White-Box Cryptography and an AES Implementation*

S. Chow, P. Eisen, H. Johnson, P.C. van Oorschot

Cloakware Corporation, Ottawa, Canada, K2L 3H1
{stanley.chow, phil.eisen, harold.johnson, paulv}@cloakware.com

Abstract. Conventional software implementations of cryptographic algorithms are totally insecure where a hostile user may control the execution environment, or where co-located with malicious software. Yet current trends point to increasing usage in environments so threatened. We discuss encrypted-composed-function methods intended to provide a practical degree of protection against *white-box* (total access) *attacks* in untrusted execution environments. As an example, we show how AES can be implemented as a series of lookups in key-dependent tables. The intent is to hide the key by a combination of encoding its tables with random bijections representing compositions rather than individual steps, and extending the cryptographic boundary by pushing it out further into the containing application. At present, we can only partly justify our AES implementation, and motivate its design, by showing how removal of parts of the recommended implementation makes the implementation less secure. Full security analysis is future work for a wider cryptographic community.

1 Introduction and Overview

There has been tremendous progress in the uptake of cryptography within computer and communications applications over the past ten years. Unfortunately, the attack landscape in the real world has also changed. In many environments, the standard cryptographic model — assuming that end-points are trusted, mandating a strong encryption algorithm, and requiring protection of only the cryptographic key — is no longer adequate, for several reasons, including increasing penetration of commercial applications involving cryptography into untrusted, commodity host environments. An example is the use of cryptography in content protection for Internet distribution of e-books, music, and video. The increasing popularity of the Internet for commercial purposes illustrates that users wish to execute, and vendors will support, sensitive software-based transactions on physically insecure system components and devices. This sets the stage for our work.

The problem we seek to address is best illustrated by considering the software implementation of a standard cryptographic algorithm, such as RSA or AES[13],

* We thank Alexander Shokurov of ISPRAS, who suggested the ambiguity metric of section 4.1 in another context, and the anonymous reviewers for many improvements.

on an untrusted host. At some point in time, the secret keying material is in memory. Malicious software can easily search memory to locate keys, looking for randomness characteristics distinguishing keys from other values[23]. These keys can then be e-mailed at will to other addresses, as illustrated by the Sircam virus-worm[6]. An even easier attack in our context is the use of a simple debugger, allowing direct observation of the cryptographic keying material at the time of use. We seek cryptographic implementations providing protection in such extremely exposed contexts, which we call the *white-box attack context* or WBAC (see section 2). This paper discusses methods we have developed and deployed for doing so.

A natural question is: if an attacker has access to executing decryption software itself, why worry about preventing secret-key extraction, when the attacker could simply use the software and platform at hand to decrypt ciphertext or access plaintext? The answer is that our techniques are targeted mainly at software-based cryptographic content protection for Internet media, rather than at more traditional communications security. In such applications, the damage is relatively small if an attacker can make continued use of an already-compromised platform, but is prevented from extracting keying material which would allow software protection goals to be bypassed on other machines, or from publishing keying material effectively creating 'global cracks' which defeat security measures across large user-bases of installed software. Our solutions can also be combined with other software protection approaches, such as node-locking techniques tying the use of software to specific hardware devices.

RELEVANT APPLICATIONS. There are many applications for which our approach is clearly inappropriate in its current form, including applications in which symmetric keys are changed frequently (such as secure e-mail or typical file encryption applications which randomly select per-use keys). Our approach also results in far slower and bulkier code than conventional cryptographic implementations, ruling out other applications. Nonetheless, we have been surprised at the range of applications for which slow speed and large size can be accommodated, through a combination of careful selection of applications and crypto operations, and careful application engineering. For example, key management involving symmetric key-encrypting keys consumes only a negligible percentage of overall computation time, relative to bulk encryption, so use of white-box cryptography here has little impact on overall performance. Examples of relevant applications include copy protection for software, conditional access markets (e.g. set-top boxes for satellite pay-TV and video-on-demand), and applications requiring distribution control for protected content playback.

