ABYSS: A Trusted Architecture for Software Protection

Steve R. White
Liam Comerford

IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

ABYSS (A Basic Yorktown Security System) is an architecture for the trusted 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. ABYSS is shown to be a general security base, in which many security applications may execute.

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. It constitutes theft of the intellectual property of the authors, in the same way as copyright and patent infringement, and plagiarism. It can be an economic burden as well, since many small software firms rely on sales of a single software package as their only source of income. By disrupting the software marketplace, theft inhibits the growth of this powerful technology.

Since the inception of small computers, attempts have been made to solve the problem of illicit duplication. Technical methods have included writing the application software so that it looks for an unusual, and supposedly uncopyable, feature on the distribution diskette [Voel86], 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. In fact, most commercial computers of any kind are open, at least to those with operator privileges. The user (or operator) is capable of examining every memory location, and every processor cycle, of the system. Once the behavior of the application is understood, it can be changed to subvert the software protection measures. Similarly, distribution media are completely open to examination and modification. 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.

The trend in the software market today is to abandon technical protection methods, and rely solely upon legal protection. This is difficult at best in the widely distributed and anonymous marketplace often addressed by low end software. It is impossible in countries and cultures which have no history of intellectual property law [Chur86]. So, the need for practical technical protection measures continues, and grows.

A practical software protection system must overcome both of the shortcomings outlined above. It must be extremely secure, and ensure that the effort involved in illicitly duplicating an application is at least as large as that involved in 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, in an attempt to meet the requirements of increased security and convenience.

Kent [Kent80] 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. He does not deal with the full spectrum of software distribution methods in use today, nor with the problem of backing up the applications onto another system should the first one fail.

Best [Best79]-[Best84b] presents a crypto-microprocessor approach, in which application software exists in plaintext only within the instruction decoder of the processor. Best's architecture requires the use of a cryptosystem which is not as secure as the modern cryptosystems often used for communications and transactional security. It limits applications

CH2416-6/87/0000/0038$01.00 © 1987 IEEE

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

to execution on a single system, and does not deal with the issue of backup. It also requires a substantial restructuring of current computing systems. In some versions of this scheme, a processor per application is required, which is costly.

Arnold and Winkel [Arno85] deal with cryptographic security effectively, but require substantial limitations on distribution methods, the processor, and the execution environment.

Purdy, et al. [Purd82], and Simmons [Simm85], present a protocol for distributing decryption keys for encrypted applications. Their distribution method requires software distributors to maintain a cryptographic facility, which may be a burden in current retail store or mail-order distribution. They also do not deal with the issue of backup.

Everett [Ever85] describes a system similar to the one presented here. Since it requires a public key scheme for key distribution, constantly-updated lists of authorized public keys must be maintained by software vendors. Also, software vendors must maintain cryptographic facilities. No provision is made for backup. No scheme is described which would allow software, installed on one system, to be transferred to another system.

Herzberg and Karmi [Herz84], and Herzberg and Pinter [Herz86], present a protection system similar to the one presented here, but they also require software distributors to have a cryptographic facility. They do provide a means of backing up applications in the event of a failure. Their backup method relies on communication with the manufacturer when backed up applications are installed, and on possession of an unused processor on which to install them. This may be problematic in many cases.

Goldreich [Gold86] extends Best's scheme by presenting a method for obscuring the pattern of memory accesses. This increases the protection of Best's scheme, but at the price of additional performance degradation.

Overview of ABYSS

ABYSS (A Basic Yorktown Security System) is an architecture for the trusted 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 [Cina86].

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.

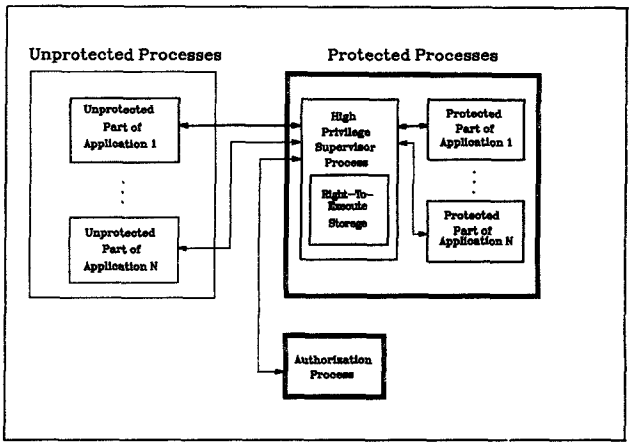

Figure 1. The Architecture of a Protected Processor System

Architecture of ABYSS

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

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 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 Figure 1), in that it is contained in a tamper-resistant package [Wein86] [Chau84] [Pric86]. 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. Such systems consist of a protected processor, and various peripheral devices such as secondary storage. Alternatively, the protected processor may be part of a larger computing system, and interact with it through the unprotected processes.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

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 manages the system's communications resources, to ensure that messages are routed correctly between the protected and unprotected parts of applications. It executes at a higher privilege level than the application processes, and restricts them to isolated protection domains [Denn83] 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 [Best79]-[Best84b] 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 with significantly less security than, say, DES.

Placing the transformation between primary and secondary memory, on the other hand, matches the bandwidth of the cryptographic facility 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.

