replaced. The replaced CPU ought to be returned to Z.

4.5. PPS/C as a TS

The transactions of PPS/C are listed in Table 2 for computers C_u, and C_w. The variables of PPS/C are: X is the total expenses of the attackers, for every user u, K_u is the key of his computer C_u. For every dealer L, Q_i is the temporal key $F_i(i)$. The set M of all the messages transmitted so far, which corresponds to the information held by the attackers.

Theorems 4.3-4.8 may be proved for PPS/C, but since they are simple and similar to the proofs for PPS/PK, we will not give them here.

5. Conclusion

The problem of software piracy causes considerable losses to software producers. The scheme presented - PPS - provides proved, reliable protection, and convenient protocols for distribution of software and replacement of CPUs. PPS requires implementation of cryptographic capabilities - public key or conventional key - inside the CPU. This is a challenge for all CPU manufacturers !

We believe that by using suitable protection methods software piracy could be rendered obsolete. Such a step will be to the benefit of all the parties involved (well, almost ...).

6. Acknowledgments

We thank Mr. Gadi Karmi for his proofreading.

7. References:

[AM84] D.J. Albert and S.P. Morse, "Combating Software Piracy by Encryption and Key Management", *Computer* April 1984

[DES77] National Bureau of Standard, "Data Encryption Standard", *Federal Information Processing Standard Publication 46*, January 1977

[DH76] W. Diffie and M. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory, Vol. IT-22*, 1976

[HK84] A. Herzberg and G. Karmi, "On Software Protection" *Proc. Fourth JCIT, Jerusalem, Israel* April 1984

[HP85] A. Herzberg and S. Pinter, "The Transaction System Model and Security Engineering", *in preparation*

[M83] M.J. Merritt, "Cryptographic Protocols", *GIT-ICS-83/06, doctoral dissertation, The Georgia Institute of Technology*, 1983.

[RSA78] R.L. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and PKCs", *Comm. ACM, Vol 21, No. 2* (Feb. 1978).

Table 1: Transactions of PPS/PK.

T#	Input	Output	Change	Steps
T1	$D_u E_u a$	a	--	D3
T2	$E_u D_u a$	a	--	--
T3	$E_u D_w a$	$D_w E_u a$	--	--
T4	$E_u E_w a$	$E_w E_u a$	--	--
T5	$D_u D_w a$	$D_w D_u b$	--	--
T6	$D_u E_w a$	$E_w D_u a$	--	--
T7	a,b	a(b)	--	--
T8	a	O(a)	--	--
T9	$E_u a$	O(a)	--	D5,I7
T10	--	$D_z E_u$	--	D1
T11	$D_z a, b$	a(b)	--	D2,D3
T12	$D_z a$	a(P)	X=X+cost	D2,D3
T13	$D_z a, F_u(i)[P, F_u(i+1)]$	a(P)	$Q_u = F_u(i+1)$	I3,I4
T14	--	$F_u(i)[P, F_u(i+1)]$	X=X+cost	I1
T15	$E_u(a; replace)$	--	$K_u = a$	R4
T16	$D_z E_u, D_z E_w$	$E_u(D_w; replace)$	$K_w = null$	R4,R5
T17	$D_z E_u, D_z E_w$	$E_u(D_w; replace)$	X=X+R	--
T18	--	D_u, E_z, Q_u	X=X+V	--

TABLE 2: : TRANSACTIONS OF PPS/C

T#	Input	Output	Change	Steps
T1	$K_u K_u a$	a	–	–
T2	$K_u K_w$	$K_w K_u$	–	–
T3	a,b	ab	–	–
T4	a	C(a)	–	–
T5	$K_u(a,prog)$	O(a)	–	D5,I7
T6	–	$K_z(K_u,key)$	–	D1,R1,R2,I2
T7	$K_z(a,key),b$	a(b,prog)	–	D3
T8	$K_z(a,key)$	a(P,prog)	X=X+cost	–
T9	$K_z(a,key),F_u(i)[P,F_u(i+1)]$	a(P,prog)	$Q_u=F_u(i+1)$	I3,I4
T10	–	$F_u(i)[P,F_u(i+1)]$	X=X+cost	I1
T11	$K_u(a,replace)$	–	$K_u=a$	R4
T12	$K_z(K_u,key),K_z(K_w,key)$	$K_u(K_w,replace)$	$K_w=null$	R4,R5
T13	$K_z(K_u,key),K_z(K_w,key)$	$K_u(K_w,replace)$	X=X+R	–