LIMITATIONS ON EXPECTED SECURITY. In the face of such an extreme threat environment, there are naturally limits to practically achievable security. A useful comparison is the commercial use of cryptographic implementations on smartcards: an inexpensive circuit mounted on plastic, with embedded secret keys, is widely distributed in essentially uncontrolled environments. This is hardly wise from a security standpoint, and successful attacks on smart cards are regularly reported. However for many applications smartcards provide a reasonable level of additional security at relatively low cost (*vs.* crypto hardware solutions), and a practical compromise among cost, convenience, and security. (Such trade-offs have long been recognized: e.g., see Cohen[9].) Our motivation is similar: we do not seek the ultimate level of security, on which a theoretical cryptographer might insist, but rather to provide an increased degree of protection given the constraints of a software-only solution and the hostile-host reality.

Theoretical feasibility of obfuscation. The theoretical literature on obfuscation and making software hard to reverse engineer appears somewhat contradictory. The NP-hardness results of Wang[24] and PSPACE-hardness results of Chow *et al.*[8] provide theoretical evidence that code transformations can massively increase the difficulty of reverse-engineering. In contrast, the impossibility results of Barak *et al.*[2], roughly speaking, show that a software *virtual black box generator*, which can protect *every* program's code from revealing more than the program's input-output behavior reveals, cannot exist. Of greater interest to us is whether this result applies to programs of practical interest, or whether cryptographic components based on widely-used families of block ciphers are programs for which such a virtual black box *can* be generated. Lacking answers to these questions, we pursue practical virtual boxes which are, so to speak, a usefully dark shade of gray.

Perhaps no perfect long-term defense against white-box attacks exists. Let us therefore clarify how our goals differ from typical cryptographic goals. We seek neither perfect protection nor long-term guarantees. Instead, we seek a practical level of protection in suitable applications, sufficient to make use of cryptography viable under the constraints of the WBAC. We believe the theoretical results above leave room for software protection of significant practical value.

Overview of white-box AES approach. This paper describes generation of WBAC-resistant AES components, with sub-components analogous to the encrypted-composed-functions of the recent literature[21, 22]. This converts AES-128 into a series of lookups in key-dependent tables. The key is hidden by (1) using tables for compositions rather than individual steps; (2) encoding these tables with random bijections; and (3) extending the cryptographic boundary beyond the crypto algorithm itself further out into the containing application, forcing attackers (reverse engineers) to understand significantly larger code segments to achieve their goals.

Organization of the paper. We discuss white-box cryptography and the white-box attack context (WBAC) in section 2, including WBAC-resistance at the cryptographic interface (section 2.2). Section 3 describes methods for hiding details of cryptographic operations, and their application to AES-128; size and performance are briefly addressed in subsection 3.6. Section 4 contains comments on security, including partial justification of our implementation by showing how removing portions of it significantly weakens our implementation. We offer concluding remarks in section 5.

2 White-Box Cryptography and Attack Context

Hosts may be untrusted for several reasons. As an example, software may be distributed to sites not controlled by the distributor — as is often the case with mobile code[21, 22], and digital rights management for software-based web distribution of books, journals, music, movies, news, or sports events. This may allow a *direct attack* by an otherwise legitimate end-user with hands-on access to the executing image of the target software. Hosts may also be rendered effectively hostile by viruses, worm programs, Trojan horses, and remote attacks on vulnerable protocols. This may involve an *indirect attack* by a remote attacker or automated attack tools, tricking users into opening malicious e-mail attachments, or exploiting latent software flaws such as buffer overflow vulnerabilities. Online shopping, Internet banking and stock trading software are all susceptible to these hazards. This leads us to what we call the *white-box attack context* (WBAC) and *white-box cryptography* (i.e., cryptography designed for WBAC-resistance). First we briefly review black-box and gray-box approaches.

2.1 Black-box, gray-box, and white-box

In traditional *black-box* models (as in: black-box testing), one is restricted to observing input-output or external behavior of software. In the cryptographic context, progressive levels of black-box attacks are known. Passive attacks are restricted to observation only (e.g. known-plaintext attacks, exhaustive key search); active attacks may involve direct interaction (e.g. chosen-plaintext attacks); adaptive attacks may involve interaction which depends upon the outcome of previous interactions (e.g. chosen plaintext-ciphertext attacks).

