Public Protection of Software

AMIR HERZBERG and SHLOMIT S. PINTER
Technion—Israel Institute of Technology

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 enable software protection, without causing substantial 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 criteria.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*cryptographic controls*; E.3 [**Data**]: Data Encryption—*public key cryptosystems*; K.5.1 [**Legal Aspects of Computing**]: Software Protection

General Terms: Algorithms, Design, Security

Additional Key Words and Phrases: Cryptographic protocols, protected CPU, security protocols, single key cryptosystems, software authorization, software distribution, software piracy

1. INTRODUCTION

Great losses to software producers are currently incurred owing 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 importance of software protection has resulted in many products that supply the means for protecting software. It is shown in [6] that many commercially available means suffer from some of the following deficiencies:

(1) Insufficient protection.
(2) Impaired backup and networking capabilities (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.

One common protection scheme uses special "signature" information in the storage media, which cannot be duplicated by conventional methods (e.g., [3]).

Authors' address: Technion-Israel Institute of Technology, Department of Electrical Engineering, Haifa 32000, Israel.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1987 ACM 0734-2071/87/1100-0371 $01.50

The major faults of such methods are 1, 2, and 5 above. Another method attaches a hardware device to the CPU, which is used for identification. An attempt has been made to standardize this method (see [1]). The major faults of those systems are 1, 3, 4, and 5.

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 [6]. In contrast with the deficiencies outlined above, PPS provides

(1) Provable, hence reliable, protection (under acceptable and well-defined assumptions).
(2) Undisturbed backup and networking capabilities (by limiting execution only to a specific CPU).
(3) Applicability to 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. However, a special coprocessor could implement PPS and operate with existing processors. The protected routines of the software would be run on the PPS coprocessor. In recent papers [2, 13, 14], two other software protection methods (henceforth referred to as AM and SPS) were presented. These methods require 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 and SPS do not provide protocols for those functions. A detailed comparison of PPS versus AM and SPS 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). In 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 seems to be much more realistic. Section 6 discusses the practical aspects of a PPS system and Section 7 gives the final conclusions.

2. 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 trap-doors 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 Original Equipment Manufacturer (OEM) is Z, then the above 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. For obvious reasons, a new CPU, which 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. Although an appropriate procedure should protect against this hazard, 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, does PPS provide some protection against this attack. After violating the integrity of a CPU, the attackers will only be able to decrypt protected code encrypted for that CPU.

2.1 PPS versus AM and SPS

(1) The modifications of PPS to the architecture of the CPU can be like those detailed in [2]; other solutions are being tested too (see Section 6).

(2) The three 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, whereas AM requires public key cryptosystems, and SPS requires conventional cryptosystems. The implementation of public key systems is much harder, but by using it PPS may provide additional protection.

(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 SPS 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 and SPS require 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 the three methods. Those motives are discussed in depth in [2]. We will not repeat these arguments.

(8) AM allows the system's OEM 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.

(9) Both AM and SPS explicitly use an execution key in order to protect the programs; thus they each transfer only the key that uses cryptographic protocols. The program itself, enciphered by the above execution key, is distributed. When the execution key is used, the size of the communicated message is reduced, and the program may be decoded more efficiently than the key. The protocols of PPS may be used to transfer the execution key (instead of the program itself) to accomplish the same results.

3. 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 [12] or by ordinary encryption methods, such as [11]. 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 seems to be more reasonable. The concept of PKCSs was first suggested in [5], and several implementations, as well as numerous applications, have been published since then.

A PKCS is based on a set of pairs of functions $\{\langle E_i, D_i \rangle\}$ such that

C1. $D_iE_i = E_iD_i = 1$

C2. For every message M, knowing $E_i(M)$ and E_i, but not D_i, does not reveal anything about M. We use E_i to denote the encrypting function (or key), and D_i or E_i^{-1} for the decrypting function (or key).

C3. For every message M, knowing M and its encryption (decryption) does not reveal the encryption or decryption keys.

In some cryptosystems, the keys are composed of several parts (in RSA [12], for example, $E_i = (n_i e_i)$).

With each computer unit C_u a pair of keys $\langle E_u, D_u \rangle$ is associated, and a pair of keys $\langle E_z, D_z \rangle$ is associated with Z. Every computer unit C_u contains the following information:

(1) D_u—the decrypting (secret) key of C_u.
(2) E_z—the encrypting key of Z (not a secret).
(3) $D_zG(E_u)$—the encrypting key of C_u, signed by Z (not a secret). The signature of Z involves, first, the verification that E_u is a proper encryption key. This

is done with the application of G (for example, checksum). Second, the secret key D_z is being applied. The resulting key $D_zG(E_u)$ is the public key given to the user.

We denote by G^{-1} the inverse of G and use it to verify the validity of $D_zG(E_u)$. Therefore G must introduce a high degree of redundancy (say 100 bits) in order to prevent the attacker from producing a faked $D_zG(a)$ by exhaustive search. For each message a, $G^{-1}(G(a)) = a$, and for each string b for which there is no message a s.t. $b = G(a)$, then $G^{-1}(b) = error$.

For indirect distribution via a software dealer L, another key, $F_L^{-1}(i)$, is required in the dealer's computer C_L.

(4) $F_L^{-1}(i)$—the software producer sells software to the dealer with this key. The key is changed between sales (i is the sale number). This key is replaced by a counter in the modified indirect-distribution protocol, given in Appendix B.

The keys D_u, $F_L^{-1}(i)$, and $F_L(i)$ 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, $D_zG(E_u)$ and E_z are not secret. However, there is no need to publish E_z.

A list of symbols used with their meaning is given in Table I.

The cryptosystem may be commutative, that is, $E_aE_b = E_bE_a$. It may also be associative, that is, $E_a(E_bE_c) = (E_aE_b)E_c$. During the security analysis (see Section 4) all the properties of the cryptosystem need to be considered.

3.1 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 when a message M is sent by user U to computer C_u or to another party B is (U, M, C_u) or (U, M, B), respectively.

