J. Cryptology (1991) 3: 113–136

Journal of Cryptology

Key Processing with Control Vectors[1]

Stephen M. Matyas
IBM Corporation (400/043), 9500 Godwin Drive,
Manassas, VA 22110, U.S.A.

Abstract. A method is presented for controlling cryptographic key usage based on control vectors. Each cryptographic key has an associated control vector that defines the permitted uses of the key within the cryptographic system. At key generation, the control vector is cryptographically coupled to the key by way of a special encryption process. Each encrypted key and control vector are stored and distributed within the cryptographic system as a single token. Decryption of a key requires respecification of the control vector. As part of the decryption process, the cryptographic hardware verifies that the requested use of the key is authorized by the control vector. This article focuses mainly on the use of control vectors in cryptosystems based on the Data Encryption Algorithm.

Key words. Cryptography, Encryption, Key management, Key distribution, Access control, Hash function.

1. Introduction

Cryptography is a means often used to protect data transmitted through a communications network. Data is encrypted at a sending device using a cryptographic algorithm such as the Data Encryption Algorithm (DEA) [1] and is decrypted at a receiving device. The DEA enciphers a 64-bit block of plaintext into a 64-bit block of ciphertext under the control of a 64-bit cryptographic key. The DEA itself is a nonsecret algorithm whereas the cryptographic keys are kept secret. Each 64-bit key consists of 56 independent key bits and 8 bits that may be used for error detection. In total there are 2^{56} different cryptographic keys that may be used with the DEA.

Since the DEA itself is not a secret algorithm, the degree of protection provided by a DEA-based cryptographic system depends on how well the secrecy of the cryptographic keys is maintained. Therefore, an important goal of sound key management is to ensure that cryptographic keys never occur in clear form outside the cryptographic hardware, except under secure conditions when keys are first initialized within the cryptographic device. For two cryptographic devices to communicate, the devices must share a common cryptographic key. In fact, key-management schemes commonly use many different keys to control access to the

[1] Date received: April 10, 1990. Date revised: October 23, 1990.

data encrypted with those keys. Therefore, the key-management scheme needs an efficient and secure means to distribute keys from one cryptographic device to another. In practice, this is ordinarily accomplished by first installing a common key-encrypting key at each device and thereafter using this key-encrypting key to distribute keys from one device to another electronically. Key distribution encompasses the processes of key generation, key delivery, and key importation. The process of installing the first, or initial, key-encrypting key consists of generating the key at one device and transporting it to the other device (for example, using a courier) where it is initialized within the cryptographic hardware (for example, using manual entry). Thereafter, automated electronic procedures are followed.

To date, cryptographers and implementers of cryptographic standards and products have evolved key-distribution schemes concerned mostly with protocols for the exchange of keys and with strategies for encrypting and authenticating keys and for ensuring the integrity of the key distribution process itself. However, methods for controlling key usage, although not overlooked altogether, have been slow to develop, mainly because until now key-management designs have needed to handle only a few types and uses of keys.

Cryptographic systems being developed today must support an increasing variety of types and uses of keys to meet the growing needs of an expanded and more sophisticated community of cryptographic system users. In fact, it can be said that a fundamental element of electronic key distribution is the means by which key usage information is conveyed, with integrity, from a generating device where keys are created, to one or more receiving devices where keys are used. Without such a capability, it may be possible for an adversary to replace keys of one type with those of another type, thereby causing a receiving device to import and use the keys incorrectly.

To illustrate the danger of importing a key of one type as a key of another type, consider a cryptographic system that supports both data-encrypting keys (type = "data-encrypting key") and key-encrypting keys (type = "key-encrypting key"). As their names imply, key-encrypting keys encrypt keys; data-encrypting keys encrypt data. In like manner, ciphertext decrypted with a key-encrypting key is treated as a key; ciphertext decrypted with a data-encrypting key is treated as data. A decrypted key is made available only to the cryptographic hardware: the key value is kept secret even from the application program that has authorized system use of the key. Decrypted data is made directly available to the application program. Thus, if it were possible to change a key-encrypting key to a data-encrypting key, then clear keys could be recovered outside the cryptographic hardware by treating encrypted keys as encrypted data and executing a decipher data instruction.

In older systems where the number of key types and uses is small, it has been common practice to infer key usage from the context of the key-exchange protocol (for example, that an encrypted data key is transmitted as the third block of 8 bytes in the second message exchanged within the key-distribution protocol). A more general, open-ended approach is needed for present and near-term systems, where the number of key types and uses is certain to be larger. To accomplish this approach, distributed keys should carry with them a record of the key-related information that spells out how and under what conditions these keys can be

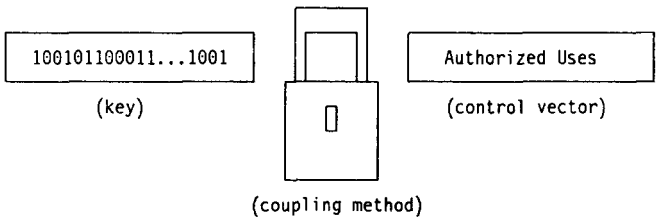


Fig. 1. Control vector concept.

processed by a cryptographic device. This key-related information should be linked cryptographically to the key, making it infeasible for an adversary to cause the cryptographic hardware to process a key without specifying and using the correct key-related information.

This article describes a method for controlling key usage through the use of a data variable called the control vector. The concept underlying the control vector can be applied to key-management designs supporting both symmetric algorithms, such as the DEA where the decryption key is the same as the encryption key, and asymmetric (public-key) algorithms, where the keys are different. However, the discussion focuses mainly on showing how the control vector can be implemented within a key-management scheme based on the DEA.

1.1. *How Control Vectors Work*

In a cryptographic system, each key has an associated control vector, as illustrated in Fig. 1. The key is composed of a randomly generated string of 0 and 1 bits. The control vector is composed of a set of encoded fields representing the authorized or permitted uses of the key. During key generation, the key and control vector are "locked" or cryptographically coupled, to prevent control-vector information from being changed. This process involves encrypting the generated key K with a variant key-encrypting key of the form $KK \oplus C$, where $KK \oplus C$ is produced as the Exclusive-OR product of key-encrypting key KK and control vector C. Upon recovery, the key-encrypting key is again combined with the control vector to produce the same variant key $KK \oplus C$. $KK \oplus C$ is then used to decrypt the encrypted key.

Since KK is a secret key, the process of forming $KK \oplus C$ from KK and C as well as the processes of encrypting and decrypting with $KK \oplus C$ can only be performed within the cryptographic hardware. Thus, the process of cryptographically coupling the key and control vector is one that cannot be duplicated by a user, or would-be adversary. A key *typed* by its creator remains *typed* for the life of the key. The method for cryptographically coupling keys and control vectors is discussed in Section 3.

One of the primary differences between the key and control vector is the manner in which these variables are created. Figure 2 illustrates both production processes. Most DEA keys are produced by the cryptographic hardware using a pseudo-random number generator. On the other hand, control vectors are produced by user-application programs from keywords specified in a control-vector generate

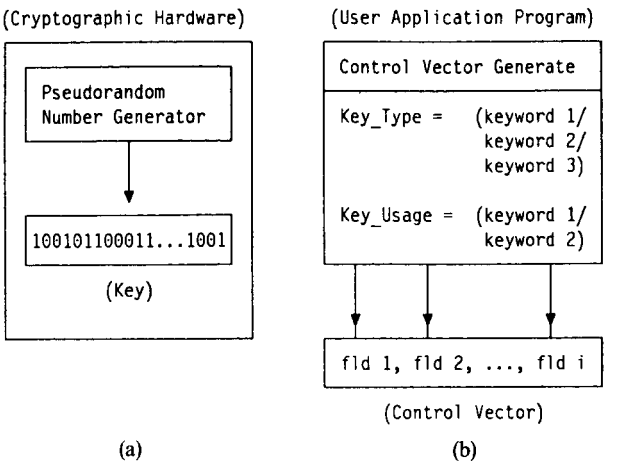


Fig. 2. Generation of (a) keys and (b) control vectors.

function. The process yields a control vector in which the keywords specified in the control-vector generate function are encoded in the control vector as separate fields.

