ABYSS: An Architecture for Software Protection

STEVE R. WHITE AND LIAM COMERFORD, MEMBER, IEEE

Abstract—**ABYSS (A Basic Yorktown Security System) is an architecture for protecting the execution of application software. It supports a uniform security service across the range of computing systems. The use of ABYSS discussed in this paper is oriented towards solving the software protection problem, especially in the lower end of the market. Both current and planned software distribution channels are supportable by the architecture, and the system is nearly transparent to legitimate users. A novel use-once authorization mechanism, called a token, is introduced as a solution to the problem of providing authorizations without direct communication. Software vendors may use the system to obtain technical enforcement of virtually any terms and conditions of the sale of their software, including such things as rental software. Software may be transferred between systems, and backed up to guard against loss in case of failure. We discuss the problem of protecting software on these systems, and offer guidelines to its solution.**

Index Terms—**Authorization, copy protection, physical security, software distribution, software license, software protection, tamper resistant.**

I. INTRODUCTION

As computers become a more important source of information and services in our lives, problems of software and data security become increasingly significant. The illicit duplication and use of commercial software is only one example of these problems, but it is increasingly worrisome in the low end of the software market.

Technical methods addressing this problem have included writing the application software so that it looks for an unusual, and supposedly uncopyable, feature on the distribution diskette [1], and the attachment of special hardware devices for each application to be used in the system.

These technical methods have not succeeded because of two complementary shortcomings. First, they are not an effective barrier to duplication. Today's low-end computers are both logically and physically open systems. The user is capable of examining every aspect of the system. Once the behavior of the application is understood, it can be changed to subvert the software protection measures. Second, existing technical methods have imposed unacceptable burdens on the legitimate user. Users are often prevented from making backup copies of their software, and from installing their software on hard disks or file servers.

A practical software protection system must overcome these shortcomings. It must ensure that the effort involved in illicitly duplicating an application is at least as hard as rewriting it from scratch. It must also be extremely convenient for the legitimate user, and flexible enough to support a broad spectrum of computing environments and software distribution systems.

A variety of authors have explored ideas which go beyond the more common diskette-based protection schemes. Kent [2] discusses a variety of secure system architectures. He mentions the valuable idea of tamper-resistant modules, which provide physical security, and uses cryptographic techniques to protect applications from exposure. Best [3]–[7] and Goldreich [8] present a crypto-microprocessor approach, in which application software exists in plaintext only within the instruction decoder of the processor.

Other authors [9]–[15] consider approaches related to the one presented here, but which have limitations. Some limit the ways in which software can be distributed. Some require the use of public key systems. Others do not deal adequately with backup.

II. OVERVIEW OF ABYSS

ABYSS (A Basic Yorktown Security System) is an architecture for protecting the execution of application software, and can be used as a uniform security service across the range of computing systems. This paper is oriented towards a solution to the problem of software protection, especially in the lower end of the market. It addresses both security and ease-of-use concerns. Both current and planned software distribution methods are supportable. Users may back up applications at any time, and install them onto any other system in the event of failure, without the intervention of any other party at that time. A general discussion of ABYSS and software protection can be found in [16] and [17].

The ABYSS architecture provides the software vendor with tools to enforce the conditions under which the application may be used. Software run under ABYSS executes exactly as it was written, and cannot be modified arbitrarily by the user.

The only information which must be kept secret are certain encryption and decryption keys. Aside from these, all of the details of both architecture and implementation may be made public without compromising the integrity of the system.

A. Architecture of ABYSS

The architecture of the system presented here is shown in Fig. 1. Applications are *partitioned* into processes

---

Manuscript received August 1, 1989; revised January 29, 1990. Recommended by T. A. Berson and S. B. Lipner.

The authors are with the IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

IEEE Log Number 9034812.

0098-5589/90/0600-0619$01.00 © 1990 IEEE

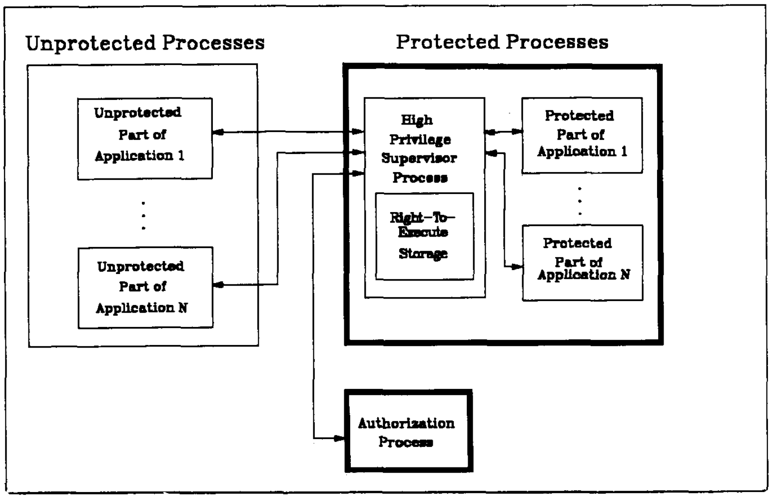

Fig. 1. The architecture of a protected processor system.

which are protected, and processes which are not. Protected application processes are executed within a secure computing environment called a *protected processor*. The conditions under which an application may execute are embodied in a logical object called a *Right-To-Execute*. These conditions are enforced by the protected processor. The movement of Rights-To-Execute into and out of protected processors may require authorization from externally-supplied *authorization processes*.

1) Protected Processors: A protected processor constitutes a minimal, but complete, computing system. It contains a processor, a real-time clock, a random or pseudo-random number generator, and sufficient memory to store protected parts of applications while they execute. It also contains secure memory for storage of Rights-To-Execute. This storage retains its contents even when the system power is off.