D1. $(U, D_zG(E_u), P)$—The user U sends to P the encryption key E_u signed by Z.[1]

D2. $(P, (D_zG(E_u), PGM), C_p)$—The encryption key of the customer's computer signed by Z and the program PGM to be distributed are entered into the computer C_p of producer P.

D3. $(C_p, [G^{-1}E_zD_zG(E_u)]PGM, P)$—The encryption procedure E_z and the verification procedure G^{-1} are applied by C_p.

D4. (P, E_uPGM, U)—The user receives the software package.

D5. (U, E_uPGM, C_u)—The program is loaded.

[1] This protocol assumes the producer verifies the identity of the user who paid and always delivers the software (i.e., the software was sold at the store). To enable distribution of software over a network, small modifications to D1 and D2 are needed and are presented in Appendix A.

Table I. Symbols Used

Symbol	Meaning
Participants	
U, W	Users.
C_α	Computer that belongs to party α of the protocols.
P	The software producer.
Z	The key generating body (the center).
L	The dealer.
S	The service shop. Its computer C_s serves as replacement for failing CPUs.
Variables	
M	Set of messages known to the attacker.
K_α	The secret register for the key of computer C_α (originally holds D_α).
K_z	Secret key of the generating body Z (PPS/C).
D_z	Secret key of the generating body Z (PPS/PK).
null	A special key that makes the CPU nonoperational.
Q_α	A secret register that holds the distribution key in C_α.
a, b, c	Variables used in Tables II and III to denote parameters set by the user of a transaction.
$CNT(i)$	Array of counters used in the alternative indirect distribution protocol (Appendix B).
Operators	
$O(PGM)$	Operation (execution) of program PGM on a processor.
G, G^{-1}	Redundancy generator (G) and verifier (G^{-1}) s.t. $G^{-1}G(M) = M$.
Values	
X	Total expenses to the attackers.
PGM	The program.
D_α, E_α	Decryption and Encryption keys (respectively) initially set for C_α.
R	Expenses estimated for cheating Z by not returning replaced CPU.
V	Expenses estimated for violating the integrity of a CPU to get secret keys.
$F_L(i)$	A key for the ith distribution of software by dealer L.
key, prog, make, order, copies, priced, counter, replace, master	Strings signifying special operations when concatenated to an encrypted block.
$COST$	The cost of a program.

D6. $(C_u, O(D_uE_uPGM), U)$—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 PGM by a computer is denoted by $O(PGM)$.

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

3.2 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 3.1, 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 dealer's CPU. The extra key is used for decryption of a message from the producer and is changed at each execution of the protocol. 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, encrypted with the key of some buyer's computer. The key used to encrypt the ith token sent to dealer L is $F_L(i)$, and C_L decrypts the token using $F_L^{-1}(i)$. The initial keys $F_L^{-1}(0)$ are known only to the software producer. For example, $F_L(0)$ 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.[2] Step I1 is done for each token i to be used. Note that information should pass only once in each direction.

I1. $(P, F_L(i)[PGM, F_L^{-1}(i+1)], L)$—The producer P gives the dealer L a token i. This is the first step of the protocol, and it may be done independently of the other steps.
I2. $(U, D_zG(E_u), L)$—User U's key is sent to the dealer.
I3. $(L, (F_L(i)[PGM, F_L^{-1}(i+1)], D_ZG(E_u)), C_L)$—The computer C_L already contains key $F_L^{-1}(i)$ that corresponds to token i.
I4. $(C_L, [G^{-1}E_zD_zG(E_u)]PGM, L)$—By applying $F_L^{-1}(i)$, computer C_L finds PGM and $F_L^{-1}(i+1)$. The encryption procedure E_z and the verification procedure G^{-1} are being applied by C_L. At the same time, C_L changes register Q_L from key $F_L^{-1}(i)$ to the new key $F_L^{-1}(i+1)$. The new key is given in the token.
I5. (L, E_uPGM, U)—From this step on, the protocol is the same as the direct distribution protocol. The user receives the software package.
I6. (U, E_uPGM, C_u)—The program is loaded.
I7. $(C_u, O(D_uE_uPGM), U)$—The computer C_u (but not U) knows D_u. While executing, the code PGM is hidden inside the processor.

Several $F_L(i)$ mechanisms may be implemented in the same processor to enable the same dealer to deal with several producers.

3.3 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 complete compatibility: 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 may appear in CPU upgrades.

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

[2] Some variations on this protocol may be more suitable to other circumstances and also be more efficient. They can be found in Appendix B.

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 [2]. 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 occurred (for example, by receiving the malfunctioning CPU and verifying its 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. $(U, D_zG(E_u), S)$—User U requires a replacement CPU from serviceperson S.
R2. $(S, (D_zG(E_u), D_zG(E_s)), Z)$—The Serviceperson asks Z for a transformation key that will change the keys of the spare CPU—C_s from $\langle E_s, D_s \rangle$ to $\langle E_u, D_u \rangle$.
R3. $(Z, E_s(D_u, \mathit{replace}), S)$—For composing the message, Z applies on the message accepted at R2 the encryption procedure E_z and the verification procedure G^{-1} to obtain E_u and E_s. Then Z obtains D_u from E_u by using tables that contain all the key pairs or by using a trap-door function. Then Z encrypts D_u concatenated with a predefined string *replace* and sends it to S.
R4. $(S, E_s(D_u, \mathit{replace}), C_s)$—Installation of a new key in C_s. The key D_u will be installed only if it is concatenated with the correct string. The public key $D_zG(E_u)$ is installed too.
R5. The CPUs may be replaced. The replaced CPU ought to be returned to Z and its number verified.

4. 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 by using the Transaction System Model [7]. 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. Using the formal model of *hidden automorphism* [10] has been attempted but found to be complicated. The model used for proving the *ping-pong protocol* [4] cannot be used either. For example, modeling the replacement of keys in that model is impossible.