Once created, the key and control vector are used to initialize or customize the cryptographic hardware. Figure 3 illustrates this process. The key customizes the cryptographic algorithm by selecting one of many possible mapping functions. The control vector customizes the hardware cryptographic instruction processor by selecting a set of possible instructions, instruction modes, and instruction processing operations executable by the cryptographic software. That is, it prescribes the authorized uses of a key. Requests for particular uses of a key occur as a result of executing particular cryptographic instructions. Instruction execution is permitted only if the control vector authorizes the requested use.

In effect, the control vector facilitates a form of cryptographic access control similar, in respects, to software access controls implemented in large-scale computers. To explain this relationship better, the topic of access control is briefly discussed.

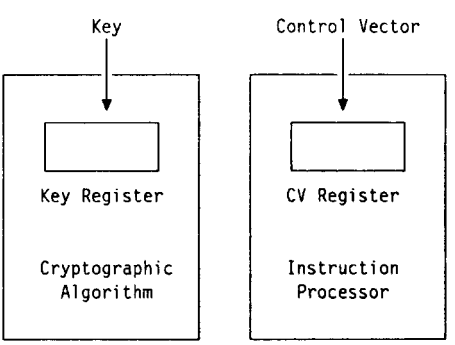


Fig. 3. Customization via the key and control vector.

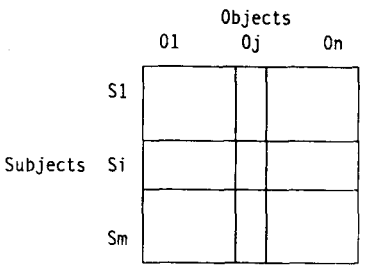


Fig. 4. Access matrix. $A(i, j)$ records the rights of subject Si to object Oj.

1.2. *Access Control*

A well-known model for describing protection systems is the access matrix model developed by Dennis and VanHorn [2], Graham and Dènning [3], Jones [4], Lampson [5], and Harrison *et al.* [6]. In a given situation, the model permits us to determine whether a subject can gain access to a given object.

The protection system is modeled as a set of subjects $\{S1, S2, \ldots, Sm\}$, a set of objects $\{O1, O2, \ldots, On\}$, and an access matrix A. The objects are the protected entities of the system, while the subjects are the active entities that use or make use of these objects. For each subject Si and object Oj, the access matrix records the rights that subject Si has to object Oj, as illustrated in Fig. 4. For example, if Si is an application program and Oj is a data file, then $A(i, j)$ might contain rights r and w to read (r) and write (w) the data file.

In operating systems, objects typically include files, records, processes (that is, executions of system programs), and segments of memory. Subjects are typically users or user-application programs. When a subject requests use of an object (for example, to read a file), the access control mechanism intercepts the request and verifies that the requested use is granted by the access matrix. If verification fails, the requested use of the object is denied.

The access-control mechanism is effective only if subjects are properly identified: one subject should not be able to pose as another subject and acquire the access rights of that other subject. Identification of users is typically performed by way of an authentication procedure during login. In addition, access controls are effective only if the access rights of each subject are protected from unauthorized modification. This is accomplished by specifying the access matrix as an object protected by itself. A special set of rights permit subjects, objects, and rights to be added to and deleted from the access matrix.

Because of their special relationship to the control vector, two other common methods for storing protection information are discussed: access lists and capabilities. An *access list*, for example, $(S1, R1), (S2, R2), \ldots, (Sm, Rm)$, is a list of subjects who are authorized access to some particular object, say Oj. Each entry (Si, Ri) gives the name of a subject, Si, and that subject's rights of access, Ri, for the given object. Relating access lists to the access matrix model, each matrix column is an access list for a particular object. The concept of a capability derives from Iliffe's codewords [7], [8]. A *codeword* is a descriptor specifying the type of an object and either its

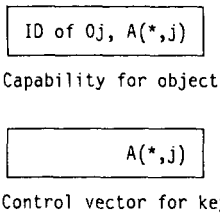


Fig. 5. Capabilities and control vectors for any subject.

value or its length and location. However, it was Dennis and VanHorn who, in 1966, introduced the term "capability" [2]. A *capability* is a pair (O, R) specifying the name of an object, O, and a set of access rights, R, for O. A capability may also specify an object's type. In operation, a capability acts like a ticket. That is, possession of the capability gives the holder R-access rights for object O. Once a capability is issued, no further validation is required.

1.2.1. *Comparison with the Control Vector.* In its simplest form, where names of subjects are not stored in the control vector, the control vector can be likened to a capability. If (Kj, Cj) represents a key Kj and associated control vector Cj, then Cj specifies a set of rights $A(*, j)$ representing how object Kj may be processed in the cryptographic hardware by any subject with access to Kj. The asterisk "*" is a wild card notation specifying any subject. However, there are differences. As illustrated in Fig. 5, a capability contains the name or identifier (ID) of an object, whereas the control vector does not; or, at the very least, need not. First, the control vector is coupled cryptographically to the key, via encryption. Thus, an adversary has no opportunity to substitute one control vector for another or to modify information stored within the control vector. Second, in practice, the control vector is coupled logically to the key by storing both the control vector and the encrypted value of the key together in a system-maintained key data set. Thereafter, the control vector and encrypted key are handled and processed as a single unit.

In principle, the control vector could contain an entire access list, for example, a list of user IDs and the rights of each user to process the given key. A convenient alternative is to make use of a separate control vector for each subject, as illustrated in Fig. 6. This facilitates short, fixed-length control vectors. In that case, the control vector Cj for subject Si contains the subject's identifier, IDi, and the rights $A(i, j)$ representing how Kj may be processed in the cryptographic hardware on behalf of Si.

Storing identifier information in the control vector can be particularly useful where identifiers are device identifiers stored within the cryptographic hardware. In such a case, key processing is permitted only if the device identifier in the control

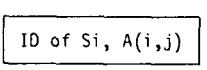


Fig. 6. Control vector for subject Si and key Kj.

vector matches the device identifier in the cryptographic hardware. Smid [9] was the first to show that sender and receiver identifiers could be integrated into the key encryption process to control key distribution effectively—a topic briefly discussed in the next section. However, in situations where access to keys by users and user-application programs must be controlled, which requires that subjects are identified and authenticated before access to the key is granted, other more traditional and often more convenient methods of access control may be used. In most cryptographic systems, access to cryptographic information is based on a strategy of key sharing: a key is first established between each pair of communicators who wish to communicate cryptographically. Thereafter, data or keys encrypted with a key available to one communicator can be decrypted and recovered only with the same key available to the other communicator. In short, possession of a key represents the subject's right to receive or use the information encrypted with that key. The strength of the key-sharing method, of course, depends on how well key secrecy and controlled access to keys can be maintained. For example, if keys are held by users and entered into cryptographic devices only when cryptographic services are required, the degree of protection will be high. This is also the case for keys stored and processed on a "smart" card capable of performing cryptographic functions. (A *smart* card is similar to a magnetic-stripe card used in banking applications to identify users for an Automatic Teller Machine (ATM). The difference between a smart card and a magnetic-stripe card is that the smart card contains a microprocessor and memory, as well as an optional encryption processor, permitting the card to be programmed and, at the programmer's option, to perform cryptographic operations.)

In large computer systems, access to keys can be effectively controlled through the use of software access controls. This is accomplished by defining a set of subjects consisting of user-application programs and a set of objects consisting of key records in a key data set, as illustrated in Fig. 7. Each key record contains one key, and access to the key record is based on a key label. (Examples of software access-control programs are Computer Associates' CA-ACF2™ [10] and International Business Machines' (IBM's) Resource Access Control Facility [11].)