The protected processor is a logically, physically, and procedurally secure unit. It is logically secure, in that an application cannot directly access the supervisor process, or the protected part of any other application, to violate their protection. It is physically secure (which is indicated by the heavy box in Fig. 1), in that it is contained in a tamper-resistant package [18]-[20]. It is procedurally secure in that the services which move information, and Rights-To-Execute in particular, into and out of the protected processor cannot be used to subvert the protection.

It is possible for the protected processor to contain the only processors and memory of the entire computing system. Or, the protected processor may be part of a larger computing system, and interact with it through the unprotected processes.

In addition to executing protected application processes, the protected processor executes a *supervisor process*. The supervisor process is responsible for ensuring the logical and procedural security of the protected processor. It executes at a higher privilege level than the application processes, and restricts them to isolated protection domains [21]. This isolation of application processes from each other, and from their unprotected parts, protects an application process from attacks originating in other application processes, or in the unprotected parts of the computing system.

The supervisor process contains a cryptographic facility for managing encryption/decryption keys. This facility decrypts the protected parts of applications as they are loaded into the protected processor. We place the cryptographic transformation between primary memory (such as RAM) and secondary memory (such as a disk). Best [3]-[7] places this transformation between primary memory and the instruction decoder of the processor. Placing it closer to the instruction decoder in the memory hierarchy forces a choice between significant performance degradation of the application, and the use of a cryptosystem which is relatively weak.

Placing the transformation between primary and secondary memory, on the other hand, allows the bandwidth of a relatively strong cryptographic facility to be matched to the data transfer bandwidth, allows efficient pipelining of the data to be decrypted, and allows decrypted instructions to be used numerous times without being decrypted each time. It also allows the efficient use of message authentication or manipulation detection codes on parts of the application.

2) Software Partitioning: For systems in which applications include unprotected processes, it is necessary to partition the application into protected and unprotected parts. The protected part is encrypted when it is outside the protected processor, and only decrypted when it is loaded into the protected processor. The unprotected part is exposed to view.

The protected part cannot be examined or modified by any party external to the protected processor. It is protected by physical security while inside the protected processor, and by cryptographic means while outside. It cannot be modified by rewriting it in a different way, because the partition should be chosen so that the protected part is difficult to reconstruct from knowing only the unprotected part.

The partition is designed so that both parts of the application must be present in order to execute the application. Eliminating accesses to the protected part from the unprotected part should result in a nonfunctional application.

3) Rights-to-Execute: The software is separated from the right to execute it. Only systems which are authorized to use an application have a Right-To-Execute for that application. Rights-To-Execute are created by software vendors, and are used by the supervisor to control the entire range of actions that can be taken with respect to the application.

A Right-To-Execute consists of:

- An encryption and/or decryption key for software packages. This is required to decrypt the application before execution.
- Information about how the Right-To-Execute may be used by supervisor software. For instance, the software vendor may choose not to allow the Right-To-Execute to be transferred to another protected processor once it is installed.
- Information about how the supervisor may permit the Right-To-Execute to be used by software decrypted under its key. The software vendor may wish to allow the application to change the information in the Right-To-Execute, for instance.
- Information about how the supervisor may permit the Right-To-Execute to be used by nonsupervisor software which is not decrypted under its key. For instance, a utility could summarize information about all Rights-To-Execute owned by a user.
- Additional information, at the discretion of software decrypted under the above key. As will be seen later, the application may store such things as an expiration date for its Right-To-Execute, and be assured that the application will not execute after that date.

4) Authorization Processes: Various supervisor services must be authorized to proceed. For instance, the software vendor must authorize the installation of the Right-To-Execute on a protected processor. Authorization processes may be carried out in a number of ways. Brief descriptions of two of these are given here for clarity in subsequent sections.

Smart Cards: Smart cards are cards the size of a credit card, which contain a microprocessor and memory. They can be constructed to perform a subset of the actions of a protected processor which deal with movement and storage of Rights-To-Execute, but not with application execution. Since authorizing supervisor services and storing Rights-To-Execute do not require memory for loadable applications, current smart cards can perform this function. They can then be used as temporary repositories of Rights-To-Execute being transferred between protected processors, and for a number of other useful services.

Tokens: Tokens have the same physical appearance as smart cards, but contain a less expensive chip called a token. The token is useful as a one-time-only authorization of supervisor services.

Both smart cards and tokens must be physically secure, to prevent information contained in them from being revealed. Techniques for chip-level security applicable to smart cards and tokens are discussed in [20].

III. TOKENS: USE-ONCE, FORGERY-RESISTANT AUTHORIZATIONS

We introduce a new authorization mechanism, called a *token process*. The token process is capable of participating in a query-response sequence with a cryptosystem exactly once. Even if the query and response are carried out over insecure channels, the response can still be obtained in such a way that it is extremely improbable that an eavesdropper can forge the behavior of a token process in a subsequent query-response sequence. The token process can be carried out by any simple computing system. It can also be carried out by a small piece of hardware, called a *token*, which is significantly less expensive than hardware capable of providing strong cryptographic services.

A. How Tokens Operate

Tokens fulfill the following criteria.

- The queries, which are generated randomly by protected processors, are sufficiently numerous that it is exteremely improbable that two queries will be the same. Since different queries generate different responses, the response from one query cannot be used as the response to a different query.
- The responses are sufficiently numerous that it is extremely improbable that a random guess at a response will be correct.
- The responses are sufficiently independent of each other, that knowing the response to one query is not significantly helpful in predicting the response to another query.
- The query-response behavior of the token is completely determined by data contained in the token. An encrypted form of these data is delivered to the querying protected processor. This can be done in conjunction with the query-response sequence, or independent of it. Once the protected processor receives the token data and decrypts it, it can predict the correct response to any query.
- The token data is erased from the token as it is read. Thus, a token can only respond to a single query. Subsequent queries reveal no useful information.