The reader is encouraged to inspect whether the formal model of PPS is truly derived from the assumptions and protocols, and if the proofs of security, based on the model, are valid. When implementing a protocol, the implementation should be checked for complete consistency with the formal model, for example, no new capabilities should be given to the attacker because of the use of a specific cryptosystem.

4.1 The Essence of the Transaction Model

We present a simple model that is used for describing systems and explore its security aspects. The model deals with *exposed systems*, that is, systems that execute distributed protocols, in which the attackers have complete control over the data transmitted. The attackers receive all messages and may alter or delete them at will. Users cannot identify the origin of the messages, except by recognizing information in the message itself. The model is used for ensuring safe states of the system.

Exposed systems are viewed as composed of honest users, attackers, and programmed processors. We are not concerned here with the correct execution of any protocol, but only with the prevention of some illegal actions. Thus, the model deals only with the capabilities of the attackers. The attackers can cooperate and share information freely and secretly, and they can cause the innocent users or processors to perform any operation that is included in the protocol.

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 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. 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 properties of functions used, and so forth.

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 $s_1, s_2, \ldots$ is a *history* starting from s_1, if for all $i > 0$, (s_i, s_{i+1}) is an ordered pair. A state s_i is *reachable* from state s_j, iff there exists a history H from s_j to s_i. Every state is also *reachable* from itself. 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. A set of states B is harmless for a set of states D if no state in D is reachable from a state in B.

We state without proof some elementary and intuitive results. The proofs are simple and are given in [7].

LEMMA 4.1. *The reachability relation is transitive.*

An important property implied by the following theorem is that a secure system, with some attackers and transactions, will surely stay secure if some of the attackers turned honest or some of the transactions were limited. Thus security results obtained from a system will hold for a more restricted version of

the system, for example, without commutativity between cryptographic operators. Therefore, it will suffice to analyze security for the most powerful coalition of attackers (referred to as *the attacker*).

THEOREM 4.2. *Given a transaction system $TS = (T, S)$, let $B \subseteq S$ be a set of states harmless for the set $D_s \subseteq S$ in TS. Then B is harmless for D_s in every $TS' = (T', S)$, such that $T' \subseteq T$.*

4.2 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:

(1) Information hidden inside a processor cannot be read.
(2) Resurrecting the software by observing the ports outside the CPU during the execution is infeasible.
(3) The cryptosystems used are secure. The security requirements have been detailed in Section 3.
(4) The producer verifies faultlessly the identity of the user who sent the payment and always delivers the software. The payment could have been implemented in the protocol, but it seemed unnecessary. Appendix A describes this modification.
(5) No information leaks from Z (except by the replacement protocol).
(6) All the keys are chosen independently—no key may be obtained by known manipulations of other keys.

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 were producers. For the analysis, assume that all the users are attackers (since the attackers can pose as honest users). The variables of PPS/PK are: X is the total expense of the attackers, and for every user u, register K_u holds the decrypting key of u's computer. Initially K_u is given the value D_u. During a CPU exchange, the decryption key K_s of a spare computer C_s is set to have the value D_u of the failing computer C_u. For every dealer L, Q_L should contain the value $F_L^{-1}(i)$ at the ith distribution. The set M of all the messages transmitted so far corresponds to the information held by the attackers. Therefore every state s in PPS/PK is defined by the quartet $s = (M, X, K, Q)$, where K is the set of decryption keys and Q is the set of the distribution keys held by the dealers.

The only source of information in PPS is the defined transactions (listed in Table II), which basically represent the capabilities of the attackers. Therefore, all of the operations available owing to the protocol must be present in the table. In addition, every property of the cryptosystem used in the specific implementation of the protocol must be present in the table. Otherwise the proof does not hold. If an attacker manages to use some transaction with proper input, the table shows the output and the change in the system (state). Therefore if PPS is in a 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 II. All the users (possible attackers) are allowed to behave as producers and distributers of software; therefore all the transactions and variables are defined for C_u

Table II. Transactions of PPS/PK

Transaction number	Input	Output	Change	Steps	Meaning
T1	$D_u E_u a$	a	—	D3	Decryption
T2	$E_u D_u a$	a	—	—	Encryption
T3	$E_u D_w a$	$D_w E_u a$	—	—	Commutativity
T4	$E_u E_w a$	$E_w E_u a$	—	—	Commutativity
T5	$D_u D_w a$	$D_w D_u a$	—	—	Commutativity
T6	$D_u E_w a$	$E_w D_u a$	—	—	Commutativity
T7	a, b	$a(b)$	—	—	Cipher
T8	a	$O(a)$	—	—	Execution
T9	$E_u a$	$O(a)$	—	D5, I7	CPU instruction
T10	—	$D_z G(E_u)$	—	D1	CPU instruction
T11	$b, D_z G(a)$	$a(b)$	—	D2, D3	CPU instruction
T12	$D_z G(a)$	$a(PGM)$	$X = X + COST$	D2, D3	CPU instruction
T13	$D_z G(a), F_u(i)[PGM, F_u^{-1}(i+1)]$	$a(PGM)$	$Q_u = F_u^{-1}(i+1)$	I3, I4	CPU instruction
T14	—	$F_u(i)[PGM, F_u^{-1}(i+1)]$	$X = X + COST$	I1	CPU instruction
T15	$E_u(a, replace)$	—	$K_u = a$	R4	CPU instruction
T16	$D_z G(E_u), D_z G(E_w)$	$E_u(D_w, replace)$	$K_w = null$	R4, R5	CPU instruction
T17	$D_z G(E_u), D_z G(E_w)$	$E_u(D_w, replace)$	$X = X + R$	—	By cheating
T18	—	D_u, E_z, Q_u	$X = X + V$	—	CPU violation
T19	$(\forall x)(E_u x)a$	a	—	—	Weak key
T20	a	$G(a)$	—	—	Certification
T21	a	$G^{-1}_{(a)}$	—	—	Verification
T22	$a(bc)$	$(ab)c$	—	—	Associativity
T23	$(ab)c$	$a(bc)$	—	—	Associativity

and C_w. The results of a transaction are changes to the variables X, Q_u, K_u or new messages ("output"). Before any transaction is used (initial state), assume that $K_u = D_u$ and $Q_u = F_u^{-1}(0)$. In the table, *PGM* denotes a program to be sold by some software producer for the sum of money, *COST*. An application of operator a on string b is denoted by $a(b)$. We omit brackets where there is no danger for confusion, and we do not differentiate between operators and strings; thus when a string should be used as an operator, we use it as a key for the cryptographic operator.