In theory, the list of rights representing how keys may be processed in the cryptographic hardware is limited in kind and number only by the imagination of the architect. In practice, these rights are usually specified in the architecture to

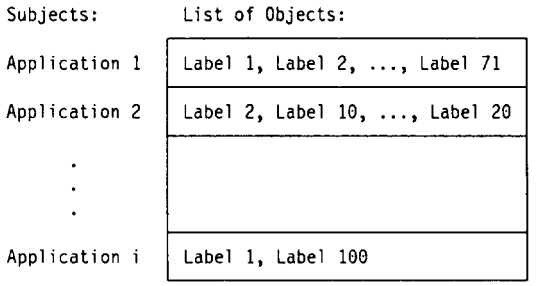


Fig. 7. Application program access to keys via key labels.

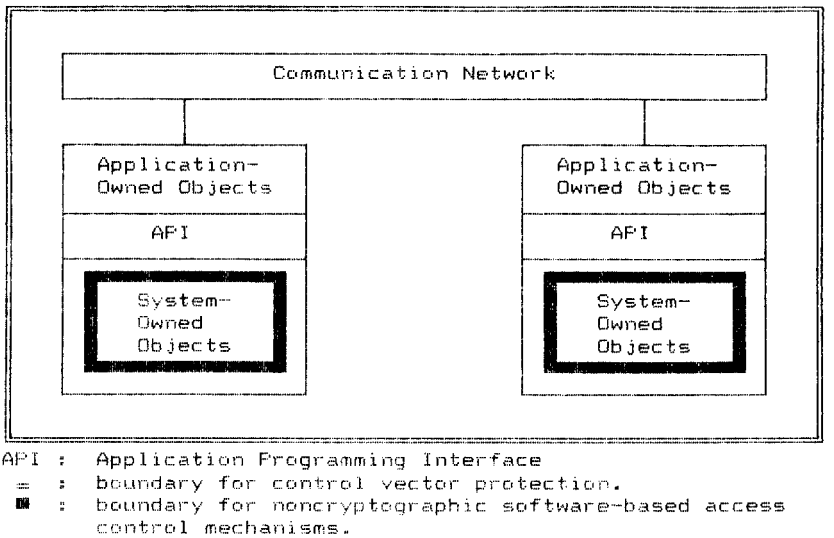


Fig. 8. Domains of protection.

optimize several competing objectives such as need or usefulness, size or storage impact, performance impact, and cost. By using software access controls to control user and user-application program access to keys and then using the control vector to specify the rights for keys to be processed in the cryptographic hardware, it is evident that each of these methods augments the services provided by the other. But is the control vector really needed? Why cannot software access controls provide a total solution? If they cannot, then why not?

1.2.2. *Boundary of Protection.* When we compare domains of protection, it is evident that the control vector provides a viable method for controlling key usage, whereas noncryptographic software-based access-control mechanisms do not. Figure 8 illustrates the domains of protection achieved with each method. In an operating system, a software-based access-control mechanism protects objects only directly under its management and control: the protection mechanism extends to the boundary of the operating system. The access matrix, or comparable structure, must be stored within this protected boundary as well. Otherwise, an insider adversary with write access to the access matrix could modify access-control information, thereby defeating the protection system. In contrast, the control vector protects keys at every network device implementing the control-vector architecture: the protection mechanism extends to the boundary of the network, and even to multiple networks connected by way of one or more interchange nodes. This is so because the key and control vector are coupled cryptographically at the time of key creation, and this coupling remains in force for the life of the key, regardless of where it is transmitted, stored, or processed. Thus, an adversary has no opportunity to substitute one control vector for another or to modify the rights of access granted by a control vector.

The following example further justifies that the domain of interest is larger than a system. Not only does the control vector serve to enforce correct key usage between communicators, for example, two terminal nodes, but the enforcement extends to intermediate nodes as well. During message transmission, control vectors permit the employment of encrypted information at an intermediate node for specific restricted uses without the information being unrestrictedly available or even vulnerable to cryptanalytic attack at the intermediate node.

In summary, the security requirement of the control vector is to provide a networkwide hardware-enforced protection mechanism for controlling key usage. It has been argued that a comparable level of protection is not obtained with software access controls implemented in the operating systems of today's large-scale computers. It has also been argued that the level of protection provided by the control vector is needed in current cryptographic systems to control key processing securely. In the next section methods are discussed in which prior key-management designs have achieved key-usage control. A set of control-vector design criteria is then presented and it is shown that none of the prior methods for controlling key usage completely satisfies the design criteria.

2. Background

In a key-management scheme developed by IBM, outlined in a group of articles in the *IBM Systems Journal* [12]–[14] and implemented in a line of IBM cryptographic products, keys are separated and controlled cryptographically through the use of variants of a master key, called *key variants*. In the key management, a 64-bit master key $KM0$ has two master key variants $KM1$ and $KM2$. (A master key is sometimes designated KM.) In the cryptographic hardware, $KM1$ and $KM2$ are produced from $KM0$ by Exclusive-ORing nonsecret mask values $v1$ and $v2$ with $KM0$, that is, $KM1 = KM0 \oplus v1$ and $KM2 = KM0 \oplus v2$, where "$\oplus$" denotes the Exclusive-OR operation. Keys stored within the cryptographic system are separated into three distinct and cryptographically separate classes. The first class is encrypted with $KM0$, the second class is encrypted with $KM1$, and the third class is encrypted with $KM2$. Each of these classes has a different assigned key use. The IBM key-management scheme has also been extended to handle 128-bit master keys. In that case, the master key variants $KM1$ and $KM2$ are produced from 128-bit key $KM0$ by Exclusive-ORing nonsecret mask values $v1$ and $v2$ with the leftmost and rightmost 64-bit parts of $KM0$, that is, $KM1 = KM0 \oplus (v1, v1)$ and $KM2 = KM0 \oplus (v2, v2)$, where "," denotes concatenation. The 64-bit universal constants $v1$ and $v2$ are defined by the key management architecture.

In a key-management scheme developed by Smid [9] of the National Institute of Standards and Technology—also incorporated in ANSI (American National Standards Institute) Standard X9.17 [15] and ISO (International Organization for Standardization) Standard 8732 [16]—keys are separated and controlled cryptographically through the use of key-manipulation processes called key notarization and key offsetting. Essentially, *key notarization* is a process in which a key-encrypting sender key ($KKij$) or a key-encrypting receiver key ($KKji$) is derived

within the cryptographic hardware from a key-encrypting key (KK) shared between two communicating devices i and j. The keys $KKij$ and $KKji$ are functions of KK and identifiers i and j. Each pair of devices, i and j, also maintains a pair of synchronized incrementing counters: $CTRij$ and $CTRji$. Essentially, *key offsetting* is a process in which a unique time-variant key ($KKij \oplus CTRij$) or ($KKji \oplus CTRji$) is produced within the cryptographic hardware by Exclusive-ORing a key value and a counter value. After a counter has been used, it is incremented by 1. At device i, the variant key $KKij \oplus CTRij$ is used to encrypt keys in the distribution channel sent to device j and $KKji \oplus CTRji$ is used to decrypt keys in the distribution channel received from device j. In contrast to the method of key-usage control in the IBM key-management scheme, where key usage is determined according to the key variant under which the key is encrypted, the ANSI X9.17 key-management scheme links the usage of a key to the method used to derive the key, per the notarization and offset processes. That is, the use of a key is dependent on how the key has been derived.

2.1. *Key Tag*

The method of control vectors is similar in many respects to a method based on *key tags* originally proposed by Jones [17]. (See also [18].) In Jones' method, a 64-bit DEA key consists of 56 independent key bits and an 8-bit key tag. That is, the 8 nonkey bits ordinarily used or reserved for error-detection purposes are used as a key tag. Although not contiguous, the 8 tag bits ($t0, t1, \ldots, t7$) logically constitute a single field. The tag bits are defined as follows: Bit $t0$ indicates whether the key is a data-encrypting key (KD) or a key-encrypting key (KK) (defined as $0 = KD$, $1 = KK$). Bit $t1$ indicates whether the key can be used for encipherment (defined as $0 = no$, $1 = yes$). Bit $t2$ indicates whether the key can be used for decipherment (defined as $0 = no$, $1 = yes$). Bits $t3$–$t7$ are spares. (A similar technique is also used to encode key-usage information within the control vector.)