Fig. 2 shows a simple conceptual realization of a token. (This is intended to be representational. Real implementations require a small amount of additional support circuitry.) It consists of two shift registers connected to a multiplexor. The registers are shifted left simultaneously in response to a signal on the multiplexor's *query* line. Each time they are shifted, one bit from either the *up* or the *down* register appears on the output line, depending upon the value of the *query* bit. At the same time, nulls

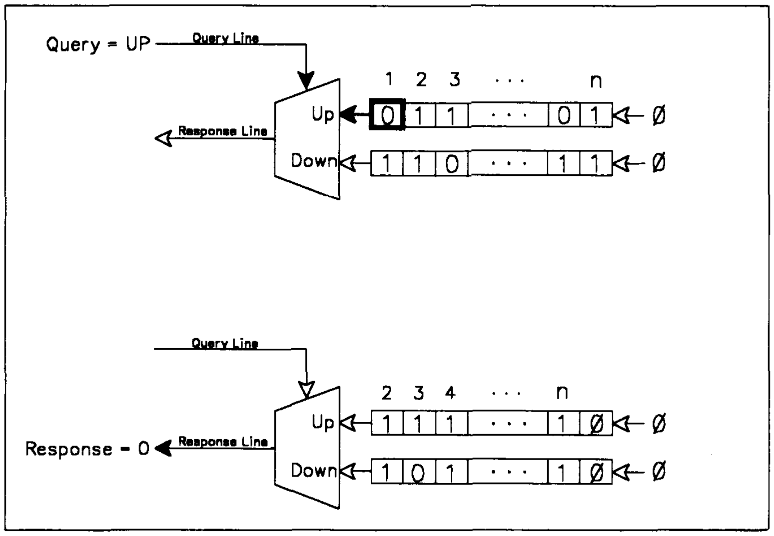

Fig. 2. How tokens work.

are shifted into both registers from the right. This cycle is repeated until the token is completely discharged.

The token is loaded by the software vendor with random binary strings into both the *up* and the *down* registers. These constitute the token data T_j, and should be effectively unique for each token j. (If an attacker possesses two tokens known to have identical token data, the entire token data can be revealed by querying only the *up* register of one token, and only the *down* register of the other.) The software vendor encrypts this data under a key A, called the *application key*, chosen by the software vendor for a particular application, to form $E_A(T_j)$.[1] The plaintext token data is protected by making the token physically secure against tampering.

The token can then act as a one-time-only authorization from the software vendor, to a protected processor which possesses the application key A. (The means by which the protected processor obtains the application key are discussed later.) To do this, the protected processor reads in and decrypts $E_A(T_j)$ to obtain the token data T_j. It then generates a random query Q, which consits of a string of bits as long as either of the token's registers. The query is presented to the token to obtain the token's response R. By construction, all of the token data are lost when it is read, even though only half of the data are revealed by the response.

The protected processor can use its knowledge of the complete token data T_j to simulate the token, and predict the correct response R' to the query Q. By comparing R to R', it can determine whether or not the token is a valid authorization, prepared by a party which knows A. Since all of the token data is discharged when it is read, this can only be done once.

In their ability to prove that they contain certain secret information without revealing a significant fraction of it, tokens resemble the "verify-only memory" of [22]. They differ from zero-knowledge proof protocols [23] in that the response from a token does reveal information about the contents of the token. The single possible response, however, does not reveal a sufficient amount of information to be useful to an attacker, as explained in the next section.

B. Forging a Token

Suppose that an attacker has observed the query and response sequence for a token. What is the probability that, armed with this information, the attacker can respond as that token would have to another query by a protected processor? If successful, this would constitute a successful forgery of a token process, and could produce an illicit authorization.

The query to which the attacker must respond is generated randomly, so it will not have a statistically significant correlation to the observed query. The probability of responding correctly to each bit in the query is the probability that that bit in the query is the same as the one previously observed (in which case the attacker knows the correct response), plus the probability that it is different, times the probability of guessing correctly. For a token with n uncorrelated bits in each shift register,

$$p_{\text{forgery}} = \left[p_{\text{same query}} + (1 - p_{\text{same query}})\, p_{\text{correct guess}}\right]^n. \tag{1}$$

If there are no statistically significant correlations present,

$$p_{\text{same query}} = \tfrac{1}{2}$$

$$p_{\text{correct guess}} = \tfrac{1}{2} \tag{2}$$

---

[1] The expression $E_K(M)$ represents a message M that has been encrypted under a key K. The cryptographic system used, and the mode in which it is used, may depend upon the situation in which they are used.

so,

$$p_{\text{forgery}} = \left(\tfrac{3}{4}\right)^n.$$

A token with shift registers of length $n = 128$ can be implemented on a very small chip, and gives $p_{\text{forgery}} < 1.02 \times 10^{-16}$.

Since it is a protected processor which generates the query to a token, the protected processor can limit the frequency of queries by controlling the amount of time it takes to generate a query. This inhibits a high-speed "guessing" attack on tokens. The average number of guesses required to come up with a single correct response to a query for a given token is

$$N = -\frac{1}{\log_2(1 - p_{\text{forgery}})}. \tag{3}$$

If the time to generate a query is required to be one second, the average time required to forge a response for an $n = 128$ bit token successfully is $t_{\text{average}} > 10^8$ years. Token forgery is covered in more detail in [24].

IV. Supervisor Services

This section describes the supervisor services which were introduced in the previous section. Each of these services can employ either a symmetric cryptosystem or a public key cryptosystem. In this discussion, they are oriented towards a symmetric cryptosystem such as DES, since it is the best match to both the cost and performance requirements of today's lower-end computers. These protocols, with only trivial modifications, can be used with a public key cryptosystem.

Each service could use tokens, or communication between protected processors, as authorization mechanisms. Only one of these authorization mechanisms is illustrated in each case for simplicity. The services are simplified for clarity. An actual implementation would include communication handshaking, verification of the completion of each step of the service, and error recovery.