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 (with 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).

Some of the transactions will not be available in certain implementations. For example, the transactions that present the commutativity or associativity of the PKCS will not be present with a noncommuting or nonassociative PKCS. But, from Theorem 4.2 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. Transaction T19 represents the possibility, in some PKCS (including RSA), of finding a message that when enciphered by a known key would produce a "weak key." This has been noted by Referee B, and we have modified the protocol to be secure even when this transaction is legal. The idea is to check that the given key is a valid key by adding redundancy (using G).

A special kind of attack may be performed by an attacker who is also a serviceperson. Such an attacker might accept replacement for a CPU from Z without returning the original CPU. This attack causes expenses to the attacker (including risk) which are denoted by R. Theorem 4.6 shows that, after using T17, there is no way to get more than two CPUs that use the same key (that originally belonged only to one of them). This ensures also that if the CPU has 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 the PPS is implemented by PKCS, even if T18 is used, the attacker must still use T12 with $D_z G(E_u)$, where u is the identity of the attacker's computer, to obtain the decrypted program *PGM*. This result enables enforcement of auditing means against such attacks.

4.3 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. Let $\varnothing$ denote the empty set and $s_0 = (\varnothing, 0, K, Q)$ is the initial state.

LEMMA 4.3. *Let $s = (M, X, K, Q)$ be reachable from s_0.*

(*i*) *$D_zG(a) \in M$, then there exists computer C_u such that $a = E_u$.*
(*ii*) *For every computer C_u, the key $D_u \notin M$.*

PROOF. (i) Only T10 produces a message that includes D_z, i.e., $D_zG(E_u)$. Since G and G^{-1} are not associative (T22, T23 not applicable), there is no way to change the E_u operated by G or to remove $G(E_u)$. (ii) The only transactions that use D_u are T9 and T15. The output of T9 is operated by O which cannot be removed. Transaction T15 has no output. Therefore D_u cannot be found. □

The producer's computer uses E_z on the input string x sent by the user to produce the encryption for the program *PGM*. This is given by $[G^{-1}(E_zx)]PGM$ for any string x. Theorem 4.4 shows that the attacker cannot reproduce the decrypted code *PGM*, given the encrypted program by T12 or T13. Reproducing the encrypted program implies $PGM \in M$.

THEOREM 4.4. *If $s_1 = (M_1, X, K, Q)$ is a harmless state for $W = W_a \cup W_b$, where $W_a = \{(M, X, K, Q) \mid PGM \in M\}$ and $W_b = \{(M, X, K, Q) \mid \exists D_u \in M\}$, then $s_2 = (M_1 \cup \{[G^{-1}(E_zx)]PGM\}, X, K, Q)$ is harmless for W.*

PROOF. By contradiction, assume W is reachable from s_2. Since s_1 is harmless for W, then $m_2 = [G^{-1}(E_zx)]PGM$ has been used to reach W. The only transaction, when W_b is unreachable, that removes E_z is T10, where $x = D_zG(E_u)$. Therefore, it remains to show that $s_3 = (M_1 \cup \{E_uPGM\}, X, K, Q)$ (the result of T10) is harmless for W. However, there is no transaction that removes E_u when D_u is not known. Thus, both W_a and W_b are unreachable from s_2, since both require the removal of E_u and D_z. □

We have shown that the original code is not obtainable. Now we prove that the code cannot be "adjusted" to another computer, that is, 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 cannot get a program without paying, then the attacker cannot get two programs without paying twice the price of the program. The only way in which the attacker may cheat is by not returning a CPU to Z (after getting the replacement), and this action costs R.

THEOREM 4.5. *If $s \in \{(M, X, K, Q) \mid X < COST\}$ is a harmless state for some set of states U_1 defined below reachable from s_0, then it is also harmless for U_2. Where:*

$$U1 = \{(M, X, K, Q) \mid (X < COST) \;\&\; (EuPGM \in M) \;\&\; (Ki = Du \neq null)\}$$

and

$$U2 = \{(M, X, K, Q) \mid ((X < \min(2 \times COST, R)) \;\&\; (E_uPGM, E_wPGM \in M) \;\&\; \exists j \neq i(K_i = D_u \neq null \;\&\; K_j = D_w \neq null)\}.$$

PROOF. If $E_uPGM \in M$, T12 or T13 must have been used. By Theorem 4.4, *PGM* is not in M. If T13 has been used to reach $E_uPGM \in M$ from s, then T14 must have been used before, since it is the only transaction that produces $F_u(i)[PGM, F_u^{-1}(i + 1)]$. But if T14 occurred, it must have been in a history

reachable from s and not before s, since X at s is smaller than $COST$. 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 $E_w PGM$ cannot be produced by T12 or T13, and, since there is no transaction that removes E_u, it remains to show that no two computers can have the same key that is not *null*. In order to get a second key, transaction T17 or T16 must be used. Since $X < R$ in U_2, only T16 can be used, but the application of T16 changes K_w to null. □

If the decrypted code is not obtainable, as shown in Theorem 4.4, and we cannot encrypt the code for another CPU, as shown in Theorem 4.5, there still remains an alternative: to generate several computers with the same keys. In this case the attacker pays only for one copy and actually obtains 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 some other software protection methods, and the solutions available are usually rather unsatisfactory. If we permit replacement of CPUs, an attacker could return a faked CPU (the returned CPU cannot be easily checked since it might be completely impossible to use it).