A function is provided to create keys, which has an input parameter with information necessary to construct a key tag. At key creation, bits $t0$–$t2$ of the tag are encoded to indicate how the key may be processed. For a KK sender key, the bits are encoded as B'110,' indicating that the key is a KK key, that it can be used to encipher KDs, and that it cannot be used to decipher KDs. For a KK receiver key, the bits are encoded as B'101,' indicating that the key is a KK key, that it can be used to decipher KDs, and that it cannot be used to encipher KDs. A KD key can be encoded as either

(1) B'011,' indicating that the key can be used to encipher and decipher data,
(2) B'010,' indicating that the key can be used to encipher but not decipher data, or
(3) B'001,' indicating that the key can be used to decipher but not encipher data.

Thus, the same key "typed" in one case as "encipherment only" and in the other case as "decipherment only" gives a kind of public-key cryptographic system. (A public-key cryptographic system is based on a public-key algorithm, where one key is used for encipherment and another different key is used for decipherment. Smid

[9] has shown that similar public-key properties are obtained with key notarization. Meyer and Matyas [19] have shown that public-key properties can be obtained with key variants.) Furthermore, a *KK* that is typed at one installation as "encipherment only" can be used to encipher keys to be used at another installation. The receiving installation holds a copy of the same *KK*, but it is typed as "decipherment only," which can therefore be used to receive keys from the sending installation.

Once created, a key and tag remain together for the life of the key. A tag appears in clear form only when the key is decrypted and processed within the cryptographic hardware.

2.2. *Need for the Control Vector*

Now that existing methods for achieving key separation and key-usage control have been discussed (that is, key variant, key notarization, and key tag), a set of control-vector design criteria is discussed. The design criteria evolved as part of an effort to develop a cryptographic architecture. It was important, in fact, crucial that the control vector have properties ensuring its suitability as a primary building block of the cryptographic architecture. The design criteria are based, in part, on the set of desirable characteristics of a good architecture put forth by Van de Goor [20].

The control-vector design criteria are listed below:

Extensibility: The control vector should have room for growth. The designer should be aware that users will be inventive beyond his imagination and that needs may change beyond his ability to predict them. A good architecture will have a provision for future developments.

Orthogonality: The control vector should permit key-processing rights to be encoded and processed separately. In a good architecture, conceptually independent functions are kept separate in their specifications.

Generality: The control vector should be useful in a broad range of applications. It should be general purpose.

Continuity: The key-processing rights encoded in the control vector should remain in force for the life of the key.

Simplicity: The control-vector specification and the encryption function for coupling the key and control vector should be simple and straightforward. An architecture that is straightforward in use is often called clean.

The cryptographic architecture satisfies the control-vector design criteria in the following ways. Extensibility is met by a provision to handle control vectors of 64, 128, and greater than 128 bits in length. Orthogonality is met by encoding key-processing rights in the control vector as separate fields. Each cryptographic instruction processes only those fields appropriate for it. Generality is met by externalizing the control vector. This enables the control vector to be processed by user-application programs and cryptographic system programs, in addition to the cryptographic hardware. Continuity is met by ensuring that the control vector remains cryptographically coupled to the key for the life of the key. The cryptographic instructions are designed to thwart manipulation attacks aimed at

modifying control-vector attributes or substituting one control vector for another. Simplicity is met by designing a single, simple set of low-level encryption and decryption functions that cryptographically couple a key and control vector. These low-level functions are used by every cryptographic instruction needing to encrypt or decrypt keys. Key information and key-processing rights are associated with the control vector. That is, the complexity associated with key-manipulation, key-handling, and key-management operations is stored as encoded data in the control vector.

2.3. *The Control Vector Contrasted With Other Methods*

Differences between the control vector and existing methods for controlling key usage primarily are due to differences in their design goals.

The goal of key notarization is to establish a secure key-distribution channel by combining sender and receiver identifier information into the key-encryption process. Key notarization does not address the general problem of controlling key usage.

The goal of the key tag is to illustrate general concepts. It illustrates how key processing can be facilitated with orthogonal key type and key-processing information encoded in the tag. It also illustrates how the IBM key-management scheme, based on three master-key variants, can be implemented using three tag bits. The tag with only eight available tag bits is limited. It has too few bits to handle a modestly complex key-management design and only a limited provision for future growth.

Because key variants are arbitrarily assigned, variant values cannot be checked in the cryptographic hardware using an algorithmic checking procedure. To enforce correct variant usage, a list of key-variant values must be stored within the cryptographic hardware. This may be impractical or uneconomical when the number of key variants becomes only moderately large (for example, 30 or more). To complicate matters, the variant has no provision for encoding key-processing attributes as separate fields. Thus, if n key-processing rights are needed, the theoretical number of variants needed to attain maximum granularity in separation and usage control is on the order of 2^n. Thus, a linear growth in the number of key-processing rights gives rise, in the worst case, to an exponential growth in the number of key variants.

Now that the security and architectural requirements for the control vector have been set forth, a particular control-vector design is discussed.

3. Control Vector

The control vector is a nonsecret cryptographic variable used by a key-management scheme to control cryptographic key usage. In principle, the control vector can be used to control the usage of any cryptographic variable, although for convenience the discussion is limited to keys.

In a cryptographic system, each key, K, has an associated control vector, C, where K and C constitute a logical 2-tuple (K, C). Each cryptographic device is designed

so that key processing can be performed only if the requested use of the key is authorized by the control vector. In effect, C grants processing rights to K. The granularity of control that can be achieved with the control vector, although somewhat dependent on the ingenuity of the designer, depends on the breadth and sophistication of the key-management scheme and the number and kind of processing options available within the cryptographic instruction set. For a limited instruction set, the degree of control exercised by way of the control vector is likely to be very simple; for a comprehensive instruction set supporting a wide range of cryptographic processing options, the degree of control may indeed be highly refined.

3.1. *Cryptographic Coupling of K and C*

Implementation of the control-vector concept requires that the key and control vector (K, C) be coupled cryptographically. Otherwise, the key-usage attributes granted to each key could be changed by merely replacing one control vector with another. Basically, there are two approaches for cryptographically coupling K and C. The first approach is based on integrating C into the functions used to encrypt and decrypt keys. The second approach makes use of a special Authentication Code (AC) calculated directly or indirectly on K and C.

The first approach has the characteristic that K is recovered correctly at a using device only if the correct control vector is specified. Conversely, specification of an incorrect control vector does not prevent the decryption and recovery of a key, but the recovered key K' is for all intents and purposes a spurious value bearing no known relationship to the real key K. It is the task of a good architecture or design to ensure that recovered spurious values of K' are of no cryptographic use to a would-be adversary. The main advantage of the approach is that for a short C, where the length of C is no greater than the length of the key-encrypting key, KK, used to encrypt K, efficient encryption and decryption functions can be devised. The additional processing introduced by the control vector is negligible.

The second approach has the characteristic that both K and C are authenticated before K is processed by the cryptographic device. But some additional processing overhead is needed to calculate AC. For instance, if AC is defined as a 32-bit Message Authentication Code (MAC), per ANSI Standard X9.9 [21], then one DEA encryption step is needed to process each 64 bits of input.

Because the first approach of integrating C into the key-encryption and key-decryption functions has more favorable performance characteristics, the approach is discussed in greater detail in the next section.

