Key handling with control vectors

by S. M. Matyas

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 via a special encryption process. Each encrypted key and control vector is 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 also verifies that the requested use of the key is authorized by the control vector. This paper focuses mainly on the use of control vectors in cryptosystems based on the Data Encryption Algorithm.

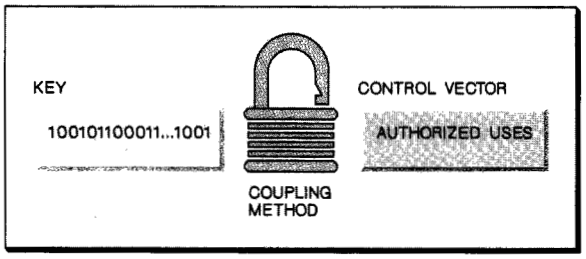
Cryptography is a means often used to protect data transmitted through a communications network. Data are encrypted at a sending device using a cryptographic algorithm such as the *Data Encryption Algorithm* (DEA)[1] and are decrypted at a receiving device. The DEA enciphers a 64-bit block of plaintext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

Since the DEA itself is a nonsecret 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 (unencrypted) 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, a key-management scheme commonly uses many different keys as a means to control access to the data encrypted with those keys. The key-management scheme therefore needs an efficient and secure means to distribute keys from one cryptographic device to another. In practice, this means is ordinarily accomplished by first installing a common key-encrypting key at each device and thereafter using this key-encrypting key to electronically distribute keys from one device to another. 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 the key to the other device (e.g., via courier) where it is initialized within the cryptographic hardware (e.g., via 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

A portion of this paper is reprinted with permission from the *Journal of Cryptology* Vol. 3, No. 2, 1990, published by Springer-Verlag, Inc.

Figure 1 Control vector concept



and with strategies for encrypting and authenticating keys to ensure 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 and thereby cause a receiving device to import and use these keys incorrectly.

To illustrate the danger of importing a key of one type as a key of another type, consider the case where a key-encrypting key (i.e., type = 'key-encrypting key') is imported as a data-encrypting key (i.e., type = 'data-encrypting key'). A key-encrypting key is used by the cryptographic hardware to encrypt and decrypt other keys. Keys encrypted and decrypted with a key-encrypting key are maintained in the secure boundary of the cryptographic hardware: They may be used by authorized application programs, but the values of the keys are kept secret. However, the data encrypted and decrypted with a data-encrypting key are directly available to the application program. Thus, if a key-encrypting key could be changed into a data-decrypting key, the keys encrypted with that key-encrypting key could be decrypted and recovered, in the same way that data are decrypted and recovered, in clear form outside the cryptographic hardware.

Within a cryptographic system, software access control methods can be used to control key usage. The permitted uses of a key specified by a cryptographic application program to the cryptographic system software, i.e., across the application program interface (API), can be checked and enforced within the cryptographic system software. Thus, the software can ensure that a key with an "encipher" attribute but no "decipher" attribute can be used with an encipher instruction but not with a decipher instruction. However, methods to achieve greater protection are possible and may indeed be prudent, or even mandatory, since an inside adversary who bypasses the cryptographic system software and gains access to the cryptographic hardware interface can defeat security by executing cryptographic instructions with keys of one type substituted for those of another type.

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 (e.g., that an encrypted data key is transmitted as the third block of eight bytes in the second message exchanged within the key-distribution protocol). But 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 processed by a using cryptographic device. This key-related information should be linked cryptographically to the key such that it is infeasible for an adversary to cause the cryptographic hardware to process a key except by specifying and using the correct key-related information.

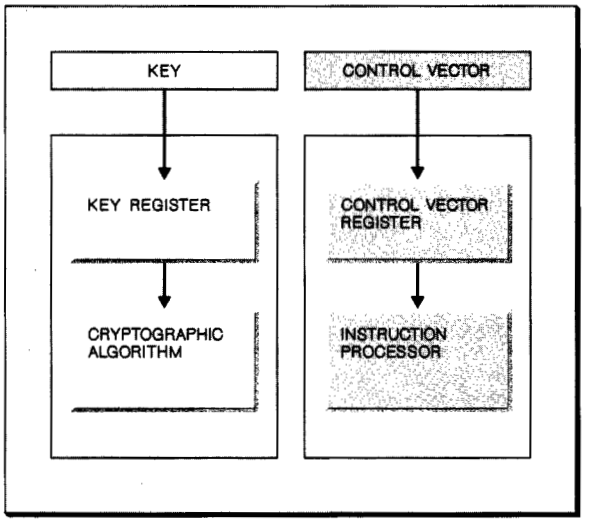
This paper describes a method for controlling key usage through the use of a data variable called the *control vector*.

How control vectors work

Within a cryptographic system, each key has an associated control vector, as illustrated in Figure 1. The key is composed of a randomly generated string of 0 and 1 bits. The control vector is com-

posed of a set of encoded fields representing the authorized or permitted uses of the key. During key generation, the key and control vector are cryptographically "locked," or coupled, to prevent information in the control vector from being changed. This process involves encrypting the generated key K with a variant key-encrypting key KK⊕C, where KK⊕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⊕C, which is then used to decrypt the encrypted key. Since the decryption of the key occurs entirely within the cryptographic hardware, use of a secret key-encrypting key KK yields a process that the user cannot perform independently. The control vector particularizes the process to one "type" of key, while maintaining key secrecy. The method for cryptographically coupling keys and control vectors is discussed in greater detail in the section entitled Control Vector.

Figure 2 Personalization via the key and control vector



The key and control vector are cryptographic variables used to initialize, or personalize, the cryptographic system. Figure 2 illustrates this process. The key personalizes the cryptographic algorithm by selecting one of many possible mapping functions. The control vector personalizes the hardware cryptographic instruction processor by selecting a set of possible instructions, instruction modes, and instruction processing operations that may be executed by the cryptographic software.

The concept underlying the control vector can be applied to key-management designs supporting both symmetric algorithms such as the DEA in which the decryption key is the same as the encryption key, and asymmetric ("public-key") algorithms in which 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. The first part of the paper, up to the section entitled Control Vector Forms and Formats, discusses general control vector concepts. The second part discusses a control vector design implemented in the IBM Transaction Security System.

Background

The ways in which prior key-management designs have achieved key-usage control can be traced. In a key-management scheme developed by IBM, outlined in a group of papers previously published in the *IBM Systems Journal*[2-4] 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. In the cryptographic hardware, KM1 and KM2 are produced from KM0 by exclusive-ORing nonsecret mask values v1 and v2 with KM0, i.e., KM1 = KM0 ⊕ v1 and KM2 = KM0 ⊕ v2, where ⊕ denotes the exclusive-OR operation. Keys stored within the cryptographic system are separated into three distinct and cryptographically separate classes, where 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 usage. (The notation KM is sometimes used in place of KM0.) 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, i.e., KM1 = KM0 ⊕ (v1,v1) and KM2 = KM0 ⊕ (v2,v2), where the comma denotes concatenation. The values v1 and v2 are 64-bit universal constants defined by the key-management architecture.

In a key-management scheme from Smid[5] of the National Institute of Standards and Technology (NIST)—also incorporated in ANSI (American National Standards Institute) Standard X9.17[6] and ISO (International Organization for Standardization) Standard 8732[7]—keys are separated and controlled cryptographically through the use of key-manipulation processes called *key notarization* and *key offset*. 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 ⊕ CTRij) or (KKji ⊕ 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 one. At device i, the variant key KKij ⊕ CTRij is used to encrypt keys in the distribution channel sent to device j, and KKji ⊕ 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 the key depends on how the key has been derived.

Key tag. The method of control vectors is similar in many respects to a method based on *key tags* originally proposed by Jones.[8] (See also Davies.[9]) In Jones' method, a 64-bit DEA key consists of 56 independent key bits and an 8-bit key tag. That is, the eight nonkey bits ordinarily used or reserved for error-detection purposes are used as a key tag. Although not contiguous, the eight tag bits (t0, t1, ..., 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) (0 = KD, 1 = KK). Bit t1 indicates whether the key can be used for encipherment (0 = no, 1 = yes). Bit t2 indicates whether the key can be used for decipherment (0 = no, 1 = yes). Bits t3 through t7 are spares. (A similar technique is also used to encode key-usage information within the control vector.)

Keys are created by a function that has an input parameter with information necessary to construct a key tag. At key creation, bits t0 through t2 of the tag are encoded as follows. 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 (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.) Furthermore, a KK "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 "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.