In order to guarantee the procedural security of the system, each service must ensure four things. It must ensure that no cryptographic keys are exposed in plaintext, that protected software is not exposed in plaintext, that protected software only executes under the terms and conditions chosen by the software vendor, and that the protected software has not been modified.

Clearly, the security of the protected processor requires that its underlying operating system be secure. Operating system security has received a great deal of attention in recent years [25]. While a general discussion of the costs of ensuring the security of any particular implementation of a protected processor is beyond the scope of this architecture paper, a few observations can be made. If the unprotected processor is a single-tasking system, as are many low-end personal computers today, the protected processor need only be single-tasking. It is quite straightforward to create a small, secure, single-tasking system that implements the supervisor services outline below. In a larger multitasking system, it may be possible to use the same (secure) operating system in both the unprotected and protected processor, and implement the supervisor services as privileged applications. This minimizes the incremental effort involved in assuring the security of the protected processor.

A. Creating Protected Software

The ABYSS architecture supports the execution of applications which are not protected, so it can allow existing applications to be migrated to systems with protected processors with no change. To protect an application, the software vendor must create a part of the application to be executed securely, encrypt it, create a corresponding Right-To-Execute, and create an authorization process for installing that Right-To-Execute. The same ABYSS processors which execute protected applications can be used to perform the critical steps in this process, so no special development systems are needed.

1) Writing Partitioned Applications: The application must be written so that at least part of it resides in the protected processor during execution. Depending upon implementation details, this may be the entire application, or it may be a small fraction of it. The section "Secure Software Partitioning" discusses the problem of partitioning an application into protected and unprotected parts.

2) Encrypting the Protected Part of the Application: Once the part P of the application to be protected is complete, it is encrypted under an application key A, chosen by the software vendor, to form $E_A(P)$. The application key may be unique to each application, or even to each copy of each application. This key need not be revealed to anyone else, and the encryption can be done outside of the protected processor, if desired, by the software vendor.

Two goals must be met by this encryption. First, it must avoid the exposure of the plaintext application. Second, it must prevent the protected part of the application from being modified, even randomly. If the application could be modified in a known way, it could be instructed to reveal its entire plaintext content. Even if the application could be modified randomly, it is still possible that the random modifications, executing in the protected processor, will cause the revelation of important information about the protected part. The second goal can be met by using message authentication or manipulation detection codes [26] when encrypting the protected part of the application, and having the protected processor use them to authenticate the application before it is executed.

3) Creating a Right-To-Execute: A plaintext file RTE_A, containing all of the necessary information for the Right-To-Execute for the application, can be created by the software vendor. It contains the application key A, and associated information about how it can be used.

This file is encrypted under a supervisor key S, which is possessed by the protected processor on which the application will be installed, to create $E_S(RTE_A)$. This encryption is done by a protected processor, as a service to the software vendor. It can be done without revealing the supervisor key S, since the encryption can be done securely inside the protected processor. For simplicity at this point, consider the supervisor key S to be common to all protected processors made by a given manufacturer. Alternatives will be discussed later.

4) Creating an Authorization Process: The software vendor must now create an authorization process which will allow the user to install the Right-To-Execute once, but only once. In this section, we will assume that tokens are used as the authorization mechanism.

To prepare tokens as authorizations, the software vendor must generate random token data T_j, which is essentially unique for each token j. The token data are encrypted under the application key to form the encrypted token data $E_A(T_j)$.

5) Shipping the Protected Software: The above components can now be distributed. The unprotected part of the software U, the protected part $E_A(P)$, the Right-To-Execute $E_S(RTE_A)$, and the encrypted token data $E_A(T_j)$ can be distributed by any means. The token data T_j, which is plaintext, must be kept physically secure, for example in a token.

A typical means of distributing these components would be to package U, $E_A(P)$, and $E_S(RTE_A)$ on a floppy diskette. This diskette could be bulk-reproduced, since every such diskette can be identical. Tokens can be designed so that T_j and $E_A(T_j)$ can be placed inside of a token. The diskette and token can be packaged together or separately, and shipped to retailers.

B. Installing a Right-To-Execute

There are several methods by which a user may install Rights-To-Execute on a protected processor. The point of each of these methods is to install the Right-To-Execute for a particular application into the secure, persistent memory of the protected processor, if and only if valid authorization exists to do so. The use of protected processors in preparing and installing Rights-To-Execute creates a trusted path for distributing Rights-To-Execute. Distribution methods based on communication between two protected processors will be described later. A token-based method proceeds as follows.

1) The user makes the token and distribution diskette available to the protected processor.

2) The encrypted Right-To-Execute $E_S(RTE_A)$ is read into the protected processor, and decrypted under the supervior key S. This gives the protected processor access to the application key A.

3) The protected processor must now obtain authorization to install RTE_A as follows.

a) Read the encrypted token data $E_A(T_j)$ from the token and decrypt it under A to obtain T_j. The fact that T_j and the protected part of the application P are both encrypted under the application key A assures the protected processor that they were created by the same party.

b) Generate a random query Q, send it to the token, and obtain a response R from the token.

c) The protected processor can now use its knowledge of what the complete token data T_j is supposed to be, and of its query Q, to simulate the token and calculate the response R' expected from a valid token. There is a valid authorization if and only if $R = R'$. If it is different, either the token was invalid, or an attempt was made by an attacker to forge the token's response.

4) If the authorization was valid, the protected processor installs the Right-To-Execute RTE_A into its secure, persistent memory. If not, RTE_A is purged from the protected processor's working memory and is not installed.

C. Loading and Executing Protected Software

Once a Right-To-Execute is installed on a protected processor, it may be used at any time to enable the execution of applications associated with it. This is done as follows.

1) The protected processor locates the Right-To-Execute and checks that it may be used to execute application programs.

2) U is loaded into the unprotected memory area, and $E_A(P)$ is loaded into the protected memory area and decrypted. If the decryption of $E_A(P)$ yields a valid message authentication or manipulation detection code, execution of U and P is begun. If not, P is purged from protected memory.