3.2. *Control-Vector Encryption/Decryption Algorithms*

The Control-Vector Encryption (CVE) and Control-Vector Decryption (CVD) algorithms used to encrypt and decrypt a key, respectively, are illustrated in Fig. 9. In the CVE algorithm in Fig. 9, C is an input control vector whose length is a multiple of 8 bytes; KK is a 128-bit key-encrypting key consisting of a leftmost 64-bit part KKL and a rightmost 64-bit part KKR, that is, $KK = (KKL, KKR)$; K is a 64-bit key or the leftmost or rightmost 64-bit part of a 128-bit key. The inputs

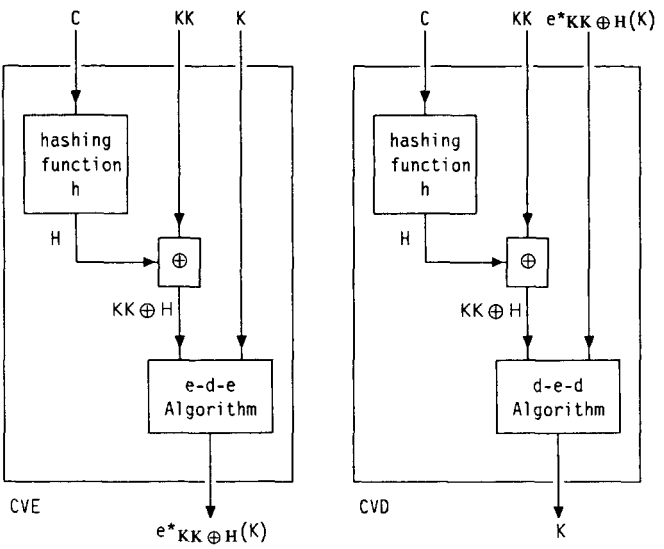


Fig. 9. CVE and CVD algorithms.

are processed as follows. C is operated on by hashing function h to produce the 128-bit output H. H is Exclusive-ORed with KK to produce 128-bit output $KK \oplus H$. Finally, K is encrypted with $KK \oplus H$ to produce output $e^*_{KK \oplus H}(K)$, where e^* indicates encryption with 128-bit key $KK \oplus H$ using an encryption–decryption–encryption (e-d-e) algorithm as defined in ANSI Standard X9.17-1985 [15] and ISO Standard 8732 [16].

An encrypted key of the form $e^*_{KK \oplus H}(K)$ is decrypted with the CVD algorithm as depicted in Fig. 9. The first portion of the CVD algorithm repeats the first portion of the CVE algorithm; that is, C is operated on by hashing function h to produce the 128-bit output H and H is Exclusive-ORed with KK to produce 128-bit output $KK \oplus H$. Then, $e^*_{KK \oplus H}(K)$ is decrypted with $KK \oplus H$ using a decryption–encryption–decryption (d-e-d) algorithm to produce output K. The d-e-d algorithm is just the inverse of the e-d-e encryption algorithm.

Although the CVE and CVD algorithms in Fig. 9 are described using key-encrypting key KK, KK could be replaced by a different key, such as a master key, KM. Since the CVE and CVD algorithms are implemented within the cryptographic hardware, specification of KK is entirely under the control of the key management.

3.3. *Hashing Function h*

The hashing function h implemented in the CVE and CVD algorithms is illustrated in Fig. 10. Hashing function h operates on input control vector C (whose length is a multiple of 64 bits) to produce a 128-bit output H.

If C is 64 bits, $h(C)$ is set equal to (C, C), where "," denotes concatenation, and the extension field (bits 45, 46) in $h(C)$ is set equal to B'00.' That is, h acts like a concatenation function. If C is 128 bits, $h(C)$ is set equal to C and the extension field

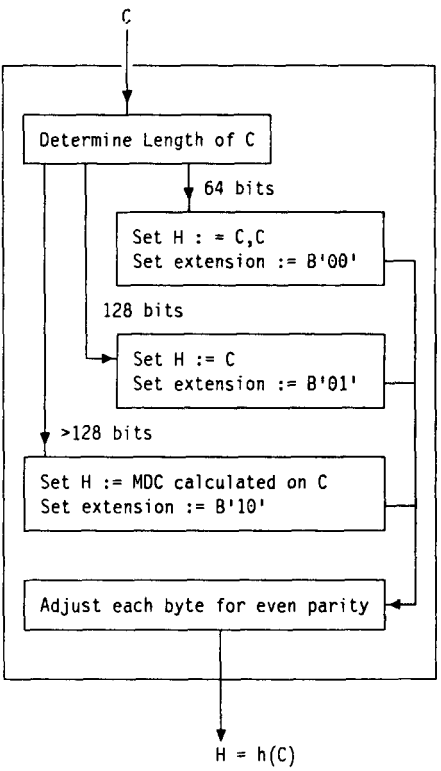


Fig. 10. Hashing function h.

in $h(C)$ is set equal to B'01.' That is, h acts like an identify function. If C is greater than 128 bits, $h(C)$ is set equal to a 128-bit Modification Detection Code (MDC) calculated by the MDC-2 algorithm shown later in Fig. 11 and the extension field in $h(C)$ is set equal to B'10.'

In each of the three cases, the eighth bit of each byte in $h(C)$ is adjusted such that each byte has even parity. This adjustment ensures that when $h(C)$ is Exclusive-ORed with KK, the variant key $KK \oplus h(C)$ has the same parity as KK (that is, if KK has odd parity, then $KK \oplus h(C)$ also has odd parity). Adjusting bits 7, 15, 23, ..., etc. (that is, the parity bits) and setting bits in the extension field in $h(C)$ have the following implications. For 64- and 128-bit control vectors, it means that these control vector bit positions must be reserved for use by hashing function h. For control vectors greater than 128 bits, it means that 110 bits in $h(C)$ are set from the calculated MDC so that $h(C)$ remains a cryptographically strong fingerprint of C.

The extension field in $h(C)$ serves to ensure, for a fixed KK, that the set of keys of the form $KK \oplus h(C)$ consists of three disjoint subsets $S1$, $S2$, and $S3$, where $S1$ denotes the keys resulting from all 64-bit C, $S2$ denotes the keys resulting from all 128-bit C, and $S3$ denotes the keys resulting from all greater than 128-bit C. This prevents a form of cheating wherein the CVD algorithm is tricked into decrypting an encrypted key $e^*_{KK \oplus h(C)}(K)$ by using a false control vector. To illustrate this, ignore

the extension field, and let $C1$ represent a control vector greater than 128 bits and $e^*_{KK\oplus h(C1)}(K)$ an encrypted key produced from KK, K, and $C1$. Instead of presenting $e^*_{KK\oplus h(C1)}(K)$ and $C1$ to the CVD algorithm, $e^*_{KK\oplus h(C1)}(K)$ and $h(C1)$ are presented. That is, we cheat by claiming that $h(C1)$ is a 128-bit control vector. Since, in that case, $h[h(C1)]$ is just equal to $h(C1)$, the CVD algorithm decrypts $e^*_{KK\oplus h(C1)}(K)$ with the key $KK \oplus h(C1)$ to recover K.

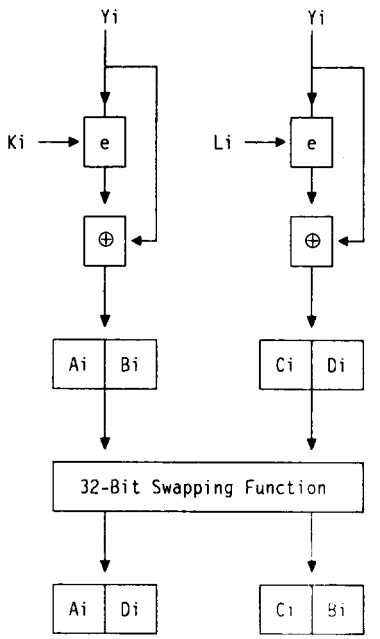
Hashing function h accomplishes two important design objectives. First, it handles both short and long control vectors, thus ensuring that a key-management scheme based on the control vector concept is *open-ended.* Second, the processing overhead to handle short control vectors (64 and 128 bits) is minimized so as to have minimal impact on the key management. A 128-bit control vector is probably more than sufficient to handle the key-usage control requirements of most current key-management systems.