The key tag differs from the control vector in several respects. First, the control vector can be implemented without affecting key parity. This independence permits the control vector to be used with existing or "off the shelf" cryptographic hardware that accepts keys only if they have correct key parity. Second, practical implementations of the control vector are possible where control vector length is unbounded. This condition permits the control vector specification to be extended, as necessary, to satisfy new requirements placed on the key management. Third, the control vector is a data variable stored and transmitted with the key in clear form. Hence, options are available for the control vector to be processed

advantageously at different levels and points within the cryptographic system. For example, the control vector can be partitioned so that key usage can be controlled (1) in the cryptographic hardware, (2) in the cryptographic system software, and (3) in the cryptographic application program, using three subvectors, which have been specified separately in an architecture. Basically, the measure of integrity obtained with such a mechanism is as good as the measure of integrity that one has over the process of ensuring that the control vector is not changed from the point in time when it is checked, either by the application or by the software, to the point in time when it is read into the cryptographic hardware where it is processed. Strategies and methodologies for implementing such a hierarchy of key-usage control are beyond the scope of this paper and are not further discussed.

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 via 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.

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. A first approach is based on integrating C into the functions used to encrypt and decrypt keys. A second approach makes use of a special authentication code (AC) calculated directly or indirectly on K and C.

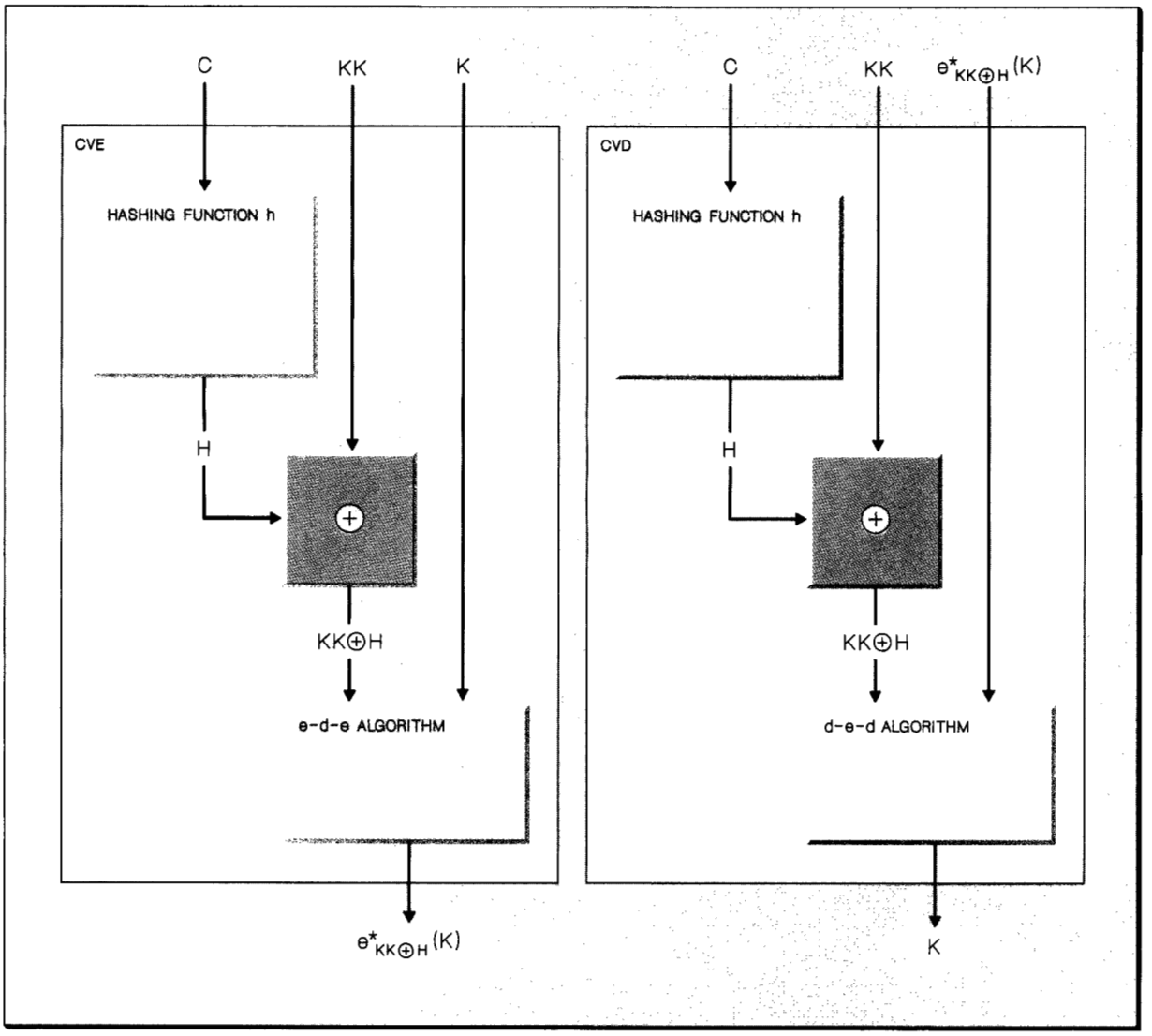
The first approach has the characteristic that K is recovered correctly at a using device only if the correct control vector is specified. Conversely, specification of an incorrect control vector does not prevent the decryption and recovery of a key, but the recovered key K′ is for all intents and purposes 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 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,[10] 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.

Control vector encryption and decryption algorithms. The control vector encryption (CVE) and control vector decryption (CVD) algorithms used to encrypt and decrypt a key, respectively, are illustrated in Figure 3. In the CVE algorithm in Figure 3, C is an input control vector whose length is a multiple of eight bytes; KK is a 128-bit key-encrypting key consisting of a leftmost 64-bit part KKL and a rightmost 64-bit part KKR, i.e., KK = (KKL,KKR); K is a 64-bit key or the leftmost or rightmost 64-bit part of a 128-bit key. The inputs are processed as follows. C is operated on by hashing function h (described in the following

Figure 3 CVE and CVD algorithms



subsection) to produce the 128-bit output H. H is exclusive-ORed with KK to produce 128-bit output KK⊕H. Finally, K is encrypted with KK⊕H to produce output $e^*_{KK\oplus H}(K)$, where e* indicates encryption with 128-bit key KK⊕H using an encryption-decryption-encryption (e-d-e) algorithm as defined in ANSI Standard X9.17-1985[6] and ISO Standard 8732.[7]

An encrypted key of the form $e^*_{KK\oplus H}(K)$ is decrypted with the CVD algorithm as depicted in Figure 3. 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⊕H. Finally, $e^*_{KK\oplus H}(K)$ is decrypted with KK⊕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 Figure 3 are described using key-encrypting key KK, KK could be replaced by a different key, such as the 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.

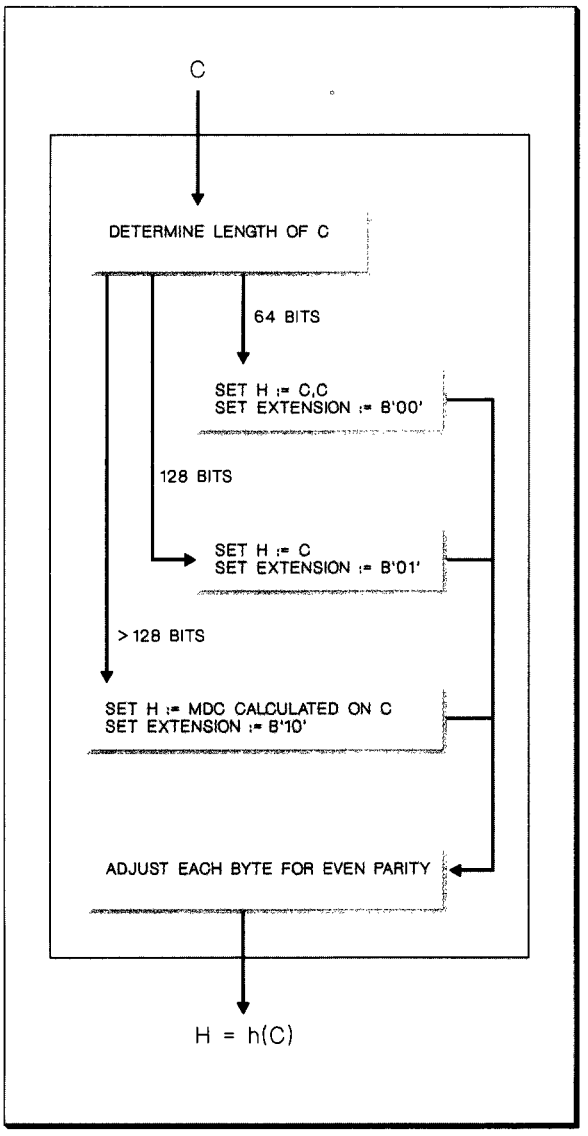
Hashing function h. The hashing function h implemented in the CVE and CVD algorithms is illustrated in Figure 4. Hashing function h operates on input control vector C (whose length is a multiple of 64 bits) to produce a 128-bit output H.