True black-box attacks are generic and do not rely on any knowledge of the internals of an algorithm. More advanced attacks appear to be 'black-box' at the time of execution, but in fact involve techniques specifically benefitting from extensive knowledge of the particular cryptographic algorithm. Examples include linear cryptanalysis and differential cryptanalysis (e.g. see Daemen and Rijmen[13]). These have some remnants of a *gray-box attack*. Other classes of cryptographic attacks that have a 'gray' aspect are so-called *side-channel attacks* and *partial-access attacks*, including timing analysis, power analysis, and fault analysis attacks [1, 3–5, 10, 11, 15, 16].

WHITE-BOX ATTACK CONTEXT. The *white-box attack context* (WBAC), in contrast, contemplates threats which are far more severe. It assumes that:

1. fully-privileged attack software shares a host with cryptographic software, having complete access to the implementation of algorithms;
2. dynamic execution (with instantiated cryptographic keys) can be observed;
3. internal details of cryptographic algorithms are both completely visible and alterable at will.

The attacker's objective is to extract the cryptographic key, e.g. for use on a standard implementation of the same algorithm on a different platform. WBAC includes the previously studied *malicious host* attack context[21, 22] and the hazards of malicious software unwittingly imported onto a host from the outside world (e.g., see Forrest *et al.*[14]). The black-box attack model and its gray-box variations are far too optimistic for software implementations to be deployed on untrusted hosts.

Security requirements for WBAC-resistance are greater than for resistance to gray-box attacks on smartcards. The WBAC assumes the attacker has complete access to the implementation, in which case typical smartcard defenses are insufficient, and typical smartcard attacks become moot. For example, an attacker has no interest in the power profile of computations if computations themselves are accessible, nor any need to introduce hardware faults if software execution can be modified at will. The smartcard experience teaches the following lesson: when the attacker has internal information about a cryptographic implementation, *choice of implementation is the sole remaining line of defense*[4, 5, 7, 10, 11, 17, 18]. Choice of implementation is precisely what is pursued in white-box cryptography.

On the other hand, implementations addressing the WBAC such as the white-box AES implementation proposed herein, are less constrained, in the sense that implementations may employ resources far more freely than in smartcard environments, including flexibility in processing power and memory. Among other available approaches, WBAC-resistant cryptographic components can also (and are often recommended to) employ a strategy of regular software updates or replacements (see related work by Jakobsson and Reiter[20]). When appropriate, such a design requires that protection need only withstand attacks for a limited period of time — thus counterbalancing the extreme threats it faces and the limits on the protection which it can afford.

2.2 WBAC-Resistance at the Cryptographic Interface

Any key input by a cryptographic implementation is completely exposed to privileged attack software sharing its host. There are two ways to avoid such exposure: (1) *Fixed key*: embed the key(s) in the implementation by partial evaluation with respect to the key(s), so that key input is unnecessary. This approach is quite adequate for many uses (but certainly not all — as discussed earlier), and is the one described in the remainder of the paper. Since such key-customized software implementations can be transmitted over bit-pipes, keys can still be changed with reasonable frequency. (2) *Dynamic key*: input encrypted and/or otherwise encoded key(s). This remains the subject of ongoing research.

A potential problem with the former is that a key-specific implementation could be extracted and used instead of its key, permitting an adversary to encrypt or decrypt any message for which the legitimate user had such capabilities. However, cryptography is seldom stand-alone; it is typically a component of a larger system. Our solution is thus to have this containing system provide the input to the cryptographic component in a manipulated or encoded form, for which the component is designed, but which an adversary will find difficult to remove. (In section 3.5, we consider a potential form for this manipulated data.) Further protection can then be provided by producing the output in another such form, so that a key-customized encryption function E_K is replaced by $E'_K = G \circ E_K \circ F^{-1}$ where F is an input encoding and G is an output encoding. This protects against key-extraction because there exists no combination of implementation components which computes E_K for any key K. By composing some prior and subsequent computational steps of the system[1] with the initial and final operations in the implementation of E'_K, we can further protect the E'_K implementation by ensuring that no precise boundary for it exists within its containing system. (Boundaries lie in the 'middle' of table lookups.)