3.4. *Modification Detection Code*

Modification Detection Codes (*MDCs*) and *Message Authentication Codes* (*MACs*) are nonsecret cryptographic variables of fixed, relatively short, length used to authenticate messages or plaintext of arbitrary, much longer, length. However, unlike the MAC which is calculated with a secret key, the MDC is calculated with a public one-way function. Thus, MDCs can be used advantageously in places where it is impractical to share a secret key. More efficient digital-signature procedures can be realized by signing MDCs calculated on messages rather than signing the messages themselves. The process of loading and executing programs within a secure memory can be improved by storing a list of authorized MDCs within the secure boundary of the cryptographic hardware. When a program is loaded, an MDC is calculated on the program and compared for equality against a specified entry in the MDC list. When applied to control vectors, MDCs permit long control vectors to be implemented with a cryptographic algorithm having relatively short fixed-length keys.

A function for calculating 128-bit MDC values, called the MDC-2 algorithm [22], is illustrated in Fig. 11. (MDCs are also discussed by Meyer and Schilling [23].) The MDC-2 algorithm is so-named because two DEA encryptions are performed for each 64-bit block of input plaintext processed by the algorithm. In Fig. 11 $K1$ and $L1$ are two 64-bit nonsecret constant keys. They are used only to process the first 64-bit block of plaintext, $Y1$. Thereafter, input values $K2$, $K3$, ..., Kn are derived from output values ($A1$, $D1$), ($A2$, $D2$), ..., ($An-1$, $Dn-1$), and input values $L2$, $L3$, ..., Ln are derived from output values ($C1$, $B1$), ($C2$, $B2$), ..., ($Cn-1$, $Bn-1$). That is, the outputs of each iteration are fed back, modified slightly, and used as the keys at the next iteration. The 32-bit swapping function merely replaces Bi with Di and Di with Bi.

The MDC-2 algorithm processes data in multiples of 64 bits, with a 128-bit minimum. No padding is performed by the algorithm, although such padding could be performed as a service by either hardware or software. When padding is required, a padding algorithm f should be used that is guaranteed not to produce synonyms. That is, if Y and Y' are two different data inputs, the padded value of Y must not equal the padded value of Y', or mathematically speaking, $Y \neq Y'$ guarantees that




```
Input Y consists of 64-bit blocks Y1,
  Y2, ..., Yn, where n must be > 1.

Output MDC := concatenation of An, Dn, Cn,
  and Bn.

For i = 1, K1 and L1 are defined:
  K1 := X'5252525252525252'
  L1 := X'2525252525252525'

For i = 2,...,n, Ki and Li are defined:
  Ki := concatenation of Ai-1 and Di-1
        except bits 1 and 2 are set equal
        to B'10'.
  Li := concatenation of Ci-1 and Bi-1
        except bits 1 and 2 are set equal
        to B'01'.

  Bits are numbered 0, 1, ..., etc. from
  most significant to least significant.

Yi, Ki, and Li are 64-bit quantities.
Ai, Bi, Ci and Di are 32-bit quantities.
```

Fig. 11. MDC-2 algorithm.

$f(Y) \neq f(Y')$. A padding algorithm satisfying this requirement is given below. The method that requires the input to consist of a whole number of bytes is based on a padding rule described in ANSI X9.23 [24]. (For convenience, the rule is described in terms of bytes not bits.) If the data length is less than 8 bytes, pad bytes are added to make the data length 16. If the data length is 8 or more bytes, pad bytes are added to make the data length a multiple of 8 bytes. Padding is done even if the current data length is a multiple of 8 bytes. All pad bytes except the last pad byte contain a value of X'FF.' The last pad byte is a pad count (in hexadecimal) of the total number of pad bytes, including the pad byte containing the pad count.

To illustrate the problem of synonyms, suppose that the above padding rule is followed, except that padding is not performed when the data length is already a multiple of 8 bytes. Thus, an input Y equal to X'FFFFFFFFFFFFFFFFFFFF FFFFFFFFFF01' is not padded, since its length is already a multiple of 8 bytes. But an input Y' equal to X'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF' is padded with X'01' to produce a value X'FFFFFFFFFFFFFFFFFFFFFFFF FFFFF01' equal to Y. Thus, inputs Y and Y' produce the same MDC.

An MDC-4 algorithm requiring four DEA encryptions per 64-bit block of input has also been designed [22], but its details are not discussed here.

3.5. *Security of the CVE and CVD Algorithms*

The method of encryption and decryption with derived keys of the form $KK \oplus H$ provides an effective means to couple K and C, since given $e^*_{KK \oplus H}(K)$ and C, where $h(C) = H$, there is no apparent computationally efficient means to find alternative

values of $e^*_{KK\oplus H}(K)$ and C', where $h(C') = H'$, that give rise to the same recovered value of K. There is also a precedent for using derived keys for key-management purposes. The IBM and ANSI key-management schemes mentioned in the background section of this article each make use of derived keys produced as the Exclusive-OR product of a secret key and a nonsecret cryptographic variable. In the IBM key-management scheme, the required nonsecret cryptographic variable is formed from a 64-bit variant mask v. In the ANSI key-management scheme, the key-offset process makes use of a nonsecret cryptographic variable formed from a 56-bit counter CTR.

It is noteworthy that the CVE and CVD algorithms are such that the leftmost 64 bits of $KK \oplus H$ may accidently equal the rightmost 64 bits of $KK \oplus H$, even though the leftmost 64 bits of KK do not equal the rightmost 64 bits of KK. However, the probability of such a random event is about equal to $1/2^{56}$ (that is, no better than guessing K). It does not appear that an adversary can gain a practical advantage from such a property, even using a direct-search or trial-and-error method by holding KK constant and varying C to produce different $KK \oplus H$. Methods of exhaustive search do not appear to be improved, nor does it appear that we can detect when the leftmost 64 bits of $KK \oplus H$ equal the rightmost 64 bits of $KK \oplus H$, since K remains encrypted and has no distinguishing feature or property that would signal an adversary that such a state has been reached. To prevent the leftmost 64 bits of $KK \oplus H$ from ever equaling the rightmost 64 bits, the CVE and CVD algorithms could set a bit, say bit i, in the leftmost 64 bits of $KK \oplus H$ to B'0' and could set the same bit i in the rightmost 64 bits to B'1.' In that case, bit i in the 64-bit control vector and bits i and $i + 64$ in the 128-bit control vector would be specified in the architecture as reserved bits (that is, unused for key management). However, the extra computation necessary to avoid this situation does not seem to be justified.

The CVD algorithm is such that a would-be adversary can cause a spurious key, K', to be recovered within the cryptographic hardware. This recovery is done by replacing input $e^*_{KK\oplus H}(K)$ with an arbitrary value, designated X, not equal to $e^*_{KK\oplus H}(K)$, that is, by specifying inputs C, KK, and X to the CVD algorithm instead of inputs C, KK, and $e^*_{KK\oplus H}(K)$. However, a good key-management design will ensure that such spurious keys are of no beneficial use to a would-be adversary. More is said about spurious keys in Section 5.