If C is 64 bits, h(C) is set equal to (C,C), where the comma 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 in h(C) is set equal to B'01'. That is, h acts like an identity function. If C is greater than 128 bits, h(C) is set equal to a 128-bit modification detection code calculated by the MDC-2 algorithm shown later in Figure 5, 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⊕h(C) has the same parity as KK (i.e., if KK has odd parity, then KK⊕h(C) also has odd parity). Adjusting bits 7, 15, 23, ..., etc. (i.e., 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 bit positions in the control vector must be reserved for use by hashing function h. For control vectors larger 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⊕h(C) consists of three disjoint subsets S1, S2, and S3, where S1 denotes the keys resulting from all 64-bit Cs, S2 denotes the keys resulting from all 128-bit Cs, and S3 denotes the keys resulting from all Cs larger than 128 bits. 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, suppose C1 is a control vector larger than 128 bits and $e^*_{KK\oplus h(C1)}(K)$ is 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, one cheats 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⊕h(C1) to recover K.

Figure 4 Hashing function h



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.

An MDC has a purpose similar to a MAC.

Modification detection code. A *modification detection code* (MDC) is a nonsecret cryptographic variable of fixed, relatively short length used to authenticate a message or plaintext of arbitrary, much longer length. An MDC has a purpose similar to a message authentication code (MAC). However, unlike a MAC, which is calculated with a secret key, an 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,[11] is illustrated in Figure 5. (MDCs are also discussed by Meyer and Schilling.[12]) 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 Figure 5, 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 based on output values (A1,D1), (A2,D2), ..., (An-1,Dn-1), and input values L2, L3, ..., Ln are based on output values (C1,B1), (C2,B2), ..., (Cn-1,Bn-1). That is, the outputs of each iteration are fed back and used as the keys at the next iteration. The 32-bit swapping function merely replaces Bi with Di and Di with Bi.

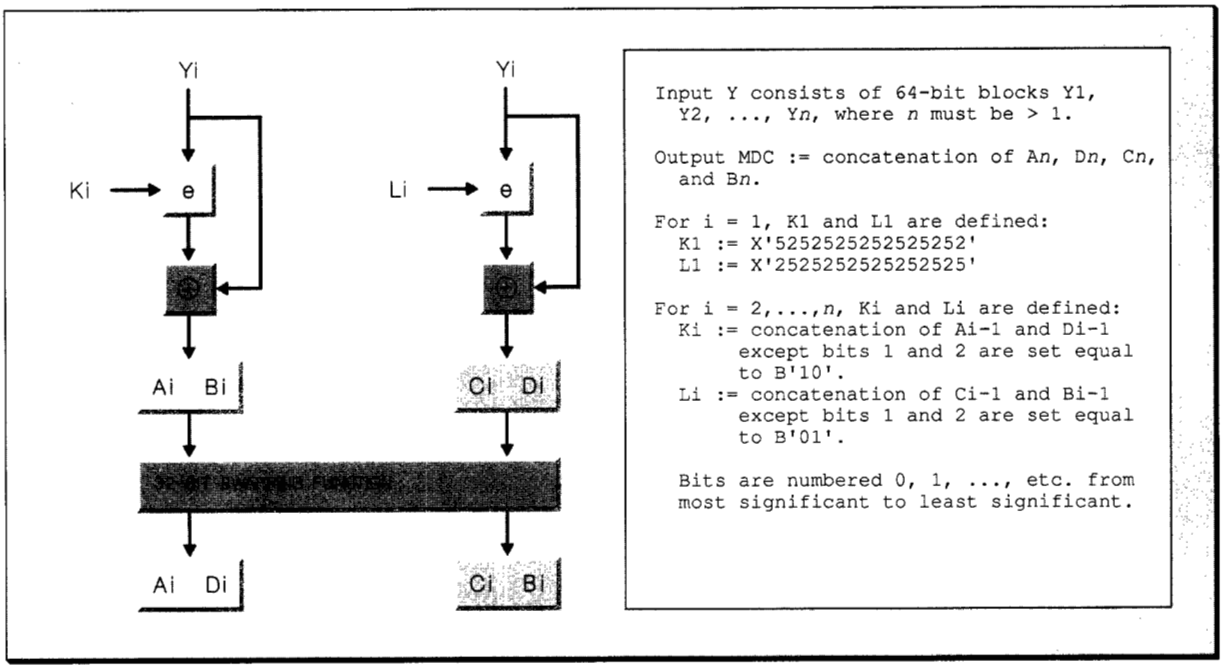
The MDC-2 algorithm processes data in multiples of 64 bits, with a 128-bit minimum. No padding is performed by the algorithm, although such padding could be performed as a service by either hardware or software. When padding is required, a padding algorithm f should be used that is guaranteed not to 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 $f(Y) \neq f(Y')$. A padding algorithm satisfying this requirement is given below. The method, which requires the input to consist of a whole number of bytes, is based on a padding rule described in ANSI X9.23.[13] (For convenience, the rule is described in terms of bytes not bits.) If the data length is less than eight bytes, pad bytes are added to make the data length 16. If the data length is eight or more bytes, pad bytes are added to make the data length a multiple of eight bytes. Padding is done even if the current data length is a multiple of eight bytes. All pad bytes except the last 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 eight bytes. Thus, an input Y equal to X'FFFFFFFFFFFFFFFF FFFFFFFFFFFF01' is not padded, since its length is already a multiple of eight bytes. But an input Y′ equal to X'FFFFFFFFFFFFFFFFFF FFFFFFFFFF' 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,[11] but its details are not discussed here.

Security of the CVE and CVD algorithms. The method of encryption and decryption with derived keys of the form KK⊕H provides an effec-

Figure 5 MDC-2 algorithm

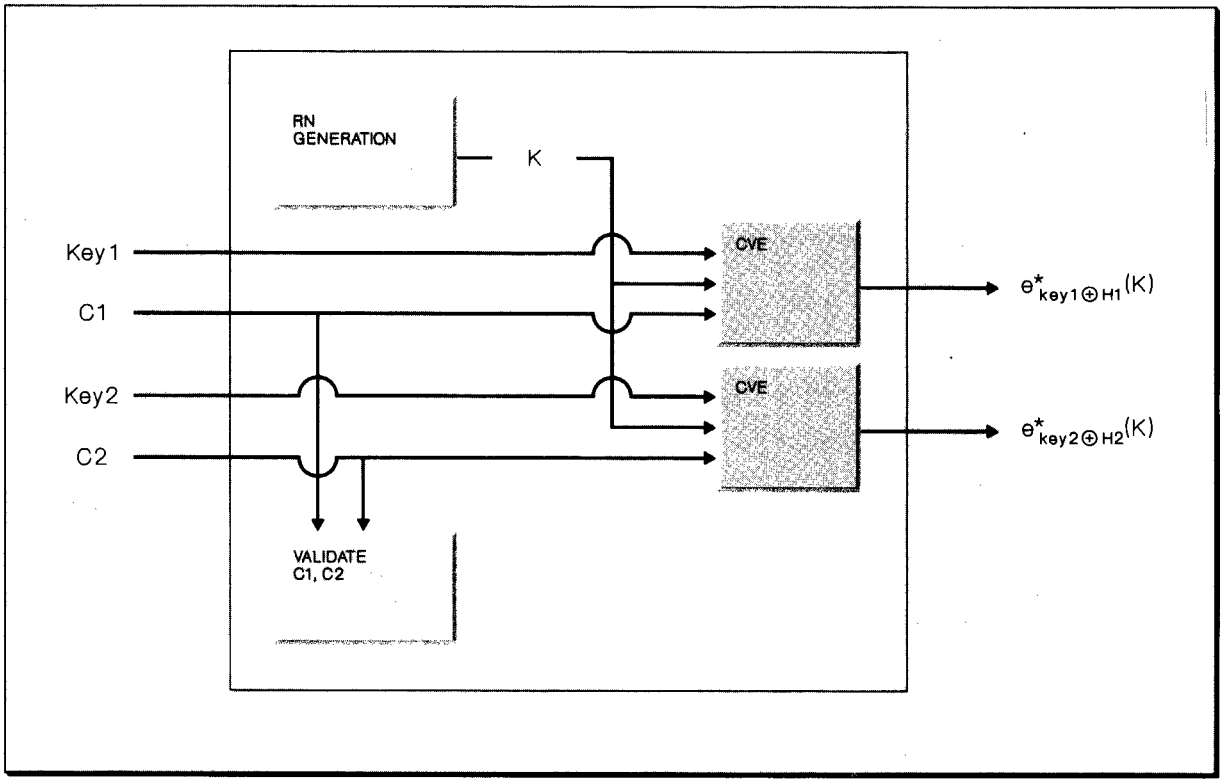


tive means to couple K and C, since given $e^*_{KK\oplus H}(K)$ and C, where h(C) = H, there is no apparent computationally efficient means to find alternative values of $e^*_{KK\oplus H'}(K)$ and C′, where h(C′) = H′, that give rise to the same recovered value of K. There is also a precedent for using derived keys for key-management purposes. The IBM and ANSI key-management schemes mentioned in the background section of this paper 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 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⊕H may accidentally equal the rightmost 64 bits of KK⊕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 $1/2^{56}$ (i.e., 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 a different KK⊕H. Methods of exhaustive search do not appear to be improved, nor does it appear that one can detect when the leftmost 64 bits of KK⊕H equal the rightmost 64 bits of KK⊕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⊕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⊕H to B'0' and 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 (i.e., 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, called "value," not equal to