3) At any time during execution, including at the very start, the protected part of the application may be allowed to access (and perhaps modify) selected parts of its own Right-To-Execute. It may use this in conjunction with the real-time clock in the protected processor, for instance, to verify that it is not being executed after its expiration date. Should the protected part of the application find that it is being executed contrary to the terms and conditions specified in its Right-To-Execute, it may effectively terminate its own execution.

D. Backing Up Rights-To-Execute

If a protected processor fails, the Rights-To-Execute which it contains can become inaccessible. Since they are necessary in order to execute the protected applications to which they belong, the protected applications could become inoperable unless there is some way to back up Rights-To-Execute.

In this section, we describe a method for doing backup which allows the user to install Rights-To-Execute on another processor at any time, without the intervention of the software vendors or the manufacturer of protected processors. A user whose protected processor fails can be up and running on another system immediately. Furthermore, this second protected processor need not be a previously unused one. Another user's protected processor can accept the backed-up Rights-To-Execute, without interfering with any Rights-To-Execute currently installed on it. This backup method nonetheless preserves the integrity of the system, and cannot be used to create extra copies of Rights-To-Execute.

It proceeds in three steps. In the first, a *backup set* of the Rights-To-Execute in a processor is created. This is similar to creating a backup of a hard disk. If the processor fails, the second step is to install the backup set on a second protected processor. The newly-installed Rights-To-Execute have an inactivation date associated with them. They will become temporarily inactive after that date unless a message is received from the hardware manufacturer, authorizing the removal of the inactivation date. After the inactivation date is removed, the Rights-To-Execute behave just as they had in the original processor.

1) Creating a Backup: There are many ways to perform the backup service. A convenient way is for the user to have a smart card, which is associated with the protected processor to be backed up, and this is the way we will use to illustrate the process. This is not critical, as any other protected processor could be used instead.

The smart card, and its associated processor, each have a supervisor key S_u within them, which is unique to the two of them. Given this supervisor key, a backup set is created as follows.

a) The protected processor and smart card use the unique supervisor key S_u to establish a secure session between each other, which is mediated by a session key B.

b) The collection $C = \{RTE_{A1}, RTE_{A2}, \cdots, RTE_{An}\}$ of Rights-To-Execute to be backed up is encrypted under B to form $E_B(C)$.

c) $E_B(C)$ is written to an external storage medium. If there is sufficient storage on the smart card, it may be kept there. It could also be kept on a diskette. This file may be copied as many times as desired.

d) The smart card stores the key B, and identifies it as the current backup key.

2) Installing a Backup: Each smart card also contains a supervisor key S_c which is common to all protected processors made by a given manufacturer. If the protected processor fails, this key may be used to establish a secure session between the smart card and any other protected processor.

Protected processors also contain common supervisor keys S_m and S'_m, which are used only for communicating with the hardware manufacturer. S_m is used only for sending information to the manufacturer, and S'_m is used only for receiving information from the manufacturer.

Installation of the backup is done as follows.

a) The smart card and the second protected processor use the common supervisor key S_c to establish a secure session between each other, mediated by a session key $\tilde{B}$.

b) The backup key B, and the unique supervisor key S_u, are encrypted under $\tilde{B}$, transmitted to the second protected processor, and decrypted. Remember that S_u is the unique supervisor key of the *first* (failed) processor.

c) The backup key B, and the unique supervisor key S_u, are erased from the smart card. This prevents the smart card from installing the backup set in another protected processor. This also prevents any subsequent backup set from the first processor from being installed.

d) The collection of backed up Rights-To-Execute $E_B(C)$ is read into the second protected processor, decrypted, and installed.

e) The Rights-To-Execute in this collection are made conditional on the existence of a Right-To-Execute associated with S_u, and RTE_{S_u} is given an inactivation date.

f) The second protected processor prepares a message for the hardware manufacturer, $E_{S_m}(S_u)$, and writes it to an external medium. The user sends this medium, along with the first (failed) processor, back to the hardware manufacturer.

3) Retaining Backed-Up Rights-To-Execute: The Rights-To-Execute in the second processor will continue to operate as usual. If no authorizing message were received prior to the inactivation date, the Rights-To-Execute which were part of the backup process would become inactive, but would not be erased. They could not be used, but would remain in the protected processor, awaiting receipt of the message from the hardware manufacturer.

The hardware manufacturer, upon receiving the first (failed) processor and $E_{S_m}(S_u)$, decrypts the message, and verifies that the serial number of the failed processor corresponds to the one associated with the unique supervisor key S_u. This correspondence is checked to ensure that the correct processor has been returned. If a different processor were returned, and the authorization were given to remove the inactivation date from the second processor anyway, the installed Rights-To-Execute would have been permanently duplicated.

Assuming that the correspondence is verified, the inactivation date is removed as follows.

a) The hardware manufacturer encrypts a message $E_{S'_m}(S_u)$, authorizing the removal of RTE_{S_u}, and the elimination of any dependency of other Rights-To-Execute on RTE_{S_u}. $E_{S'_m}(S_u)$ is sent to the user.

b) The second protected processor reads $E_{S'_m}(S_u)$ and decrypts it. Since it was encrypted under S'_m, it is guaranteed to have originated from the hardware manufacturer. The protected processor then erases RTE_{S_u}, and eliminates all dependencies on it. The installed Rights-To-Execute may now be used just as they were on the original processor.

E. Transferring Rights-To-Execute

A Right-To-Execute need not be permanently associated with the protected processor on which it was first installed. Conceptually, it is straightforward to transfer a Right-To-Execute From Alice's protected processor to Bob's by establishing a secure session, transmitting the Right-To-Execute from the Alice's processor to Bob's, installing the Right-To-Execute on Alice's processor, and erasing it from Bob's.