4. Key Generation and Distribution

To make effective use of the control vector, the key-management scheme must provide a generating function G for the generation of keys, as illustrated in Fig. 12. Generating function G produces outputs $e^*_{key1\oplus H1}(K)$ and $e^*_{key2\oplus H2}(K)$ from an internally generated random key K and from input values $C1$, $C2$, $key1$, and $key2$. $C1$ and $C2$ are control vectors, and $key1$ and $key2$ are 128-bit keys specified by the key management. In an actual implementation, $key1$ and $key2$ might represent master keys of the generating device i, key-encrypting keys shared between the generating device i and designated receiving device j, key-encrypting keys shared between two designated receiving devices j and k, or some combination thereof. The

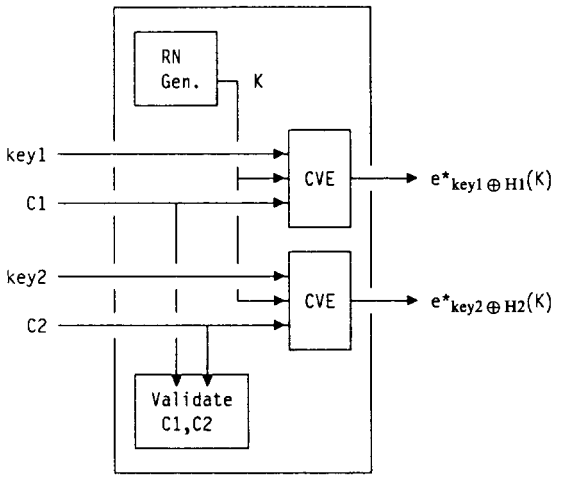


Fig. 12. Generating function G.

values $H1$ and $H2$, in the expressions $e^*_{key1 \oplus H1}(K)$ and $e^*_{key2 \oplus H2}(K)$, are hash values calculated within function G from the input control vectors $C1$ and $C2$, respectively.

The first output $e^*_{key1 \oplus H1}(K)$ is produced by operating on inputs $key1$, K, and $C1$ with encryption algorithm CVE. Likewise, the second output $e^*_{key2 \oplus H2}(K)$ is produced by operating on inputs $key2$, K, and $C2$ with encryption algorithm CVE. Generating function G also validates ($C1$, $C2$) to ensure that both control vectors are consistent with and conform to the architectural specification (that is, $C1$ and $C2$ represent a valid pair permitted by the key management). This validation is called control-vector enforcement or control-vector checking. The outputs $e^*_{key1 \oplus H1}(K)$ and $e^*_{key2 \oplus H2}(K)$ are produced only after ($C1$, $C2$) has been validated; otherwise execution of generating function G is aborted. The valid control vector pairs ($C1$, $C2$) are just those arrived at during the key-management design process.

The key usage attributes in $C1$ and $C2$ might be equal or different. For example, $C1$ could grant K the right to generate MACs, whereas $C2$ could grant K only the right to verify MACs. Thus, one using device can generate MACs, whereas a second using device can only verify MACs.

Generating function G, illustrated in Fig. 12, can be used to distribute keys in a variety of key-distribution environments. In a peer-to-peer environment, key distribution from one device to another, for example, device i to device j, is handled by specifying inputs (KMi, $C1$) and ($KKij$, $C2$) to function G. That is, master key KMi of device i is specified in place of $key1$ and key-encrypting key $KKij$ (installed at devices i and j) is specified in place of $key2$. The encrypted key outputs are therefore $e^*_{KMi \oplus H1}(K)$ and $e^*_{KKij \oplus H2}(K)$, which are stored as key tokens ($e^*_{KMi \oplus H1}(K)$, $C1$) and ($e^*_{KKij \oplus H2}(K)$, $C2$), respectively. Key token ($e^*_{KMi \oplus H1}(K)$, $C1$) is stored at device i and key token ($e^*_{KKij \oplus H2}(K)$, $C2$) is transmitted in a key-distribution channel to device j.

At device j, an import function I is executed to re-encipher $e^*_{KKij \oplus H2}(K)$ to the form $e^*_{KMj \oplus H2}(K)$, as illustrated in Fig. 13, where KMj is the master key of device

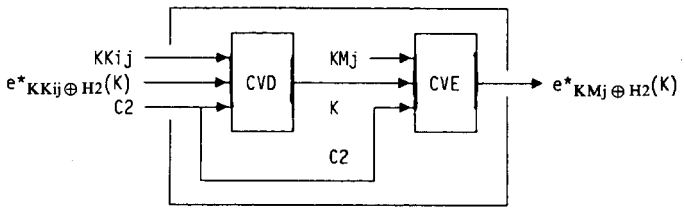


Fig. 13. Key import with import function *I*.

j. Import function *I* consists of two steps:

(1) execution of the CVD algorithm to decrypt $e^*_{KKij \oplus H2}(K)$ with *KKij* and *C*2 to recover *K* and
(2) execution of the CVE algorithm to encrypt *K* with *KMj* and *C*2 to produce $e^*_{KMj \oplus H2}(K)$.

The key token $(e^*_{KMj \oplus H2}(K), C2)$ is stored at device *j*.

The key tokens $(e^*_{KMi \oplus H1}(K), C1)$ and $(e^*_{KMj \oplus H2}(K), C2)$ are now of a form to be processed by the cryptographic hardware at devices *i* and *j*, respectively.

Of course, the processes of key generation and key import are a bit more complicated than represented here, since key-encrypting keys are encrypted under the master key and stored in a key data set. The only key stored in clear form in the cryptographic hardware is the master key. Thus, before *KKij* can be processed by import function *I* or by generating function *G*, it must be decrypted. This extra level of detail is omitted from the present discussion.

The description of key generation and key distribution illustrates several properties of key handling using the control vector. The usage of a key is determined by its creator, where one encrypted copy of the key may have one usage and another encrypted copy of the key may have another usage. During key distribution, keys and control vectors may be translated from encryption with one key to encryption with another key, for example, from *KKij* and *KMj* using import function *I*. But the process is such that keys and control vectors remain linked or coupled together so that we cannot replace the control vector of one key with that of another.

To control key usage effectively, we must link the usage of a key to usage information encoded in the control vector. A method for accomplishing this link is discussed in the following section.

5. Controlling Key Usage

The main features of key-usage control can be conveniently illustrated with a toy, or example, system. Consider a cryptographic system implementing a set of cryptographic instructions *I*1, *I*2, *I*3, and *I*4, where *I*1 and *I*2 each have one encrypted key input and *I*3 and *I*4 each have two encrypted key inputs. For convenience, the six encrypted key inputs are designated *P*1, *P*2, ..., *P*6. The relationship among the instructions and the encrypted key inputs is shown in Fig. 14.

Within the toy system, every generated key can be used or processed in up to six

Instruction	Input Parameter
I1	P1
I2	P2
I3	P3, P4
I4	P5, P6

Fig. 14. Instructions and encrypted key inputs.

ways, that is, as $P1$ in $I1$, as $P2$ in $I2$, as $P3$ or $P4$ in $I3$, and as $P5$ or $P6$ in $I4$. To control key processing adequately, six key-usage fields $U1$–$U6$ are specified within the control vector, as shown in Fig. 15.

Each Uj (for $j = 1, \ldots, 6$) is defined as follows:

$Uj = 1$: the key associated with this control vector can be processed as input parameter Pj.

$Uj = 0$: the key associated with this control vector cannot be processed as input parameter Pj.

Thus, the natural one-to-one correspondence between the key parameters and the key-usage fields designed within the control vector enables the key management conveniently to control *how* a key is used on the basis of *where* the key is used.

As a notational convenience, let $\langle u1, u2, u3, u4, u5, u6 \rangle$ represent the encoding of the usage fields $U1$–$U6$. The remainder of the bits of the toy system in C are spares, and thus are ignored by the cryptographic hardware. For example, the encoding $\langle 100000 \rangle$ permits K to be processed as input key parameter $P1$ in cryptographic instruction $I1$. The encoding $\langle 110000 \rangle$ permits K to be processed either as input key parameter $P1$ in cryptographic instruction $I1$ or as input key parameter $P2$ in instruction $I2$.

When an instruction has two or more execution modes controlled by an input mode parameter, the assignment of input key parameters can be made on the basis of individual instruction modes. Thus, better granularity in key-usage control is achieved.