Figure 6 Generating function G



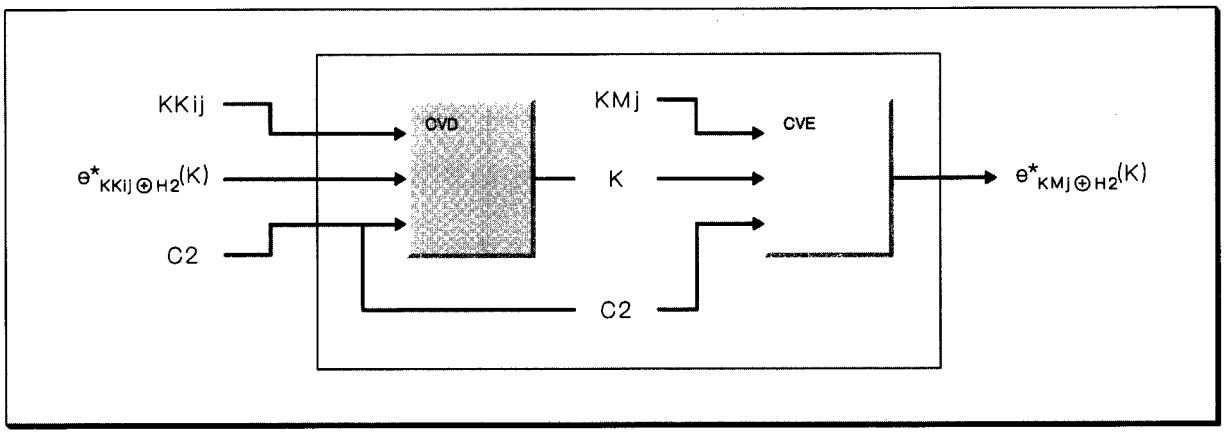
$e^*_{KK \oplus H}(K)$, i.e., by specifying inputs C, KK, and "value" to the CVD algorithm instead of inputs C, KK, $e^*_{KK \oplus H}(K)$. However, a good key-management design will ensure that such spurious keys are of no beneficial use to a would-be adversary. More is said about spurious keys in the section entitled Controlling Key Usage.

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 Figure 6. 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 the master key 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 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. Different key-generation modes and their uses within a key-management scheme are discussed in greater detail in a companion paper in this issue.[14]

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. Function G also validates (C1,C2) to ensure that both control vectors are consistent with and conform to the architectural specification (i.e., C1 and C2 represent a valid pair permitted by the key management). This validation is called control vector enforcement or control vector checking.

Figure 7 Key import with import function I



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 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 Figure 6, 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, say 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 Figure 7, where KMj is the master key of device j. Import function I consists of two steps: (1) execution of the CVD algorithm to decrypt $e^*_{KKij \oplus H2}(K)$ with KKij and C2 to recover K and (2) execution of the CVE algorithm to encrypt K with KMj and C2 to produce $e^*_{KMj \oplus H2}(K)$. The key token ($e^*_{KMj \oplus H2}(K)$, C2) is stored at device j.

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, e.g., from KKij to KMj using import function I. But the process is such that keys and control vectors re-

Table 1 Instructions and encrypted key inputs

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

main linked or coupled together so that one cannot replace the control vector of one key with that of another.

In order to control key usage effectively, one must link the usage of a key to usage information encoded in the control vector. A method for accomplishing this linkage is taken up next.

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 I1, I2, I3, and I4, where I1 and I2 each have one encrypted key input and I3 and I4 each have two encrypted key inputs. For convenience, the six encrypted key inputs are designated P1, P2, ..., P6. The relationship among the instructions and the encrypted key inputs is just as given in Table 1.

Within the toy system, every generated key can be used or processed in up to six ways, i.e., 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 designed within the control vector as part of the architecture, viz.,

U1	U2	U3	U4	U5	U6	...

Each Ui (for i = 1, ..., 6) is defined as follows:

Ui = 1 : The key associated with this control vector can be processed as input parameter Pi.
Ui = 0 : The key associated with this control vector cannot be processed as input parameter Pi.

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 to conveniently control *how* a key is used on the basis of *where* the key is used.

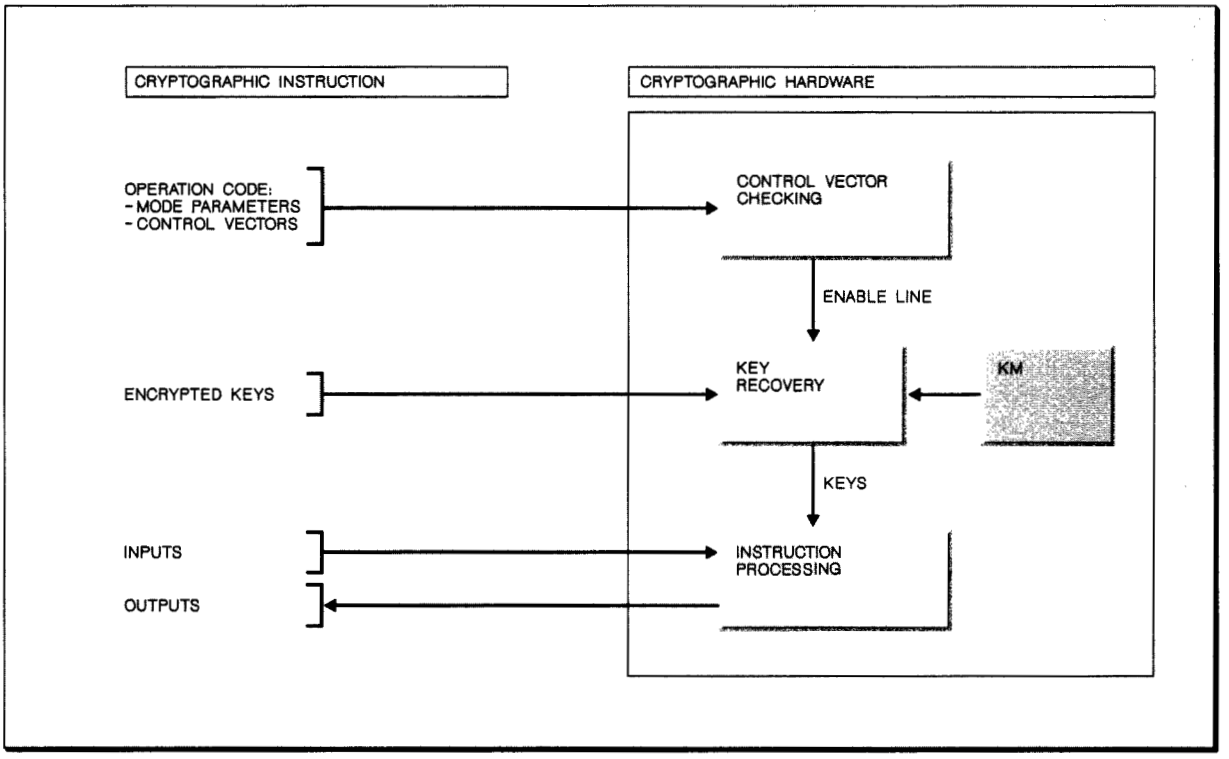
As a notational convenience, let ⟨u1,u2,u3,u4,u5,u6⟩ represent the encoding of the usage fields U1 through U6. The remainder of the bits of the toy system in C are spares, and thus are ignored by the cryptographic hardware. The encoding ⟨100000⟩ permits K to be processed as input key parameter P1 in cryptographic instruction I1. The encoding ⟨110000⟩ 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 Figure 8. 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 using the CVD algorithm previously described in Figure 3. Thereafter, the decrypted keys as well as additional input information are processed by the cryptographic instruction to produce one or more outputs.

If one cheats by specifying $e^*_{KM \oplus C1}(K)$ and C2 instead of $e^*_{KM \oplus C1}(K)$ and C1 (i.e., 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-

Figure 8 Control vector checking process



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 the other, since the keys recovered within the cryptographic hardware in that case are K and K′ (i.e., the first device has K and the second device has K′, or vice versa).