An obvious question arises: *can use of F and G, as above, weaken the ordinary black-box security of E_K*? Although we have no proof, our belief is that such weakening is exceedingly unlikely, because F and G are chosen *randomly, without reference to K*. Intuitively, one would expect that if there is any significant probability that two key-independent, randomly chosen bijections F and G render $G \circ E_K \circ F^{-1}$ weaker than E_K, then cipher E must itself be seriously flawed for key K.

In the next section, we consider further general methods for performing WBAC-resistant cryptography, using AES as our example.

3 WBAC-Resistant AES

Since every cryptographic step might leak information, we make steps ambiguous by injecting randomness into each step, in a form not easily separated from the step itself. We have implemented an AES *implementation generator* program which takes an AES key and a source of randomness as input, and outputs a key-customized WBAC-resistant AES implementation. Implementations can be highly varied, allowing implementations to be diversified through both time and space.[2] The remainder of this section describes the sort of AES implementation our generator produces, using standard terminology[13, 19].

Broadly, our strategy is to compose each step in the AES algorithm with randomly chosen bijections. Since different steps, or different AES keys, may result in exactly the same composed functions depending on the choice of random bijections, the steps become ambiguous. To facilitate such encoding, we represent each AES component as a lookup table (an array of 2^m n-bit vectors, where m is the table's input width and n is its output width). Composition of lookup tables is straightforward and can be done by the implementation generator, so that the implementation itself contains only encoded lookup tables (with the functionalities shown in Fig. 1).

Taken to the absurd extreme, our implementation could consist of a single lookup table of about 5.4×10^{39} bytes representing the 128×128 bit AES bijection,

[1] Such as *xor*s, binary shifts, and bit-field extractions and insertions, which are conveniently representable as linear operations on vectors over GF(2).

[2] Using generated implementations which differ both among sites, and over time at a single site, helps foil pre-packaged attacks based on particular instances[9, 14].