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 decrypt programs distributed by T12 or T13 (i.e., D_w is a meaningful decryption key if $D_z G(E_w)$ is known, where E_w is the encryption key corresponding to D_w). We next define a set of states U_1 that contains all the states in which there exist two computers with equally meaningful keys, and the attacker did not pay R—that is, the attacker returned the replaced CPU. We show that U_1 is not reachable.

THEOREM 4.6. *Let* $s_0 = (M_0, X_0, K_0, Q_0)$ *be a state such that* M_0 *is the empty set,* $X_0 = 0$ *and all the keys in* $K_0 \cup Q_0$ *are chosen independently. Then* s_0 *is harmless for* $U_1 = \{(M, X, K, Q) \mid \exists j \neq i(K_i = K_j = a \neq null)\ \&\ (D_z G(a^{-1}) \in M)\ \&\ (X < R)\}$.

PROOF. Since $X < R$, then T17 cannot be used to reach U_1. 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_u, the state s_0 is still harmless for U_1. □

The following theorem finds the expenses of the attacker for obtaining n computers with identical keys. We prove that if U_1 is unreachable (as proved by the previous theorem), then for any number $q > 1$ of computers with equally meaningful keys, the set U_q (with q such computers for which the attacker pays less than $q \times R$) is unreachable.

THEOREM 4.7. *If* s *reachable from* s_0 *is harmless for* $U_1 = \{(M, X, K, Q) \mid (X < R)\ \&\ \exists i \neq j(a = K_i = K_j \neq null)\ \&\ (D_z G(a^{-1}) \in M)\}$, *then for any* $q > 1$ *it is harmless to* $U_q = \{(M, X, K, Q) \mid (X < q \times R)\ \&\ (exists\ I\ s.t.\ |I| \geq q)\ \&\ ((i, j \in I)\ implies\ (a = K_i = K_j \neq null)\ \&\ (D_z G(a^{-1}) \in M))\}$.

PROOF. Assume to the contrary that U_q is reachable from s. Since s is harmless for U_1, we must have used T15 at least $q - 1$ times to set new keys into the computers. To use T15 we need $E_u(a, \textit{replace})$. But in U_q we have $D_zG(a^{-1}) \in M$ (i.e., a is the decryption key of some processor); by Lemma 4.3, $a^{-1} = E_ub$; therefore, $a = D_ub^{-1}$. By Lemma 4.3, D_u is not known, and therefore the only way we could get $E_u(D_ub^{-1}, \textit{replace})$ is by T16 or T17 (implying that $b^{-1} = \textit{null}$). Note that if T16 is used, the number of computers with $K_i = a$ does not increase, since the original computer is destroyed at ($K_w = \textit{null}$). Therefore, T17 must have been used. With each execution of T17, the keys of any set of computers with equal keys could be changed, but trivial induction shows that the number of executions of T17 required to change the keys of q computers is at least q, and then $X \geq q \times R$. Thus, no reachable state may be in U_q. □

The last 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 a CPU. We next show that if some public key has not been published by Z (i.e., U_1 is unreachable without integrity violation), then the key is not known even if the CPU is violated (U_2 is unreachable too).

THEOREM 4.8. *In PPS/PKV, if s is harmless for* $U_1 = \{(M, X, K, Q) \mid (D_zG(a) \in M) \;\&\; (X < V)\}$ *then it is harmless for* $U_2 = \{(M, X, K, Q) \mid (D_zG(a) \in M)\}$.

PROOF. There is no transaction, including T18, that performs D_z on a given string. □

5. PPS IMPLEMENTED WITH A 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. The most well-known method is DES [11].

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—These are used to ensure that keys are not invented.
(2) Secrecy—The program is encrypted by the distributor, who cannot decrypt programs encrypted by other distributors.

5.1 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—the identifying key of C_u.

For indirect distribution via a software dealer L, additional key $F_L(i)$ is required in the dealer's computer C_L.

(3) $F_L(i)$—temporal key for indirect software distribution.

The idea is to implement E_x, D_x with conventional keys, and the protocols are given in the following sections. All the computers (not the users) share K_z and may be viewed as cooperative and honest participants. Therefore, information is transferred between them secretly and authenticated by decryption (using K_z or K_u) and by adding redundancy. The redundant information includes strings, like program, key, and so forth, and application of a certification operator G. This operator prevents the creation of strings that would be indistinguishable from meaningful encrypted information. It is equivalent to G discussed in PPS/PK (see Section 3).

Table III contains the corresponding transactions that form a TS denoted by PPS/C.

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

5.2 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, $K_zG(K_u$, *key*), P)—The user sends key K_u hidden by K_z and certified by G.

D2. (P, ($K_zG(K_u$, *key*), *PGM*), C_p)—The producer enters into the computer both users' keys and the program *PGM* to be distributed.

D3. (C_p, $[G^{-1}K_zK_zG(K_u$, *key*)](*PGM*, *prog*), P)—The encrypted program is given to the producer. A string (i.e., *prog*) should be concatenated to *PGM*, to prevent decryption of programs by encrypting them again. This is required only when encryption and decryption are the same. The given key is decrypted by applying K_z and then verified with G^{-1}.

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 encrypted program is loaded into the user's computer.

D6. (C_u, $O(K_uK_u$(*PGM*, *prog*)), U)—The computer executes the program by removing the encryption and deleting the concatenated string (*prog*). While executing, the program is kept inside the CPU with no access to the user.