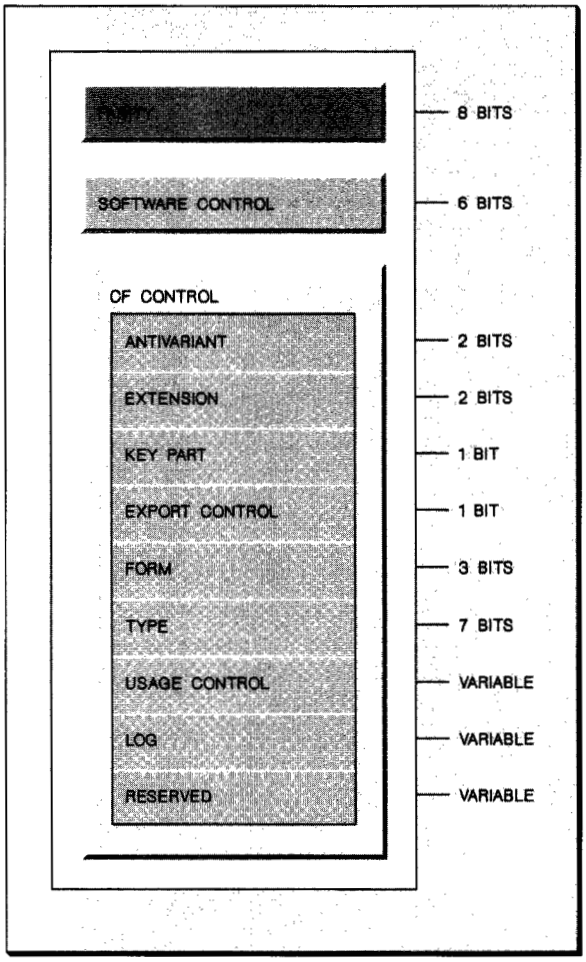
Instead of a control vector specification like the one shown at the beginning of this section, where a single control vector contains the usage attributes for all instructions, there may be multiple control vectors. A more intuitive control vector specification is achieved if separate control vectors are made a part of 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 processing 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 below. Control vector checking is similar except for the additional type field which must be checked.

Type 1	U1	U2	...

Type 2	U3	U4	U5	U6	...

The toy system provides a convenient method for introducing the general principles of control vector design. However, the power and versatility of the control vector are best understood and appreciated from the study of an actual implementation. Therefore, the remainder of the paper discusses a control vector design implemented in IBM's Transaction Security System. The key-management and cryptographic system design

Figure 9 Format of the control vector base



pursued in the Transaction Security System are discussed in companion papers in this issue.[14–16]

Control vector forms and formats

The key-management scheme implemented in the IBM Transaction Security System supports 64-bit control vectors and, in a few cases, 128-bit control vectors.

A 64-bit control vector, or the first 64 bits of a 128-bit control vector, are called a *control vector base*. The second 64 bits of a 128-bit control vector are called a *control vector extension*. The control vector extension supports special-purpose key management such as installation-specific key-management controls and also provides space for future uses.

Format of the control vector base. Figure 9 illustrates the general format of the control vector base. The control vector base consists of three primary subvectors: a parity subvector, a cryptographic facility access program (CFAP) subvector, and a cryptographic facility (CF) subvector. The CF and CFAP are the hardware and software components of a cryptographic system (also see Reference 14).

Parity subvector. The *parity* subvector (depicted as a single field) is composed of eight parity bits whose bit locations are structured according to the architecture to coincide with the eight parity bits in a 64-bit DEA key. The parity subvector has even parity, i.e., the parity bit in each byte is adjusted so that the number of "1" bits in each byte is an even number. The parity bits are set by the cryptographic software and are ignored by the cryptographic hardware. Maintaining even parity in the control vector ensures that for a given key all keys derived from that given key will have the same parity as the original key. Even parity avoids potential hardware incompatibilities that may result during key processing, e.g., if processed by an encryption or decryption chip that requires keys to have odd parity.

CFAP control subvector. A *CFAP control* subvector is a collection of fields used by the CFAP for key-handling and key-management purposes. This subvector is not processed by the CF, i.e., the hardware ignores this field. A bit in the CFAP control subvector indicates whether the associated encrypted key may be specified as an input parameter to the CFAP at the application program interface (API) or whether the key must be stored in a cryptographic key data set and accessed on the basis of a key label specified as an input parameter to the CFAP at the API. Greater integrity is achieved if keys are managed by the system. In that case, a system-level access control program such as the IBM Resource Access Control Facility (RACF) can be the means to restrict the use of cryptographic keys to authorized users only.

CF control subvector. The *CF control* subvector is a collection of fields used by the cryptographic facility for key-handling and key-management purposes.

CF control subvector field descriptions. The CF control subvector contains the following nine fields: antivariant, extension, key part, export control, form, type, usage control, log, and reserved.

Antivariant. The *antivariant* field is used to cryptographically distinguish control vectors from variants, i.e., to ensure that an encrypted key of the form $e^*_{KK\oplus V}(K)$, where V is the concatenation of a 64-bit variant mask v with itself (i.e., V = v,v), cannot be substituted and used in a meaningful way for an encrypted key of the form $e^*_{KK\oplus H}(K)$, where H is derived from a control vector using hashing function h, and vice versa. In a key-management scheme based on key variants, each variant mask v is a 64-bit value formed by replicating a single eight-bit value eight times. Each eight-bit value has even parity, and the values are selected so that no two eight-bit values are complements of each other. The antivariant bits are designed in an architecture so that at least two bytes in the control vector are unequal, thus ensuring that the set of all values of the form h(C) and the set of all 128-bit variant mask values (v,v) are disjoint. This permits architectures based on control vectors and variants to coexist without harmful side effects, i.e., encrypted keys conforming to one architecture cannot be substituted and used for keys of another architecture.

Encrypted keys conforming to one architecture cannot be substituted and used for keys of another architecture.

Extension. The *extension* field (not to be confused with the control vector extension) indicates the length of C, i.e., 64-bit, 128-bit, or greater than 128-bit. The extension field in C is structured to coincide with the extension field in h(C), which was discussed in the subsection on hashing function h.

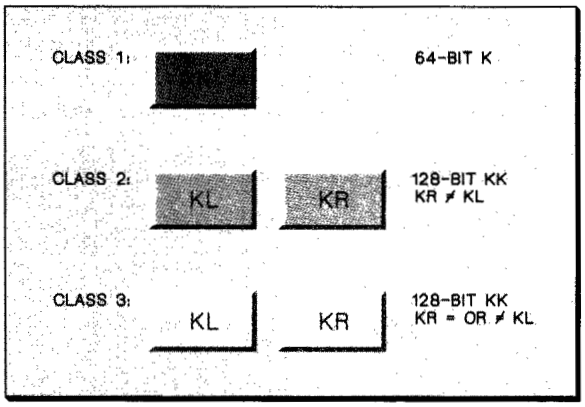
Key part. The *key-part* field indicates whether the cryptographic variable associated with the control vector is (1) a key or (2) an intermediate key value. An intermediate key value consists of a single key part or the exclusive-OR product of two or more key parts. The key-part field in the control vector works in combination with the cryptographic instructions to control the importing of keys. The cryptographic instructions perform key installation via the entry of two or more key-part values, which when combined within the cryptographic hardware, form the key. Thus, each part of the key may be entered separately by different people.

Export control. The *export control* field indicates whether the key or cryptographic variable associated with the control vector (1) may be exported or (2) may not be exported. The term "export" refers to a key-management process in which a key or cryptographic variable is translated from encryption under the master key to encryption under a key-encrypting sender key KESK, which permits the key to be transmitted to a receiving device where it is imported. The task of "exporting" a key is performed by a cryptographic instruction which re-enciphers the key from encipherment under the master key to encipherment under a key-encrypting sender key. However, this operation is performed only if the export control field in the control vector of the key indicates that the key may be exported. A cryptographic instruction is also available to reset the export control field from the "export allowed" state to the "export not allowed" state. However, the export control field cannot be changed from the export-not-allowed state to the export-allowed state.

Form. The *form* field indicates whether the 64-bit variable associated with the control vector is (1) a class 1 key, (2) the leftmost 64 bits of a class 2 key, (3) the leftmost 64 bits of a class 3 key, (4) the rightmost 64 bits of a class 2 key, or (5) the rightmost 64 bits of a class 3 key. The term "class" is used here only to help distinguish the five cases. A class 1 key is a 64-bit key. A class 2 key is a 128-bit key in which the leftmost and rightmost 64-bit parts of the key are independently generated. A class 3 key is a 128-bit key in which the leftmost and rightmost 64-bit parts of the key are independently generated or deliberately set equal for compatibility with 64-bit keys. Figure 10 illustrates the three supported classes of keys.

The form field provides cryptographic separation among the three classes of keys and, for class 2

Figure 10 Classes of keys



and class 3 keys, between the leftmost and rightmost 64-bit parts of the key. In effect, it ensures that the class of the key specified at the instruction interface is permitted by the instruction, and it ensures that the leftmost 64 bits and rightmost 64 bits of 128-bit keys cannot be interchanged and used incorrectly.