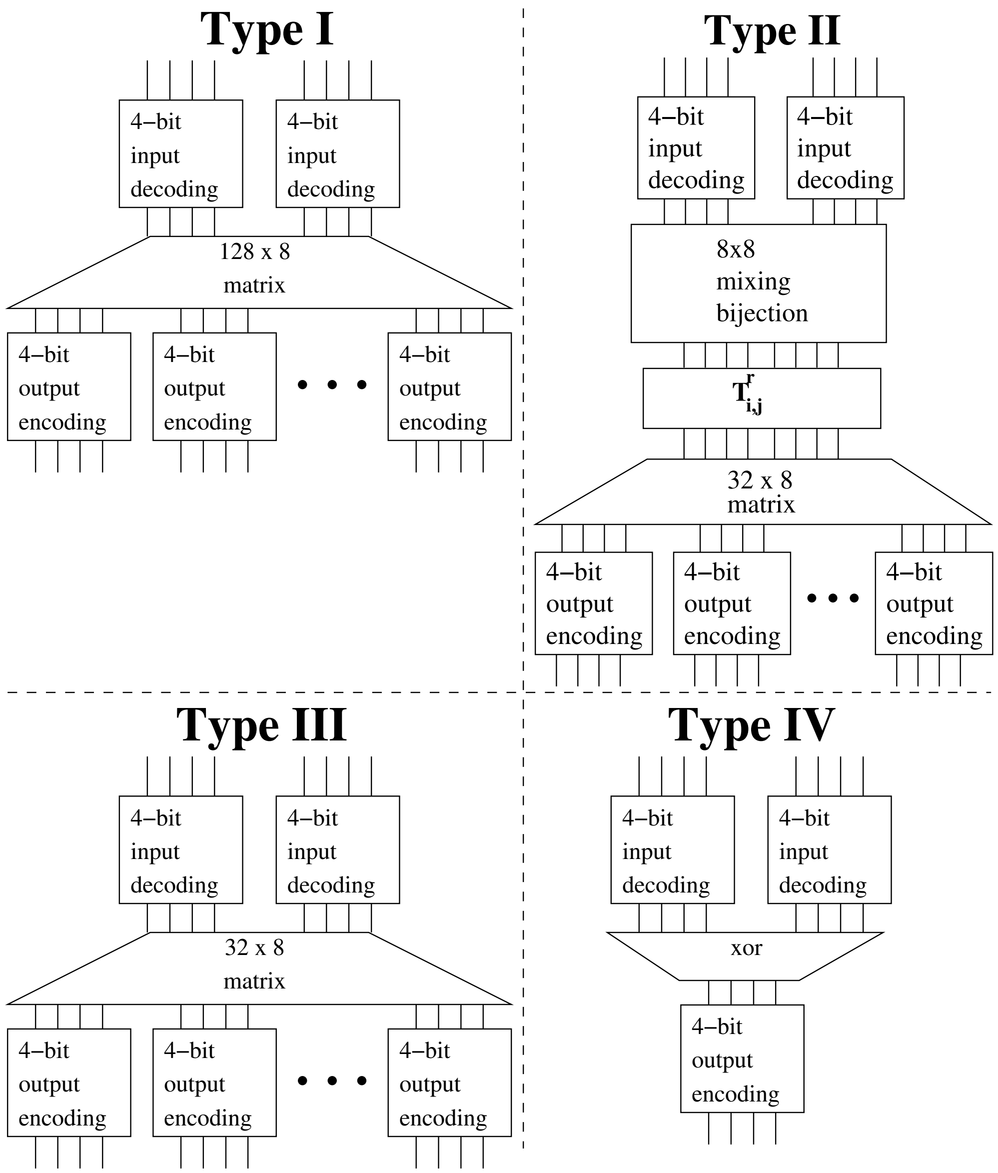

Fig. 1. Functionality of the four types of tables.

mapping plaintext to ciphertext for a particular AES-128 key. Such an implementation could only be attacked as a black box. We attempt to approximate this impossible ideal using tables of very much smaller size.

3.1 Partial Evaluation with Respect to the AES Key

As noted in subsection 2.2, the implementation generator produces key-customized instances of AES-128. We integrate the key into the *SubBytes* transformation by creating 160 (one per cell per round) 8×8 lookup tables, denoted $\mathbf{T}^r_{i,j}$:

$$\mathbf{T}^r_{i,j}(x) = S(x \oplus k^{r-1}_{i,j}) \qquad i = 0, \ldots, 3,\ j = 0, \ldots, 3,\ r = 1, \ldots, 9$$

where S represents the AES S-box, and $k^r_{i,j}$ is the byte of subkey in position i, j at round r. These "T-boxes" compose the *SubBytes* step with the previous round's *AddRoundKey* step.

The T-boxes in the tenth round must also absorb the post-whitening key. They are calculated as

$$\mathbf{T}^{10}_{i,j}(x) = S(x \oplus k^9_{i,j}) \oplus k^{10}_{sr(i,j)},$$

where $sr(i, j)$ denotes the new location of cell i, j after the *ShiftRows* step.

Of itself, partial evaluation provides no security: the key could easily be extracted from the T-boxes. Subsequent encoding renders such partial evaluation useful.

3.2 Simple, Concatenated, Networked, and Min-Loss Encoding

Let X be a transformation from m bits to n bits. To encode X, we choose an m-bit bijection F and an n-bit bijection G, and compute $X' = G \circ X \circ F^{-1}$ as the encoded version of X, so that both inputs and outputs are encoded. F is the *input encoding* and G is the *output encoding* for X'.

To avoid huge tables, we can construct an input or output encoding as the *concatenation* of smaller bijections. Consider bijections $F_1, F_2, \ldots, F_k$ of sizes $n_1, n_2, \ldots, n_k$, where $n_1 + n_2 + \ldots + n_k = n$. Using $\|$ for vector concatenation, we define the *function concatenation* $F_1 \| F_2 \| \ldots \| F_k$ as that bijection F such that $F(b) = F_1(b_1, \ldots, b_{n_1}) \| F_2(b_{n_1+1}, \ldots, b_{n_1+n_2}) \| \ldots \| F_k(b_{n_1+\ldots+n_{k-1}+1}, \ldots, b_n)$ for any n-bit vector $b = (b_1, b_2, \ldots, b_n)$. Plainly, $F^{-1} = F_1^{-1} \| F_2^{-1} \| \ldots \| F_k^{-1}$.