When encrypted keys and control vectors are specified as inputs to a cryptographic instruction, each control vector is checked to ensure that the requested use of the key is permitted, as illustrated in Fig. 16. That is, control-vector checking ensures that the key usage implied by the specification of a key as a particular input parameter Pj in a particular instruction or instruction mode Ik, is permitted by the control vector. If checking succeeds, the key-recovery process is enabled and processing continues; otherwise instruction processing is aborted. The key-recovery process decrypts the input encrypted keys. Where necessary, the master key, KM, is input to the process, thus permitting keys encrypted under KM to be decrypted

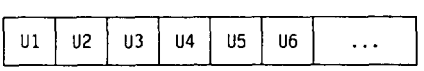


Fig. 15. Control vector layout.

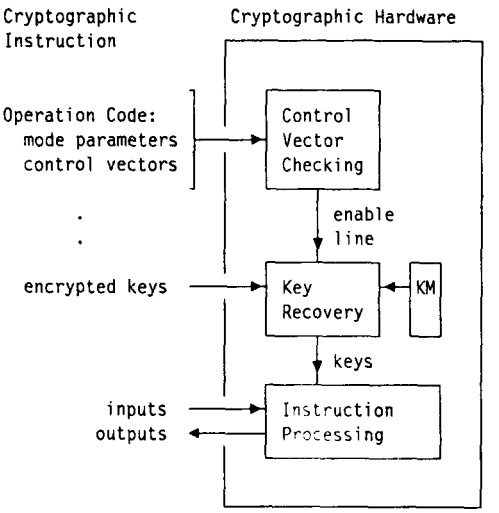


Fig. 16. Control-vector checking process.

using the CVD algorithm previously described in Fig. 9. Thereafter, the decrypted keys as well as additional input information are processed by the cryptographic instruction to produce one or more outputs.

If we cheat by specifying $e^*_{KM \oplus C1}(K)$ and $C2$ instead of $e^*_{KM \oplus C1}(K)$ and $C1$ (that is, a false control vector $C2$ is specified instead of $C1$), one of two things will happen. If control-vector checking fails, the instruction is aborted. If control-vector checking succeeds, the key-recovery process will recover a spurious key $K' \neq K$. As mentioned several times previously, it is the task of the key-management scheme to ensure that such spurious keys are of no beneficial use to a would-be adversary. In practice, it is rather easy to ensure, since cryptographic applications generally involve two communicating parties who must each possess the same cryptographic key. Thus, for practical purposes, one communicating party cannot cheat on the other, since the keys recovered within the cryptographic hardware in that case are K and K' (that is, the first device has K and the second device has K', or vice versa).

Instead of a control-vector specification like the one shown at the beginning of this section, where a single control vector contains the usage attributes for all instructions, there may be multiple control vectors. A more intuitive control-vector specification is achieved if separate control vectors are included in the architecture for each broad category or type of key, such as data keys, key-handling keys, and PIN-handling keys. For example, $I1$ and $I2$ might be data instructions and $I3$ and $I4$ might be key-handling instructions, in which case it may be advantageous to group $I1$ and $I2$ to form a first set called Type 1 and to group $I3$ and $I4$ to form a second set called Type 2, as illustrated in Fig. 17. Control-vector checking is similar except for the additional type field that must be checked.

The reader will appreciate that a full discussion of the principles of control-vector design is beyond the scope of this introductory paper. However, broader and more detailed control-vector designs are possible, and have been developed. A key-

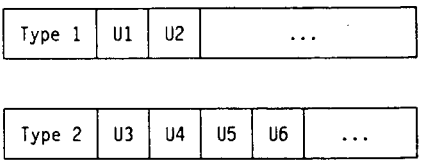


Fig. 17. Type 1 and Type 2 control-vector layouts.

management based on one such control-vector design has been implemented within IBM's new Transaction Security System.

Acknowledgments

The author wishes to acknowledge Dr. C. H. Meyer and Mr. B. Brachtl who collaborated with him on an initial idea for controlling key usage that eventually led to the control vector [25]. The author also wishes to thank Mr. J. Wilkins for his helpful suggestions in the preparation of this article.

References

[1] American National Standard X3.92-1981, *Data Encryption Algorithm*, American National Standards Institute, New York (December 31, 1981).

[2] J. B. Dennis and E. C. VanHorn, Programming semantics for multiprogrammed computations, *Communications of the Association for Computing Machinery*, **9**(3), 143–155 (1966).

[3] G. S. Graham and P. J. Denning, Protection—principles and practice, *Proceedings of the Spring Joint Computer Conference*, Vol. 40, AFIPS Press, Montvale, N.J., 1972, pp. 417–429.

[4] A. K. Jones, Protection in Programmed Systems, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1973.

[5] B. W. Lampson, Protection, *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971, pp. 437–443.

[6] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, Protection in operating systems, *Communications of the Association for Computing Machinery*, **19**(8), 461–471 (1976).

[7] J. K. Iliffe and J. G. Jodeit, A dynamic storage allocation system, *Computer Journal*, **5**, 200–209 (1962).

[8] J. K. Iliffe, *Basic Machine Principles*, Elsevier/MacDonald, New York, 1st edn. 1968, 2nd edn. 1972.

[9] M. E. Smid, *Notarization System for Computer Networks*, NBS Special Publication 500-54, U.S. Department of Commerce, National Bureau of Standards, Washington, D.C., October 1979.

[10] Computer Associates, *CA-ACF2*, Computer Associates International, Incorporated, Garden City, New York, 1988.

[11] IBM Corporation, *Resource Access Control Facility (RACF) General Information Manual* (GC28-0722), IBM Corporation, 1988.

[12] W. F. Ehrsam, S. M. Matyas, C. H. Meyer, and W. L. Tuchman, A cryptographic key management scheme for implementing the Data Encryption Standard, *IBM Systems Journal*, **17**(2), 106–125 (1978).

[13] S. M. Matyas and C. H. Meyer, Generation, distribution and installation of cryptographic keys, *IBM Systems Journal*, **17**(2), 126–137 (1978).

[14] R. E. Lennon, Cryptography architecture for information security, *IBM Systems Journal*, **17**(2), 138–150 (1978).

[15] American National Standard X9.17-1985, *American National Standard for Financial Institution Key Management (Wholesale)*, American Bankers Association, Washington, D.C., 1985.

[16] International Standard ISO 8732, *Banking—Key Management (Wholesale)*, International Organization for Standardization, ISO Central Secretariat, Geneva, 15 November 1988.

[17] R. W. Jones, Some techniques for handling encipherment keys, *ICL Technical Journal*, **3**(2), 175–188 (1982).

[18] D. W. Davies and W. L. Price, *Security for Computer Networks*, 2nd edn., J. Wiley, New York, 1989, pp. 154–157.

[19] C. H. Meyer and S. M. Matyas, *Cryptography—A New Dimension in Computer Data Security*, Wiley, New York, 1982, pp. 421–423.

[20] A. J. Van de Goor, *Computer Architecture & Design*, Addison-Wesley, Reading, Mass., 1989, pp. 3–17.

[21] American National Standard X9.9-1986, *American National Standard for Financial Institution Message Authentication (Wholesale)*, American Bankers Association, Washington, D.C., 1986.

[22] D. Coppersmith, S. Pilpel, C. H. Meyer, S. M. Matyas, M. M. Hyden, J. Oseas, B. Brachtl, and M. Schilling, *Data Authentication Using Modification Detection Codes Based on a Public One Way Encryption Function*, U.S. Patent No. 4,908,861 (March 13, 1990).

[23] C. H. Meyer and M. Schilling, Secure program load with modification detection code, *Proceedings of the Fifth Worldwide Congress on Computer and Communications Security and Protection (SECURICOM 88—SEDEP)*, 8, Rue de la Michodiere, 75002 Paris, pp. 111–130.

[24] American National Standard X9.23-1988, *American National Standard for Financial Institution Encryption of Wholesale Financial Messages*, American Bankers Association, Washington, D.C., 1988.

[25] B. Brachtl, S. M. Matyas, and C. H. Meyer, *Controlled Use of Cryptographic Keys Via Generating Station Established Control Values*, U.S. Patent No. 4,850,017 (July 18, 1989).