Control vector type. The *control vector type* field indicates what type of 64-bit variable is associated with the control vector. The control vector type field consists of a *main-type* and a *sub-type*. The main-type defines broad types of keys and cryptographic variables, whereas the sub-type defines particular types of keys within each main-type. Together, the main-type and sub-type fields facilitate the definition of a set of generic key and cryptographic variable types using a hierarchical naming structure. Figure 11 shows the control vector main-types and sub-types.

Keys and cryptographic variable names are formed as a concatenation of the main-type name and the sub-type name, except that the word "key" appearing in main-type is moved to the end. Thus, main-type = "data key," and sub-type = "compatibility" becomes data compatibility key. A short description of the use of each key and cryptographic variable type is provided below. In the specification of names for main-type, the term "cryptovariable" is used as an abbreviation for cryptographic variable.

The *data compatibility key* is used to maintain compatibility with devices not implementing the control vector. Usage control bits permit the key to encipher data, decipher data, generate MACs, and verify MACs.

The *data privacy key* is used to support cryptographic applications requiring encipherment and decipherment of data. Usage control bits permit encipher-only and decipher-only operations.

The *data MAC key* is used to support cryptographic applications requiring the generation and verification of MACs. Usage control bits permit generation-only and verification-only of MACs.

The *data privacy-translate key* is used to support cryptographic applications requiring ciphertext translation. Usage control bits permit decrypt-only or encrypt-only operations in the ciphertext translate instruction. A data privacy-translate key operates together with a data privacy key. A data privacy-translate key can only decrypt ciphertext originally encrypted with a data privacy key. Likewise, plaintext re-encrypted with a data privacy-translate key can only be decrypted with a data privacy key.

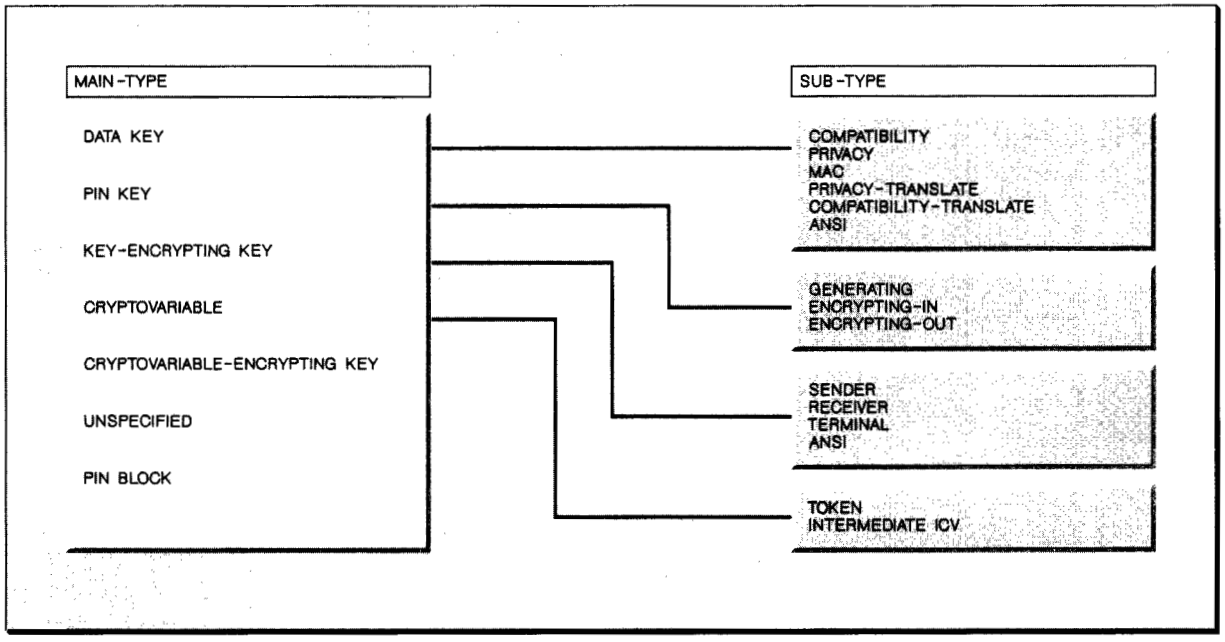
The *data compatibility-translate key*, like the data privacy-translate key, is used to support cryptographic applications requiring ciphertext translation. Usage control bits permit decrypt-only or encrypt-only operations in the ciphertext translate instruction. A data compatibility-translate key operates together with a data compatibility key. A data compatibility-translate key can only decrypt ciphertext originally encrypted with a data compatibility key. Likewise, plaintext re-encrypted with a data compatibility-translate key can only be decrypted with a data compatibility key.

The *data ANSI key* is used to support ANSI X9.17 key management. Usage control bits permit the key to encipher data, decipher data, generate MACs, and verify MACs.

The *PIN-generating key* is used to support financial transactions requiring dynamic user PIN (personal identification number) verification. Usage control bits permit the key to generate clear and encrypted PINs and to generate reference PINs used in PIN verification.

The *PIN-encrypting-in key* is used to decrypt inbound PINs processed by the PIN instructions. Us-

Figure 11 Control vector hierarchy



age control bits permit the key to be used in CF instructions for verifying encrypted PINs, generating clear offsets, translating PINs, and reformatting PINs.

The *PIN-encrypting-out key* is used to encrypt outbound PINs processed by PIN instructions. Usage control bits permit the key to be used in CF instructions for formatting and encrypting PINs, generating formatted and encrypted PINs, translating PINs, and reformatting PINs.

The attributes "in" and "out" associated with the PIN-encrypting key cut down the amount of freedom a would-be adversary would have to manipulate the cryptographic instruction interfaces in an attempt to subvert security. In effect, they permit a finer degree of control to be achieved in the management and processing of PINs.

The *key-encrypting sender key* is used at a sending device to encrypt keys transmitted to a receiving device. Usage control bits permit the key to be used in CF instructions for generating keys, exporting keys, and translating keys.

The *key-encrypting receiver key* is used at a receiving device to decrypt keys received from a sending device. Usage control bits permit the key to be used in CF instructions for generating keys, importing keys, and translating keys.

The key-encrypting sender key and key-encrypting receiver key control vectors permit establishment of a unidirectional, or one-way, key-distribution channel between two devices.

The *key-encrypting terminal key* is used to maintain compatibility with terminal devices not implementing the control vector. Usage control bits permit certain key types to be encrypted under this key for electronic distribution to a terminal.

The *key-encrypting ANSI key* is used in support of ANSI X9.17 key management to electronically transmit and receive keys.

The *cryptovariable-encrypting key* is used to encrypt and decrypt cryptographic variables processed by certain cryptographic instructions.

The *PIN block*, *cryptovariable token*, *cryptovariable intermediate ICV*, and *unspecified* control vectors are not defined here.

The architectural advantages of multiple types of control vectors, as specified by the control vector

type field, are these. First, *shorter* control vectors are possible, since each key type is processed by fewer cryptographic instructions. Thus, fewer usage control bits are required to control key processing. Second, if the control vector type and usage control fields are designed with key generation in mind, the control vector type field can be used advantageously to simplify control vector checking during key generation. The idea is to define the control vector type and usage control fields in such a way that generating function G can validate each input control vector pair (C1,C2) on the basis of checking the control vector type fields in C1 and C2, but not checking the usage control fields. To do this, function G must store a list of valid pairs of control vector types. The usage control fields are designed so that any combination of usage bits, in either C1 or C2, is permitted. That is, no specification of the usage control field leads to a conflict in consistency or security. Third, the specification of keys and cryptographic variables via the control vector type field, where control vector types indicate broad application areas served by these keys and cryptographic variables, leads to an *intuitive* specification. Also, the key types are consistent with key types defined for existing cryptographic products. In addition, key types are chosen to closely parallel the generic uses of keys from the perspective of the user and application program. Structuring the control vector type field in this way makes it easier to understand, and it should permit the key-management architecture to be more easily extended.

Usage control. The *usage control* field specifies the permitted uses of the 64-bit variable (64-bit key, leftmost or rightmost 64-bit parts of a 128-bit key, or a 64-bit cryptographic variable) associated with the control vector. The usage control field consists of a collection of subfields, unique for each control vector type. That is, the subfields defined for a data privacy key are different from those defined for a key-encrypting sender key, or for a PIN-encrypting-in key, etc.

Characteristics of the usage control field and the architectural rules pursued in its design follow. Generally, each subfield in the usage control field is a bit which either enables (set to 1) or disables (set to 0) the use of the key or cryptographic variable as a specific input parameter to a specific cryptographic instruction. However, there are cases where this level of control is limited only to particular instruction modes. In other cases it is broadened to include a group of like or related cryptographic instructions. Generally speaking, the usage control field enables a cryptographic application, via the cryptographic software, to limit or restrict the usage of a key in order to achieve a different measure or level of cryptographic security. Instruction mode parameters

The usage control field specifies the permitted uses of the 64-bit variable associated with the control vector.

are the external means used to control instruction execution. The control vector type and usage control fields are designed to facilitate and simplify control vector checking. They are also designed to provide a high degree of security by avoiding control vector specifications that lead to conflicting key-usage attributes.