5.3 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. This protocol resembles the protocol in Section 3.2;

Table III. Transactions of PPS/C

Transaction number	Input	Output	Change	Steps	Meaning
T1	$K_u K_u a$	a	—	—	Decryption
T2	$K_u K_w$	$K_w K_u$	—	—	Commutativity
T3	a, b	$a(b)$	—	—	Encryption
T4	a	$O(a)$	—	—	Execution
T5	$K_u(a, prog)$	$O(a)$	—	D5, I7	Special instruction
T6	—	$K_z(K_u, key)$	—	D1, R1, R2, I2	Special instruction
T7	$K_z(a, key), b$	$a(b, prog)$	—	D3	Special instruction
T8	$K_z(a, key)$	$a(PGM, prog)$	$X = X + COST$	—	Special instruction
T9	$K_z(a, key), F_u(i)[PGM, F_u(i + 1)]$	$a(PGM, prog)$	$Q_u = F_u(i + 1)$	I3, I4	Special instruction
T10	—	$F_u(i)[PGM, F_u(i + 1)]$	$X = X + COST$	I1	Special instruction
T11	$K_u(a, replace)$	—	$K_u = a$	R4	Special instruction
T12	$K_z(K_u, key), K_z(K_w, key)$	$K_u(K_w, replace)$	$K_w = null$	R4, R5	By Z
T13	$K_z(K_u, key), K_z(K_w, key)$	$K_u(K_w, replace)$	$X = X + R$	—	Cheating Z
T14	$(ab)c$	$a(bc)$	—	—	Associativity
T15	$a(bc)$	$(ab)c$	—	—	Associativity

a more efficient version is possible, similar to the improvement suggested to the protocol of Section 3.2 in Appendix B.

I1. $(P, F_L(i)[PGM, F_L(i+1)], L)$—Producer P sells token i to dealer L. This step may be done (for several tokens) before the other steps of the protocol.
I2. $(U, K_zG(K_u, key), L)$—User U's encrypted key is sent to the dealer.
I3. $(L, (K_zG(K_u, key), F_L(i)[PGM, F_L(i+1)]), C_L)$—The dealer uses token i.
I4. $(C_L, K_u(PGM, prog), L)$—The encrypted program is given to the dealer. At the same time, C_L changes from $F_L(i)$ to $F_L(i+1)$.
I5. $(L, K_u(PGM, prog), U)$—The dealer transfers the encrypted program to the user.
I6. $(U, K_u(PGM, prog), C_u)$—The program is entered into the user's computer.
I7. $(C_u, O(K_uK_u(PGM, prog)), U)$—The computer executes the program by decrypting and removing the string *prog*.

5.4 CPU Replacement Protocol (PPS/C)

The following protocol in PPS/C is for the replacement of a user's CPU. The serviceperson S replaces C_u with C_s with the help of Z.

R1. $(U, K_zG(K_u, key), S)$—User U's key is sent to Serviceperson S, is certified by G, and encrypted by K_z.
R2. $(S, (K_zG(K_u, key), K_zG(K_s, key)), Z)$—The serviceperson sends both encrypted keys to Z.
R3. $(Z, K_s(K_u, replace), S)$—Note that in PPS/C, Z does not have to keep track of the keys.
R4. $(S, K_s(K_u, replace), C_s)$—$K_u$ is installed in C_s (replacing K_s).
R5. The CPUs are replaced. The old CPU ought to be returned to Z.

5.5 PPS/C as a TS

The transactions of PPS/C for computers C_u, and C_w are listed in Table III. The variables of PPS/C are: X is the total expense of the attackers, and for every user u, K_u is the key of u's computer C_u. For every dealer L, the temporal key $F_L(i)$ is kept in Q_L. The set M represents all the messages transmitted so far, and corresponds to the information held by the attackers.

Theorems equivalent to Theorems 4.3–4.8 may be proved for PPS/C, but since they are similar to those in Section 4 we will not state them here.

6. PRACTICAL IMPLEMENTATION OF PPS SYSTEMS

6.1 Architecture

The security of PPS relies on keeping the decrypted program only inside the CPU. Therefore modifications are required to the CPU architecture, either by a new design or by integrating an existing CPU with new components.

Two possible architectures are shown in Figure 1. In Architecture A, proposed for the software protection system in [2], instructions are decrypted just before being fetched into the CPU. The decrypted instructions may be kept on a secret queue. The major disadvantage of this method is the overhead due to decrypting before each fetch. To minimize this overhead, the architecture uses a pipeline

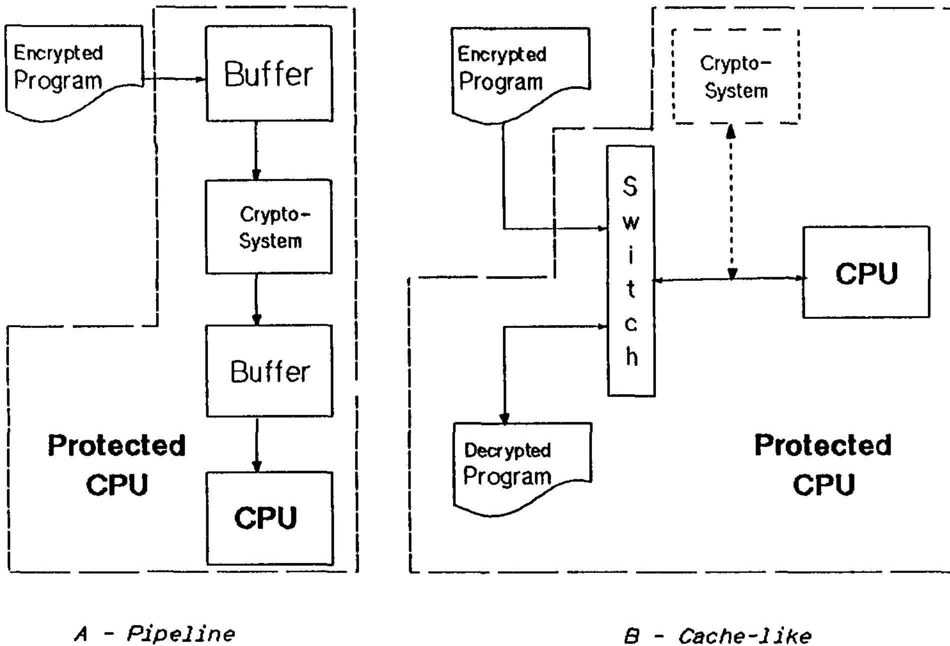

Fig. 1. Architectures for protected CPU.

approach that has instructions decrypted while other instructions are already fetched and executed. Actually, this mechanism may be integrated with the fetch mechanism, which is also implemented (usually) in a pipeline.