1) A Necessary Connection Between Transfer and Backup: There is a hole in the transfer method outlined above, which permits Rights-To-Execute to be duplicated. The same hole exists in any system which supports both transfer and backup. Rights-To-Execute could be duplicated as follows.

a) A backup of $C = \{RTE_{A1}, RTE_{A2}, \cdots, RTE_{An}\}$ is made from Alice's protected processor.

b) The Rights-To-Execute in C are transferred to Bob's processor. Naturally, they are erased from Alice's processor in the process.

c) The backed-up Rights-To-Execute are installed in Carol's processor. This is unaffected by the fact that Alice's processor (the one backed up) no longer contains these Rights-To-Execute, as a result of transferring them to Bob.

d) Alice's processor is broken, and returned to the manufacturer as failed.

e) The manufacturer verifies the failure, and permits C to be permanently installed in Carol's processor.

At this point, both Bob's and Carol's processors have C installed permanently, and the Rights-To-Execute have been successfully duplicated. While Alice's processor must be sacrificed in the process, this may be a small price to pay if the collection of Rights-To-Execute was very valuable.

2) Transfer with Backup: This hole is plugged by coupling the services of transfer and backup. Each time a transfer is made, all previous backup sets must be invalidated, and a new one created. A revised transfer service, which transfers RTE_{Ak} from Alice's processor to Bob's, is as follows.

a) Alice's protected processor and its associated smart card use their unique supervisor key S_u to establish a secure session with each other, mediated by the session key B'.

b) $C' = \{RTE_{A1}, \cdots, RTE_{Ak-1}, RTE_{Ak+1}, \cdots, RTE_{An}\}$, the revised backup set, is encrypted under B', and stored as described previously. The associated backup key B' replaces the previous backup key B on Alice's smart card. Alice's smart card is now capable of installing only C', the most recent backup set. It is not capable of installing C, or any other previous backup set.

c) Alice's and Bob's protected processors use a common supervisor key S_c to establish a secure session with each other, mediated by the session key T.

d) RTE_{Ak} is encrypted under T by Alice's processor, transmitted to Bob's, and erased from Alice's. Bob's processor decrypts RTE_{Ak} and installs it.

The method of changing the encryption key for data sets, to ensure that they are up to date, is described in a somewhat different context in [2]. In general, this method permits information, such as Rights-To-Execute, to be encrypted, stored on secondary media, and still be guaranteed to be up to date.

V. SECURE SOFTWARE PARTITIONING

Software partitioning is necessary if part of the application must execute from unprotected memory. Cost constraints may require that the protected processor be limited in available processing power and memory. Software partitioning is a means of creating a dependence of the unprotected part of the system on the protected part. This can extend the protection of the protected part to the entire application.

It is difficult to construct a partitioning method that is both demonstrably secure, and convenient enough to use in practical application development efforts. In this section, we outline some ideas that may lead to a better understanding of this problem.

If it were possible to reconstruct the full application without seeing the protected part, the protection of the application would be compromised. It is sufficient if reconstructing the protected part of the software is more difficult than rewriting it from scratch. We define two kinds of complexity in partitioned software: semantic and combinatorial.

A. Semantic Complexity

Semantic complexity reflects the difficulty of reconstructing the protected part by examining the environment of its interaction with the unprotected part. At one end of a spectrum of partitioning methods, selected obscure parts of the application are protected. For instance, an application may contain a proprietary algorithm, all of which could be protected. If that part of the program was difficult to write initially, it may be difficult for an attacker to reconstruct it. At the other end of this spectrum, random parts of the application could execute in the protected processor. For instance, every tenth line of program could be executed by the protected processor. This is semantically complex to the extent that it is difficult to understand a program that has a large number of lines missing.

B. Combinatorial Complexity

Combinatorial complexity reflects the difficulty of exhaustively characterizing the behavior of the protected part by watching what it does. Consider an application in which there are *n access points* in the unprotected part, at which accesses are made to the protected part. At each access point, a k bit argument is passed to the protected part, and the protected part performs some calculation. If this results in $\Omega(2^k)$ independent states of the system,[2] essentially all possible values of the argument must be tried by an attacker to completely characterize the effects of that calculation. This is the case, for instance, in a one-to-one function of the argument, whose value is returned by the calculation. The characterization can be made even more difficult if some or all of the results are stored in the protected part instead.

If no state is stored in the protected part, each access made to it by the unprotected part is independent of every other access. In this case, the combinatorial complexity, as measured by the number of times the attacker must execute the protected part, is $\Omega(n)$ in n.

If state can be stored, each access point can write its argument into some location in protected storage, and re-

---

[2] The expression $\Omega(x)$ arises in complexity theory, and represents a quantity which (asymptotically) is at least as large as x [27].

turn a value read from another location. If the identity of these locations are obscured in the unprotected part, by hashing for instance, it is straightforward to show that $\Omega(n^2)$ executions of the protected part are required for characterization. Ideally, a partitioning method should have a combinatorial complexity which is $\Omega(2^n)$, as well as $\Omega(2^k)$.

VI. ATTACKS ON THE SYSTEM

Attempts at forging a token process were discussed earlier. In this section, we discuss a number of other possible attacks on the system, and their consequences.

A. Plaintext Software: The Asset Being Protected

Software protection is an unusual application of cryptography. Often, messages being encrypted have value for only a small amount of time. The contents of a telephone call made a year ago may have little value, for instance. Furthermore, many cryptographic systems can be constructed so that compromise of a single key, or the compromise of a single cryptographic facility, are of limited value.

These things are not true of practical software protection systems. The asset being protected is the plaintext application itself. If an attacker can obtain the plaintext application, the attacker is in the same position as the original software vendor. The application can be copied, altered, and redistributed at will. It may even be reprotected with the protection system before distribution.

The essential point, which is so often overlooked in software protection discussions, is that, once an attacker obtains the plaintext of the application, the technical protection of that application has been rendered useless.