Log. The *log* field, used only in the key-encrypting sender key and key-encrypting receiver key control vectors, provides an audit trail of the usage control field. If (C1,C2) are the control vectors of a to-be-generated key, where C1 is a key-encrypting sender key control vector and C2 is a key-encrypting receiver key control vector, the usage control field in C1 is stored in the log field in C2 and the usage control field in C2 is stored in the log field in C1. In this way the user of each key is permitted to see the usage attributes of the other key.

Reserved. The *reserved* field provides a control-vector-based key management with a means for extending the architecture to meet future requirements, with minimal or no impact on products implementing the control vector.

Example of key-usage control

The process of key-usage control can be illustrated by looking at a typical control vector layout and the control vector checking steps used in a typical cryptographic instruction to check

the control vector. For example, consider a data privacy key control vector of the form X'0003600003000000'. Table 2 shows the bit layout for this control vector.

The control vector type field (bits 08–14) is encoded as B'0000001', indicating that the key is a data privacy key. The export control field (bit 17) is encoded as B'1', indicating that the key can be exported. The usage control field (bits 18–19) is encoded as B'10', indicating that the key can be used to encipher data but not decipher data. That is, the "encipher" usage control bit is B'1' and the "decipher" usage control bit is B'0'. The antivariant field (bits 30 and 38) is encoded as B'01', indicating that the field is properly set. The CFAP control field (bits 32–37) is unchecked by the cryptographic hardware. The form field (bits 40–42) is encoded as B'000', indicating that the key is a 64-bit key. The key part field (bit 44) is encoded as B'0', indicating that it is a key, not a key part. The extension field (bits 45–46) is encoded as B'00', indicating that the control vector is a 64-bit control vector. The remainder of the control vector bits are reserved and untested.

A data privacy key and control vector are processed in the CF instructions that encipher and decipher data. The control vector checking steps performed in the CF instruction that enciphers data are listed below:

1. Control vector type field encoded as data compatibility key or data privacy key or data ANSI key
2. "Encipher" bit in usage control field in "enabled" state
3. Form field encoded as "64-bit key"
4. Extension field encoded as "64-bit CV" or "128-bit CV"
5. Valid antivariant field required
6. Key-part field encoded as "key"

This relatively simple set of control vector checking steps illustrates several things about the control vector checking process. First, the checking steps are independent of one another, so that they can be performed in any order. In fact, because of this *independence*, it is possible to allocate control vector checking among multiple processors. In that case, each processor performing control vector checking reports the outcome to the processor that executes the instruction, and if trustworthy positive responses are received from each processor, instruction execution is enabled. For example, a smart card used at a workstation could, in theory, be personalized to perform installation-unique control vector checking, e.g., using installation-specified fields in the control vector extension. In that case, control vector checking of the control vector base is performed by the cryptographic hardware of the device, and control vector checking of the control vector extension is performed by the smart card. Such installation-unique checking might include a provision for keys to be used only during specific time periods, or days of the week.

Table 2 Bit layout for data privacy key control vector X'0003600003000000'

Bit Positions	Bits
00–07	0000 0000
08–15	0000 0011
16–23	0110 0000
24–31	0000 0000
32–39	0000 0011
40–47	0000 0000
48–55	0000 0000
56–63	0000 0000

Second, each control vector checking step can test a field for one value, or for one of several possible permitted values. For example, the encipher instruction permits data to be enciphered with three different key types, viz., data compatibility key, data privacy key, and data ANSI key. The form, antivariant, and key-part fields are tested for specific values. This testing means that the control vector checking process accepts as valid any one of possibly many different control vectors, not just one control vector.

Third, control vector checking is *sparse*. For example, of the 2^{56} possible 64-bit control vector base values that can serve as input to the encipher instruction, 6×2^{40} of these are valid. Of the 16 tested bits, there are six valid combinations and, of the 40 untested bits, there are 2^{40} valid combinations, giving a total of 6×2^{40} valid combinations. The bits in the parity field are ignored. In contrast to control vector checking, for a key-management scheme based on key variants, where 64-bit key-variant values are somewhat arbitrarily assigned by the architecture, the analogous checking process is therefore based on a 64-bit encoded key variant. In that case, to check

any one of the 6×2^{40} different 64-bit vectors requires a lookup table containing 6×2^{40} entries—which will not work! The power of the control vector over the key variant is evident.

Table 2 shows an example of a control vector base value that passes the control vector checking process for the encipher instruction. The reader may find it instructive to trace each checking step and verify that the encoded value in each control vector field checked by the encipher instruction is indeed valid.

Control vector checking is sparse.

Control vector enforcement

The process of control vector checking described in the section entitled Control Vector Forms and Formats is in reality only one way in which the cryptographic facility (CF) can ensure that the set of control vectors specified to a cryptographic instruction are consistent and permitted. This more general process is called control vector enforcement. Control vector checking is just one way to accomplish control vector enforcement.

The following methods or combinations of methods may be used to accomplish control vector enforcement:

- Specify control vector in CFAP and check control vector bits in CF: This method checks bits and fields within the control vector to ensure that they contain permitted values. In certain cases, cross-checking of bits and fields among two or more control vectors is necessary to ensure that they contain only permitted combinations of values.
- Specify control vector in CFAP and set control vector bits in CF: This method sets bits and fields within the control vector to prescribed values (i.e., by overwriting the bits and fields of the control vectors passed at the instruction interface).
- Generate control vector in CF from information specified by CFAP: This method generates control vectors from parameter information passed at the instruction interface.
- Table lookup of control vector in CF from index specified by CFAP: This method uses a table of control vectors stored within the CF. An index value passed at the instruction interface selects the control vector or vectors used by an instruction.
- Control vector implicitly specified: The CF instruction uses a fixed set of control vectors. Since there is no variability, the control vectors are stored as constants in the instruction.

In the key-management scheme implemented in the Transaction Security System, control vector enforcement is accomplished using the first and third methods (i.e., control vector checking and control vector generation). The first method permits errors in the control vector to be detected and reported to the CFAP. The third method reduces the number of control vectors that must be specified at the instruction interfaces, thereby reducing complexity. However, this paper deals only with control vector checking.

Concluding remarks

Control vectors contrasted with variants. The control vector consists of a set of structured fields whose encoded values and meanings are defined by the architecture, and a set of unstructured fields and code points reserved for future use. The structured fields specify a control vector type and a set of key-management and key-usage control fields that record information about the key or cryptographic variable and define its permitted uses. The architecture is extended by defining new fields in the control vector or by adding to the specification of an existing field.

In contrast, the variant mask is a single field consisting of a set of encoded mask values (generally a small set). The undefined code points are reserved for future use. Each variant mask value defines a key or cryptographic variable with a particular set of assigned key-management and key-usage attributes. The key-management architecture is extended by defining additional variant mask values.

The major difference between the control vector and the variant mask is that the former can support a very rich key-management scheme with

many key-management and key-usage control attributes, and the latter cannot. First, note that for two comparable key-management schemes that must support multiple orthogonal key-management and key-usage control attributes, one based on control vectors and the other on variant mask values, the number of encoded control vector values N1 is about equal to the number of variant mask values N2. In either case, a control vector or a variant mask value is validated by confirming that it is an element of a set of allowed values S, where S is a subset of the set of all possible values. But for control vectors, set membership is conveniently determined, even for large N1, by checking one or more fields in the control vector. However, with variants there is no shortcut; one must store and search a list of N2 values. Therefore, for large N2, variant mask checking cannot be implemented efficiently.

An important difference between the key tag and the control vector is that the key tag is available in clear form only within the cryptographic hardware.

Control vectors contrasted with key tags. Several attributes of the Jones' key tag make it inappropriate for use as a general methodology for controlling key usage. Each 64-bit key has only eight bits available for a key tag. The control vectors defined by the key-management scheme implemented in the Transaction Security System already make use of the majority of the bits in the control vector base, and therefore, the key tag could not be used to implement this key-management architecture. Moreover, DEA-based products and equipment that use the eight nonkey bits for parity checking, or that must maintain compatibility with those products and equipment that do, cannot use the key tag.

An important difference between the key tag and the control vector is that the key tag is available in clear form only within the cryptographic hardware. The control vector is carried along with the encrypted key, in clear form, in an external key token. The key tag is recovered in clear form when the key is decrypted, whereas the control vector is supplied to the cryptographic hardware at the instruction interface, thus permitting the key to be decrypted. Therefore, with the key tag, key-usage control is limited to that which can be performed in the cryptographic hardware. In contrast, with the control vector, key-usage control can be effected in multiple locations, including the cryptographic hardware, the cryptographic software, a cryptographic software-provided installation exit, or the application program. Another difference is that, with the control vector, key usage is controlled at the instruction interface. With the key tag, key usage is controlled at the DEA interface (i.e., to control elementary encrypt and decrypt operations with the key).