In Architecture B, shown in Figure 1, this disadvantage is eliminated. The main idea is to decrypt and load the protected routines into a protected memory that is mapped as a part of the computer memory but is not accessible by any program. This approach resembles a cache, and perhaps may even be integrated with a cache. The internal memory is accessible only for execution or for loading an encrypted program. Part of this memory should be Non Volatile RAM (NV-RAM) to keep the keys and to enable changing them (in the indirect distribution and CPU replacement protocols). Another part should be an ordinary RAM (to hold the decrypted program and for temporary storage). A ROM is needed for holding the PPS algorithm. When instructions are fetched from a block of addresses, they are handed from the internal memory. This is done by having the switch use the address bus and the IF (Instruction Fetch) signal. Thus, it is simple to create a protected CPU by adding components to an existing one. Note that it is not essential to use a special chip for the cryptosystem, since it could be implemented in code (or microcode). Actually, the above architecture can also be used for implementing a secure operating system on a CPU that does not have any hardware protection mechanism.

A prototype system according to Architecture B is being constructed for the IBM-PC (using Intel 8088). The total cost is estimated to be less than $100, and the speed penalty is expected to be the time required to load the protected routines (before execution of the program).

6.2 Cryptographic Operators

Care must be taken when selecting the cryptographic operators to be used. It should be verified that the operators do not have extra properties that are not present as transactions in Tables II and III. The operators may, however, lack any of the properties. If there are properties additional to those listed in the tables, they should be added to the tables and the proofs need to be checked for possible faults.

Note that sometimes properties result from interactions between the operators. For example, G must be chosen so that it would not be feasible to produce $D_z G(x)$ without knowing D_z. Therefore, G must have a high degree of redundancy (say 100 bits). Also, some simple operators, such as concatenation of zeros, may be insufficient at least when using specific cryptosystems (e.g., [8]). However, using an error-detecting code with high degree of redundancy for G would be the obvious choice.

If the cryptographic operators do not have all of the properties assumed in this paper, some simplifications of the protocols may be possible. For example, if the conventional cryptosystem used in PPS/C is asymmetric (i.e., a different procedure is used for encryption and for decryption), the string *prog* concatenated to the program may be eliminated.

6.3 Protocols

The protocols presented here may be modified to meet the goals of specific applications. The security analysis should, however, be repeated afterwards.

Modifications may support additional goals. For example, Appendix A provides a protocol modification for direct distribution in a network. Other goals may be limiting the number of executions of the program, producing a copy that may be run on several computers, charging for encryption, and so forth. Modification may also simplify implementation or improve efficiency. For example, Appendix B provides an alternative protocol for indirect distribution (via a dealer). This protocol is more efficient (but more complex) than the protocol presented in Section 3.2.

Upgrades in the protocols are enabled by adding a protocol to replace the PPS algorithms themselves. To implement this protocol, the PPS algorithm must reside in NV-RAM.

Another possible modification of PPS is transferring a key (*execution key*) instead of the actual program [2, 6]. The program is then transferred and encrypted by that key. The CPU operates the program using the execution key. The security analysis of such a modification, compared to that of PPS (given in Section 4), would not change. As described in the references, this modification might greatly improve the performance of the CPU with PPS and make it reasonably fast compared to a similar CPU without PPS.

7. 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 a

change in the architecture of the CPU. However, this can be done by adding components to an existing CPU.

If implemented by major CPU or computer manufacturers, PPS may also prove effective against compatible equipment manufacturers' ("clones").

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 . . .).

APPENDIX A: Direct Network Distribution

The following changes are needed, assuming users can be identified by their $D_zG(E_u)$. This identification can be made by keeping a public directory or a credit confirmation center. We omit the initial interaction between the user and the computer, in which the user enters the details of the order. The order is sent signed by the same key used in the other protocols. It may be better to use a different key that identifies the user, and then the key-generating body would not hold the signature of the user.

Replace D1, D2 with the following:

D1′. $(U, (D_zG(E_u), E_pD_u(order, NAME, priced, COST)), P)$—The user's key and a signed order, produced by C_u, is sent to ensure that P gets paid. The text of the order is mostly *not* determined by U. Only the *NAME* of the program and the *COST* the user is ready to pay for it is entered. We assume that $D_zG(E_p)$ was given to the user's computer.

D2′. $(P, (NAME', COST', D_zG(E_u), E_pD_u(order, NAME, priced, COST)), C_p)$— The producer's computer is instructed to give program $NAME'$ for price $COST'$. The computer C_p checks that the order is accurate and only then proceeds. The Producer also validates that the user is legitimate.

APPENDIX B: An Alternative Indirect Software Distribution Protocol

There are three disadvantages of the indirect software distribution protocol presented in Section 3.2. First, the producer cannot prove that the dealer ordered the software, and hence billing may be difficult. Second, the program is sent each time from the producer to the distributor. It would be cheaper if the program were sent only once, as a *mastercopy*, and the distributor only buys *tokens* to convert the mastercopy into executable copies. Third, the protocol requires a secret key in the distributor's computer for each producer; this could be difficult to implement. The following protocol is free from all those disadvantages.

For this protocol we assume that each dealer has a counter $CNT(i)$ associated with each order i. The purpose of the CNT counters is to control the number of copies sold. After all the software bought in order i has been sold, the same counter may be used for another order. Identities are validated in the same way as in Appendix A. The program is sent only once per producer–dealer pair. It is sent in a special format referred to as *mastercopy* (actually, even the *mastercopy* is common to all distributers and may be made public). To actually distribute the programs, the dealer must accept a *token* from the producer. Each token enables distribution of several copies; the number of copies is selected by the

dealer when placing the order (steps I1′–I3′). The token specifies the counter to be used, the initial value, and final value. As each copy is produced, the specified counter is incremented until it reaches the final value (i.e., the token has been spent). It is the dealer's responsibility not to use the same counter for two programs. Counters should not be cyclic to prevent the dealer from using the same token twice. In the description below we have omitted the communication between user and computer and the communication between user and dealer before the user places the order (this is similar to steps I1′ and I2′).