B. Cryptanalytic Attacks

The cryptographic protection of keys and applications is at least as strong as the protection of the cryptosystem against a nonchosen plaintext attack. Modern cryptosystems, such as DES, are sufficiently resistant to these attacks to permit their use in most software protection situations. If an attacker had a general method of breaking a cryptosystem like DES, there are much more attractive targets to attack than mass-market software!

A cryptanalytic attack on an application key could be attempted, using the response from a token query and the encrypted token data. This attack is more difficult than a nonchosen plaintext attack, since not all of the corresponding plaintext is available in the token response.

C. Physical Attacks

A physical attack on the protected processor itself, if successful, could reveal both supervisor and application keys. But, just as important, it could reveal the plaintext for any application that can be executed on the compromised processor. If the processor is compromised, the attacker can simply purchase every application which is to be compromised. They can each be installed, loaded onto the compromised processor, and their plaintext can be read out in turn.

Since it is necessary for virtually every application to be installable on every processor, the physical security of the protected processor is extremely important. Reference [18] describes a low-cost tamper-resistant package which is useful for protecting information in multichip systems. The physical security of smart cards and tokens is also critical, though losses due to a compromised token can be limited to the ability to install a single application. Chip-level security, which is applicable to smart cards and tokens, is discussed in [20].

D. On Trusting the Hardware Manufacturer

All software protection methods place a great deal of trust in the party that manufactures the protection method. Cryptographic keys must be kept secret, for instance, and must not be used illicitly to reveal information which is encrypted with them.

In [14], the authors maintain that public key cryptosystems are superior to symmetric cryptosystems for software protection, since public key cryptosystems can prevent the hardware manufacturer from knowing the private keys of each protected system. While this is true, it is of limited utility if the hardware manufacturer is dishonest.

The hardware manufacturer could build a "protected" system which is identical to those distributed commercially, but which is not physically secure. Applications could be purchased for that processor as usual, and their plaintext exposed. Although the hardware manufacturer cannot compromise another user's processor per se, the applications themselves can be compromised. If the hardware manufacturer cannot be trusted, significant assets are at risk, regardless of what cryptosystem is used.

ABYSS allows each software vendor to distribute only to protected processors made by manufacturers that they trust. This is done by creating Rights-To-Execute which can be loaded only onto processors which have the trusted manufacturer's supervisor key.

VII. NEW CAPABILITIES

ABYSS opens up a wide variety of new capabilities in computing systems, which were either too expensive, or too inconvenient previously. Two important capabilities are the technical enforcement of the terms and conditions of software sales, and the protection of software distribution channels.

A. Technical Enforcement of Terms and Conditions

A software vendor can specify the conditions under which the Right-To-Execute can be used for an application, or even for a particular copy of an application. This enables each software vendor to enforce a very wide range of conditions on the use of software, and to do so technically. In many cases, only legal enforcement was available previously.

The software vendor may choose not to allow a Right-To-Execute to be transferred to another system, once it is installed. This enforces the condition that execution only occur on a given system.

Any terms and conditions that can be embodied in data in the protected processor, as a part of the Right-To-Execute, can be enforced by the application itself. By storing an expiration time and date, or a maximum number of allowed uses of the application, it is possible to enforce the concept of limited lifetime software.

A single copy of an application can be kept on a file server. The protected processor associated with this server can store a Right-To-Execute which enables up to some fixed number of copies to execute, for instance. When a user requests the application, a single Right-To-Execute is also transferred to the workstation, and the server's count is decremented. The transferred Right-To-Execute may require renewal every minute or so to continue to be valid. The workstation can request renewal of the Right-To-Execute from the server at any time, and thus continue to possess a valid Right-To-Execute as long as is necessary. If the workstation does not check back with the server, the server knows that the workstation's Right-To-Execute has expired, so the server can increment its count of Rights-To-Execute for that application. This *software lending library* is much like a lending library for books, in which the books automatically reappear in the library when they are due.

B. Protection of Distribution Channels

Separating the software from its Right-To-Execute permits a large variety of distribution channels to be used for either. Many electronic distribution methods are not employed today because of the difficulty of preventing illicit interception of the applications. ABYSS can protect all of these distribution channels.

When there can be secure two-way communication between the software vendor and the user, Rights-To-Execute may be distributed on demand to individual workstations. Both software, and Rights-To-Execute can then be distributed on local or wide area networks, or by download from host systems to workstations. By using supervisor keys that are unique to each processor, installation of Rights-To-Execute on only that processor can be ensured.

Compact disks have been suggested as possible distribution media for software. Ironically, one of their disadvantages is that they are capable of storing so many applications on a single disk, that it is difficult to market the entire set of applications together. With ABYSS, it is possible to sell a disk containing hundreds of protected applications at a very low price, and sell individual Rights-To-Execute separately. Similarly, distribution by FM or cable TV broadcast can be protected.

VIII. CONCLUSION

ABYSS is an architecture that enables the protected execution of applications on protected processors, through its use of logical, physical, and procedural security. Software is separated from its Right-To-Execute, and strong cryptographic methods are used to manage both. Secure cryptographic channels can be used to move both software and Rights-To-Execute between protected processors. Tokens are introduced as a new use-once authorization mechanism. They are useful when authorizations are distributed physically, rather than electronically.

ABYSS does not require changes to current or planned software distribution methods. It is nearly transparent to the legitimate user. Software installation is automated, and there is no change in how software is executed. Rights-To-Execute for software can be transferred between systems easily. Files containing the software may be stored on the user's hard disk, on network file servers, or on a mainframe, and may be backed up at will. Furthermore, Rights-To-Execute can themselves be backed up, to preclude loss in the event of a failure of the protected processor.

At the same time, all current and planned software distribution system are supported, and protected. Software authors and vendors may use the system to enforce the terms and conditions of their software sales technically. This opens up many new opportunities for marketing software.