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 executes securely on the protected processor, in that it cannot be examined or modified by any party external to the protected processor. It cannot be examined externally while it is available for execution because of the logical and physical security of the processor, which inhibit all external access to the plaintext of the protected part. The encrypted form of the protected part cannot be modified directly because of the cryptographic techniques used to detect if it has been tampered with. It cannot be modified by rewriting it in a different way, because the partition is 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 results in a nonfunctional application. This intertwining of the protected and unprotected parts extends the logical protection of the protected part to the entire application.

Rights-To-Execute

The software is separated from the right to execute it. Only authorized users of 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.
- Identifying information about the Right-To-Execute. This can be used to aid the supervisor in searching for the applicable Right-To-Execute.
- Information about how the Right-To-Execute may be used by supervisor software. This allows the software vendor to control what the supervisor may do with the Right-To-Execute. 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 non-supervisor software which is not decrypted under its key. It may be useful to allow other applications to report certain information about the Right-To-Execute. 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.

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. Otherwise, it could be installed on any number of protected processors. Clearly, authorization processes must be difficult to forge, to prevent illicit authorizations. 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.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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. By eliminating the memory in which applications reside during execution, and the real-time clock, current smart cards can contain sufficient cryptographic and storage facilities to authorize supervisor services cryptographically, and to store a limited number of Rights-To-Execute. 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.

Token Cards: Token cards 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 [Pric86].

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 cryptographic services such as DES.

How Tokens Operate

Tokens fulfill the following criteria.

- The queries, which are generated randomly by protected processors, are sufficiently numerous that it is extremely 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. This means that a token can only respond to a single query.

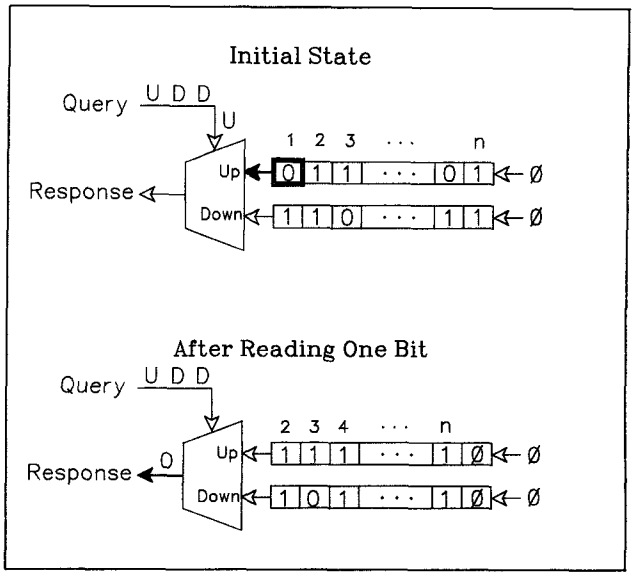

Figure 2. How Tokens Work

Figure 2 shows a simple conceptual realization of tokens. (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 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 by loading 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 $E_A(T_j)$.* 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 consists of a string of bits as long as either of the token's registers. The query is presented to the token to obtain the

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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.

Tokens thus serve as counting devices for the software vendor. A vendor who distributes N tokens is guaranteed that there are only N authorizations in the hands of users.

In their ability to prove that they contain certain secret information without revealing a significant fraction of it, tokens resemble the interactive zero-knowledge proofs of [Gold85]. Tokens are much simpler, though, since they only need to respond to a single query about their information.

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,

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

If there are no statistically significant correlations present,

$$\begin{aligned} p_{same\ query} &= \tfrac{1}{2} \\ p_{correct\ guess} &= \tfrac{1}{2} \\ \text{so,} \quad p_{forgery} &= (\tfrac{3}{4})^n \end{aligned} \quad (2)$$

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

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

$$N = -\frac{1}{\log_2(1 - p_{forgery})} \quad (3)$$

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 $t_{average} > 10^8$ years. Token forgery is covered in more detail in [Stro86].

Secure Sessions Between Protected Processors

Tokens are useful authorization mechanisms when it is impossible or inconvenient to establish a direct communication channel between two protected processors. Where it is possible to establish a such a channel, protected processors possessing an encryption/decryption key in common can set up a cryptographically secure channel between them, mediated by a session key. There are a number of standard ways to do this.

A User's View of the System

The services by which Rights-To-Execute are created, moved, used in executing software, and backed up, are described in detail in the next section. In this section, we illustrate the simplicity of the system to those who use it, by showing a user's view of the system.

For concreteness, we take the case in which applications are purchased through a retail store, as quite a bit of low-end software is today. The applications are intended to be used on a workstation which has an ABYSS architecture, and a hypothetical user interface.

Software Purchase

Retail purchase can be virtually identical to today's practices. The user takes a WonderCalc package off of the shelf, pays for it, and takes it home. No part of the application package need be personalized to the user's system, and the user may remain anonymous. Any application package in the store may be installed on any system, with no prior knowledge of the identity of the system necessary.

Software Installation

The application package contains documentation (hopefully!), a floppy diskette, and a token card. The user inserts the diskette into a drive and the token card into a slot used for just that purpose, and types:

```
install wondercalc
```

The token card authorizes the installation of the right to execute WonderCalc in the user's protected processor. The token is discharged, but the diskette is unchanged. The WonderCalc software may be copied, put onto a hard disk, etc.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.