I1′. (L, (*order*, N, *copies*, *NAME*, *priced*, *COST*, *counter*, i), C_L)–The dealer uses C_L to produce an order.

I2′. (C_L, (D_L(*order*, N, *copies*, *NAME*, *priced*, *COST*, *counter*, i, *value*, j), $D_zG(E_L)$), L)—After accepting a correct request, C_L uses D_L to produce a signed order in which j is the value of the counter $CNT(i)$.

I3′. (L, (D_L(*order*, N, *copies*, *NAME*, *priced*, *COST*, *counter*, i, *value*, j), $D_zG(E_L)$), P)—The dealer sends a signed order to the producer.

I4′. (P, (*make*, *PGM*, *master*, *NAME*), C_p)—The producer P uses C_p to produce a mastercopy of *PGM*.

I5′. (C_p, (D_p(*master*, *PGM*, *NAME*), $D_zG(E_p)$), P)—After accepting a correct request, C_p uses D_p to produce a signed mastercopy of *PGM*. Steps I4′ and I5′ are done only once per program (the same mastercopy may be used for all dealers). The public key $D_zG(E_p)$ is also kept by P.

I6′. (P, (D_L(*order*, N, *copies*, *NAME*, *priced*, *COST*, *counter*, i, *value*, j), $D_zG(E_L)$), C_p)—The producer transfers the signed order to be delivered (or rejected) by C_p.

I7′. (C_p, E_LD_p(*make*, N, *copies*, *NAME*, *counter*, i, *value*, j), P)—If the order is valid, C_p delivers a token for N copies.

I8′. (P, (E_LD_p(*make*, N, *copies*, *NAME*, *counter*, i, *value*, j), D_p(*master*, *PGM*, *NAME*), $D_zG(E_p)$), L)—The producer transfers the mastercopy, the public key, and the token to the dealer L. The mastercopy and public key need not be sent if another token of the same program has already been accepted by the same dealer.

I9′. (U, ($D_zG(E_u)$, E_LD_u(*order*, *NAME*, *priced*, *COST*)), L)—The user orders the program from L. We have omitted the communication between user and computer to produce this order.

I10′. (L, ($D_zG(E_u)$, E_LD_u(*order*, *NAME*, *priced*, *COST*), E_LD_p(*make*, N, *copies*, *NAME*, *counter*, i, *value*, j), D_p(*master*, *PGM*, *NAME*), $D_zG(E_p)$), C_L)—The user's order, the mastercopy, the token, and the public keys are given to dealer L's computer C_L.

I11′. (C_L, $[G^{-1}E_zD_zG(E_u)](PGM)$, L)—If all is valid (i.e., the public keys are certified by G, the order is valid, the same NAME appears in the token, the order and the mastercopy, and $CNT(i) < j + N$), then C_L produces the copy for U and increments $CNT(i)$.

I12′. (L, $E_u(PGM)$, U)—The user receives the encrypted program.

I13′. (U, $E_u(PGM)$, C_u)—The user loads the program for usage.

I14′. (C_u, $O(D_uE_uPGM)$, U)—Only C_u knows D_u. The computer executes the program and keeps it hidden from the user.

ACKNOWLEDGMENTS

We thank Professor Shimon Even for his helpful comments and suggestions. In particular, he motivated the modified versions of the protocols that appear in the appendixes. We also thank the referees for their helpful comments, especially referee B who has pointed out how transaction T19 may be used to break the original PPS/PK protocol, when RSA is the cryptosystem in use.

REFERENCES

1. ADAPSO. Proposal for software authorization system standards. ADAPSO, 1300 N. 17th St. Arlington, Va., Oct. 1985.
2. ALBERT, D. J. AND MORSE, S. P. Combating software piracy by encryption and key management. *Computer* (Apr. 1984).
3. DANCOTEC COMPUTER. Copybook User Guide. Dancotec, Bakkefaldet 36, 2840 Holt, Denmark, Mar. 1986.
4. DOLEV, D., EVEN, S., AND KARP, R. M. On the security of ping-pong protocols. *Inf. Control 55* (1982), 57–68.
5. DIFFIE, W., AND HELLMAN, M. New directions in cryptography. *IEEE Trans. Inf. Theory IT-22* (1976).
6. HERZBERG, A., AND KARMI, G. On software protection. In *Proceedings of the 4th Jerusalem Conference on Information Technology.* (Jerusalem, Apr. 1984). North-Holland, Amsterdam, 1984.
7. HERZBERG, A., AND PINTER, S. S. The transaction system model and security engineering. To be published.
8. JONGE, W., AND CHAUM, D. Attacks on some RSA signatures. In *Advances in Cryptology—CRYPTO 85* (1985). Springer Verlag, New York, 1985, pp. 18–27.
9. KENT, S. T. Protecting externally supplied software in small computers. Tech. Rep. 255. Massachusetts Institute of Technology/LCS, Cambridge, Mass., Sept. 1980.
10. MERRITT, M. J. Cryptographic protocols. GIT-ICS-83/06. Ph.D. dissertation, The Georgia Institute of Technology, Atlanta, Ga., 1983.
11. NATIONAL BUREAU OF STANDARDS. Data Encryption Standard. FIPS Publication 46, National Bureau of Standards, U.S. Department of Commerce, Washington, D.C. Jan. 1977.
12. RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (Feb. 1978), 120–126.
13. SIMMONS, G. J. How to (selectively) broadcast a secret. In *Proceedings of the 1985 Symposium on Security and Privacy* (Oakland, Calif., Apr. 1985). IEEE, New York, 1985, pp. 108–113.
14. SIMMONS, G. J., PURDY, G. B., AND STUDIER, J. A. A software protection scheme. In *Proceedings of the 1982 Symposium on Security and Privacy* (Oakland, Calif., 1982). IEEE, New York, 1982, pp. 99–103.

Received February 1986; revised February 1987; accepted June 1987