ACKNOWLEDGMENT

The authors greatly appreciate the contributions of the ABYSS group, W. C. Arnold, T. J. Nolan, B. Strohm, and S. H. Weingart, without whom these ideas would not have matured into working prototypes, of A. Chandra, who suggested the privilege structure and helped with early incarnations of this architecture, and of F. N. Parr, who helped simplify and refine several key features of the architecture. We also thank several anonymous reviewers for helpful suggestions.

REFERENCES

[1] J. Voelker and P. Wallich, "How disks are 'padlocked'," *IEEE Spectrum*, p. 32, June 1986.
[2] S. T. Kent, "Protecting externally supplied software in small computers," Ph.D. dissertation, Lab. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, Sept. 1980.
[3] R. M. Best, "Microprocessor for executing enciphered programs," U.S. Patent No. 4 168 396, issued Sept. 18, 1979.
[4] —, "Preventing software piracy with crypto-microprocessors," in *Proc. IEEE Spring COMPCON 80*, San Francisco, CA, Feb. 25-28, 1980, p. 466.
[5] —, "Crypto microprocessor for executing enciphered programs," U.S. Patent No. 4 278 837, issued July 14, 1981.
[6] —, "Cryptographic decoder for computer programs," U.S. Patent No. 4 433 207, issued Feb. 21, 1984.
[7] —, "Crypto microprocessor that executes enciphered programs," U.S. Patent No. 4 465 901, issued Aug. 14, 1984.
[8] O. Goldreich, "Towards a theory of software protection," in *Proc. Crypto '86*, Santa Barbara, CA, 1986, p. 35-1.
[9] G. B. Purdy, G. J. Simmons, and J. A. Studier, "A software protection scheme," in *Proc. 1982 Symp. Security and Privacy*, Oakland, CA, Apr. 26-28, 1982, p. 99.
[10] A. Herzberg, and G. Karmi, "On software protection," in *Proc. Fourth JCIT*, Apr. 1984, p. 388.
[11] M. G. Arnold and Mark D. Winkel, "Computer systems to inhibit unauthorized copying, unauthorized usage, and automated cracking of protected software," U.S. Patent No. 4 558 176, issued Dec. 10, 1985.

[12] D. Everett, "Padlock," *Comput. Bull.*, ser. 3, no. 1, pt. 1, p. 16, Mar. 1985.

[13] G. J. Simmons, "How to (selectively) broadcast a secret," in *Proc. 1985 Symp. Security and Privacy*, Oakland, CA, Apr. 22–24, 1985, p. 108.

[14] A. Herzberg and S. S. Pinter, "Public protection of software," in *Advances in Cryptology: Proc. Crypto 85*, H. C. Williams, Ed., 1986, p. 158.

[15] R. Mori and S. Tashiro, "The concept of a 'software services system (SSS)," *Trans. Inst. Electron. Inf. Commun. Eng. D (Japan)*, vol. J70D, no. 1, p. 79, Jan. 1987.

[16] S. R. White and L. Comerford, "ABYSS: A trusted architecture for software protection," in *Proc. 1987 Symp. Security and Privacy*, Oakland, CA, Apr. 27–29, 1987, p. 38.

[17] V. J. Cina, Jr., S. R. White, and L. Comerford, "ABYSS: A basic Yorktown security system: PC software asset protection concepts," IBM Res. Rep. RC 12401, Dec. 18, 1986.

[18] S. H. Weingart, "Physical security for the μABYSS system," in *Proc. 1987 Symp. Security and Privacy*, Oakland, CA, Apr. 27–29, 1987, p. 52.

[19] D. Chaum, "Design concepts for tamper responding systems," in *Advances in Cryptology: Proc. Crypto 83*, D. Chaum, Ed. New York: Plenum, 1984, pp. 387.

[20] W. L. Price, "Physical security of transaction devices," Nat. Physical Lab., NPL Tech. Memo DITC 4/86, Jan. 1986.

[21] D. E. R. Denning, *Cryptography and Data Security*. Reading, MA: Addison-Wesley, 1983, p. 192.

[22] C. H. Bennet, G. Brassard, S. Breidbart, and S. Wiesner, "Quantum cryptography, or unforgeable subway tokens," in *Advances in Cryptology, Proc. Crypto 82*, Chaum, Rivest, and Sherman, Eds. New York: Plenum, 1983, p. 267.

[23] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," in *Proc. 17th ACM Symp. Theory of Computing*, 1985, p. 291.

[24] B. Strohm, L. Comerford, and S. R. White, "ABYSS tokens," IBM Res. Rep. RC 12402, Dec. 18, 1986.

[25] M. Gasser, *Building A Secure Computer System*. New York: Van Nostrand Reinhold, 1988.

[26] R. R. Jeuneman, S. M. Matyas, and C. H. Meyers, "Message authentication with manipulation detection codes," in *Proc. 1983 Symp. Security and Privacy*, p. 33.

[27] P. W. Purdom, Jr. and C. A. Brown, *The Analysis of Algorithms*. New York: Holt, Rinehart and Winston, 1985.

Steve R. White received the Ph.D. degree in theoretical physics from the University of California, San Diego, in 1982.

He accepted a postdoctoral fellowship at IBM Research, where he later became a Research Staff Member. He has published in the fields of optimization by simulated annealing, software protection, computer security, and computer viruses, and holds several patents in security-related fields. He is currently manager of the Distributed Security Systems group and the High Integrity Computing Laboratory at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY. His research interests include the role of physically secure computing components, and the long-term implications of computer viruses and other self-replicating programs in distributed systems.

Liam Comerford (S'73–A'73) has been a Research Staff Member at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, since 1979. He has participated in research and development activities involving holographic semiconductor device fabrication, laser personalization of integrated circuits, semiconductor optical device design, manufacture, testing, packaging, and system application. He was the coinventor and manager of the ABYSS Project. His recent activities have centered on user interface device technology development.