Advantages of the control vector. For some cryptographic devices, especially those offering limited, special-purpose, or self-contained cryptographic functions, key variants may provide suitable key separation and key-usage control. For general-purpose cryptographic devices, where very high-speed cryptographic operations are required or it is impractical or infeasible to implement full hardware control vector checking, key separation and key-usage control may be advantageously effected through the use of a subset of frequently used control vector values, called a generic subset. In that case, the control vector values are stored in tabular form within the cryptographic hardware, and a fast table lookup method is used to enforce proper control vector usage. This also ensures compatibility with other devices implementing the same control vector set. This method of control vector implementation is the one pursued in a companion paper in this issue.[17]

However, in situations where the cryptographic hardware implementation permits full field-by-field control vector checking, a much finer granularity in key separation and key-usage control is possible. There are several other advantages of a key-management architecture based on control vectors. Many of these are already obtained in the key-management scheme implemented in the Transaction Security System.

The control vector has no restriction on length. In theory, a key-management architecture based on

control vectors can be grown indefinitely (i.e., the architecture is said to be *open-ended*).

The control vector consists of a set of fields separately specified in an architecture. These fields control key usage at the CF cryptographic instruction interface and, to a lesser degree, at the CFAP cryptographic function interface (i.e., at the API).

The control vector consists of a set of fields separately specified in an architecture.

Positioning the controlling mechanisms at these points permits a fine degree of key-usage control to be realized in the key-management architecture.

Experience has shown that each CF cryptographic instruction needs to check only a fraction of the control vector fields within any one control vector. That is, only *sparse* checking is needed. Therefore, a certain economy of scale is realized in the control vector checking process, which is unattainable with other methods (e.g., using key variants).

Experience has also shown that many control vector fields have no dependence on other control vector fields, and the checking performed on these fields is totally independent of the checking performed on other fields. When dependencies do exist, they generally involve only a few fields, so that the necessary cross-checking is far less than, in theory, it might be. This independence and weak dependence among the control vector fields means that control vector checking can be performed as a series of independent checking steps, in no particular order.

Because control vector checking consists of a series of independent checking steps, control vector checking can be performed by multiple parallel or distributed processors. As already pointed out, installation-specific control vector fields in the control vector extension could, in theory, be checked by a smart card which has been personalized in advance with an installation-specific control vector checking program. The control vector checking performed on the control vector base, e.g., at a workstation, is left unaltered and can service many different installation and network configurations.

Because control vector checking consists of a series of independent checking steps, control vector checking can be serialized and performed advantageously by different software programs and hardware components. For example, consider a cryptographic service request issued by an application program to CFAP, which in turn results in a cryptographic instruction issued to the CF. A user-defined field in the control vector extension could be checked by the application program prior to its issuing the service request to the CFAP. An installation-defined field in the extension could then be checked by an installation-provided program (executed via a CFAP installation exit), and a CFAP field in the control vector base can be checked by the CFAP prior to issuing the cryptographic instruction. The CF fields in the control vector base are then checked by the instruction prior to executing the instruction. If checking fails at any stage, the process is halted.

The design is such that control vector checking is independent of (a) the functional processing performed by each instruction, (b) the encryption and decryption algorithms (CVE and CVD) used by the key management to protect keys and to cryptographically couple the keys and control vectors, and (c) the key-distribution protocol. Thus, new fields and code points can be designed in the control vector within the architecture and additional control vector checking can be added to existing checking procedures without affecting (a) present control vector fields, (b) present control vector checking procedures, (c) present instruction functional processing, and (d) present instruction interfaces.

The control vector permits a key-management architecture to be designed such that the majority of the complexity associated with the key-management architecture is embodied as encoded fields within the control vector. The control vector checking procedures may be made as general as possible, so that common checking routines can be used by several cryptographic instructions. Conversely, the common set of functions for gen-

erating keys, producing derived keys, and encrypting and decrypting keys using the control vector are made as simple and straightforward as possible. That is, key-management complexity is localized in the control vector, not in the functions that process control vectors and keys. Thus, the control vector gives rise to fewer and simpler cryptographic functions to ensure cryptographic separation, thereby reducing the work necessary to certify the cryptographic security.

Since the control vector is a structured variable, the encoded fields and code points present in the control vector match the cryptographer's intuition about the way a key-management scheme operates. This structuring aids in understanding the key-management architecture at all levels of cryptographic product and application design and improves the implementation and use of those products.

Each CF instruction and the control vector checking steps for each CF instruction can, but need not, be implemented within the same component of the cryptographic system. Thus, an implementer has a choice of where control vector checking can be implemented most advantageously within the cryptographic device.

Acknowledgments

The author wishes to acknowledge C. H. Meyer and B. Brachtl who collaborated with him on an initial idea for controlling key usage that eventually led to the control vector. The author also wishes to acknowledge D. B. Johnson, R. K. Karne, A. V. Le, R. Prymak, and J. D. Wilkins for their efforts in codeveloping the control vector.

Cited references

1. American National Standard X3.92-1981, *Data Encryption Algorithm*, American National Standards Institute, New York (December 31, 1981).
2. 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**, No. 2, 106–125 (1978).
3. S. M. Matyas and C. H. Meyer, "Generation, Distribution, and Installation of Cryptographic Keys," *IBM Systems Journal* **17**, No. 2, 126–137 (1978).
4. R. E. Lennon, "Cryptography Architecture for Information Security," *IBM Systems Journal* **17**, No. 2, 138–150 (1978).
5. M. E. Smid, *Notarization System for Computer Networks*, NBS Special Publication 500-54, U.S. Department of Commerce, National Bureau of Standards (now NIST), Washington (October 1979).
6. American National Standard X9.17-1985, *American National Standard for Financial Institution Key Management (Wholesale)*, American Bankers Association, Washington (1985).
7. International Standard ISO 8732, *Banking—Key Management (Wholesale)*, International Organization for Standardization, ISO Central Secretariat, Geneva, Switzerland (15 November, 1988).
8. R. W. Jones, "Some Techniques for Handling Encipherment Keys," *ICL Technical Journal* **3**, No. 2, 175–188 (November 1982).
9. D. W. Davies and W. L. Price, *Security for Computer Networks*, Second Edition, John Wiley & Sons, Inc., New York (1989), pp. 154–157.
10. American National Standard X9.9-1986, *American National Standard for Financial Institution Message Authentication (Wholesale)*, American Bankers Association, Washington (1986).
11. 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).
12. C. H. Meyer and M. Schilling, "Secure Program Load with Modification Detection Code," *Proceedings of the 5th Worldwide Congress on Computer and Communications Security and Protection* SECURICOM 88 - SEDEP, 8, Rue de la Michodiere, 75002 Paris, France (1988), pp. 111–130.
13. American National Standard X9.23-1988, *American National Standard for Financial Institution Encryption of Wholesale Financial Messages*, American Bankers Association, Washington (1988).
14. S. M. Matyas, A. V. Le, and D. G. Abraham, "A Key-Management Scheme Based on Control Vectors," *IBM Systems Journal* **30**, No. 2, 175–191 (1991, this issue).
15. D. B. Johnson et al., "Common Cryptographic Architecture Cryptographic Application Programming Interface," *IBM Systems Journal* **30**, No. 2, 130–150 (1991, this issue).
16. D. B. Johnson and G. M. Dolan, "Transaction Security System Extension to the Common Cryptographic Architecture," *IBM Systems Journal* **30**, No. 2, 230–243 (1991, this issue).
17. P. C. Yeh and R. M. Smith, Sr., "ESA/390 Integrated Crytographic Facility: An Overview," *IBM Systems Journal* **30**, No. 2, 192–205 (1991, this issue).

Stephen M. Matyas *IBM Federal Sector Division, 9500 Godwin Drive, Manassas, Virginia 22110.* Formerly a member of the Cryptography Center of Competence at the IBM Kingston Development Laboratory, Dr. Matyas is currently a member of the Secure Products and Systems department at Manassas, Virginia. He has participated in the design and development of all major IBM cryptographic products, including the IBM Cryptographic Subsystem, and recently he has had the lead role in the design of the cryptographic architecture for IBM's recently announced Transaction Security System. Dr. Matyas

holds 26 patents and has published numerous technical articles on all aspects of cryptographic system design. He is the coauthor of an award-winning book entitled *Cryptography—A New Dimension in Computer Data Security*, published by John Wiley & Sons, Inc. He is a contributing author to the *Encyclopedia of Science and Technology*, and *Telecommunications in the U.S.—Trends and Policies*. Dr. Matyas received a B.S. in mathematics from Western Michigan University and a Ph.D. in computer science from the University of Iowa. He is the recipient of an Outstanding Innovation Award for his part in the development of the Common Cryptographic Architecture. He is presently an IBM Senior Technical Staff Member.

Reprint Order No. G321-5428.