Since encodings are arbitrary, results are meaningful only if the output encoding of one step matches the input encoding of the next. For example, if step X is followed by step Y (i.e., if we compute $Y \circ X$), we encode them as

$$\begin{aligned} Y' \circ X' &= (H \circ Y \circ G^{-1}) \circ (G \circ X \circ F^{-1}) \\ &= H \circ (Y \circ X) \circ F^{-1} \end{aligned}$$

so that $Y \circ X$ is properly computed. The steps are separately represented as tables corresponding to Y' and X', so that F, G, and H are hidden.

If X is an $n \times m$ transformation (i.e. mapping m bits to n bits), $m \geq n$, it need only lose $m - n$ bits of input information. However, if there are only 2^{n-k} possible outputs, X drops $m - n + k$ bits: i.e., X loses k *excess* bits of input information. We can replace the table for X' with a table for X'' which does not lose these excess bits, and make the input encodings of successors to X'' interpret multiple encoded values as having the same decoded significance where appropriate. We can continue this process through the network, so that each lookup table carries as much information as its input and output widths allow. In this fashion, we hide such loss points in our lookup table network, delaying losses to later points in computation, and increasing the size of the attacker's search space for encodings. We call such concealment *min-loss encoding*.

The security offered by such encodings is the topic of section 4.

3.3 Non-Linear Encoding of Large Linear Transformations

Practical considerations limit our table sizes. This is not problematic for *SubBytes*, but becomes an issue for the 32-bit *MixColumns* step. To handle such a wide function, we use its linearity over GF(2): *MixColumns* is represented using four copies of a 32×32 matrix MC over GF(2).

We consider MC 'strips': we block MC into four 32×8 sections, MC_0, MC_1, MC_2, MC_3. Multiplication of a 32-bit vector $x = (x^0, x^1, \ldots, x^{31})$ by MC can be considered as four separate multiplications of the 8-bit vector $(x^{4i}, \ldots, x^{4i+7})$ by MC_i (resulting in four 32-bit vectors which we denote y_0, y_1, y_2, y_3), followed by three 32-bit additions over GF(2) (*xor*s) to compute the final 32-bit result, y. We further subdivide the additions into twenty-four 4-bit *xor*s with appropriate concatenation (e.g. $((y_0^0, y_0^1, y_0^2, y_0^3) + (y_1^0, y_1^1, y_1^2, y_1^3)) \| ((y_0^4, y_0^5, y_0^6, y_0^7) + (y_1^4, y_1^5, y_1^6, y_1^7)) \| \ldots)$. (See Fig. 2.) Using these strips and subdivided *xor*s, each such step can be represented using a small lookup table. In particular, computation of y_0, y_1, y_2, y_3 can be performed with four 8×32 tables, which we denote Ty_0, Ty_1, Ty_2, Ty_3, while the 4-bit *xor*s become twenty-four 8×4 tables.

We note that the *xor* tables, regardless of the order in which they are used, take in 4 bits from each of two previous computations. Since the output encodings of these computations must be matched by the input encodings for the *xor* tables, we require that all encodings be 4-bit bijections.[3] In particular, we use concatenated encodings both for the 32-bit output encodings to Ty_i and the 8-bit input encodings to the *xor* tables.

The need to use 4-bit encodings imposes a similar limitation on the input encodings for the T-boxes, since the *MixColumns* outputs feed into the T-box step. In theory, we could use true 8-bit output encodings for the T-boxes, but by composing the T-boxes with the Ty_i's, an adversary could easily cancel these encodings out. Therefore, we perform this composition ourselves, creating new Ty_i's which compute the *SubBytes* and *AddRoundKey* transformations as well

[3] We can ignore linearity: there are $2^n!$ n-bit bijections, of which $2^n \prod_{i=0}^{n-1}(2^n - 2^i)$ are affine, so there are $16! \approx 2.09 \times 10^{13}$ 4-bit bijections of which 322,560 (less than .000002%) are affine.

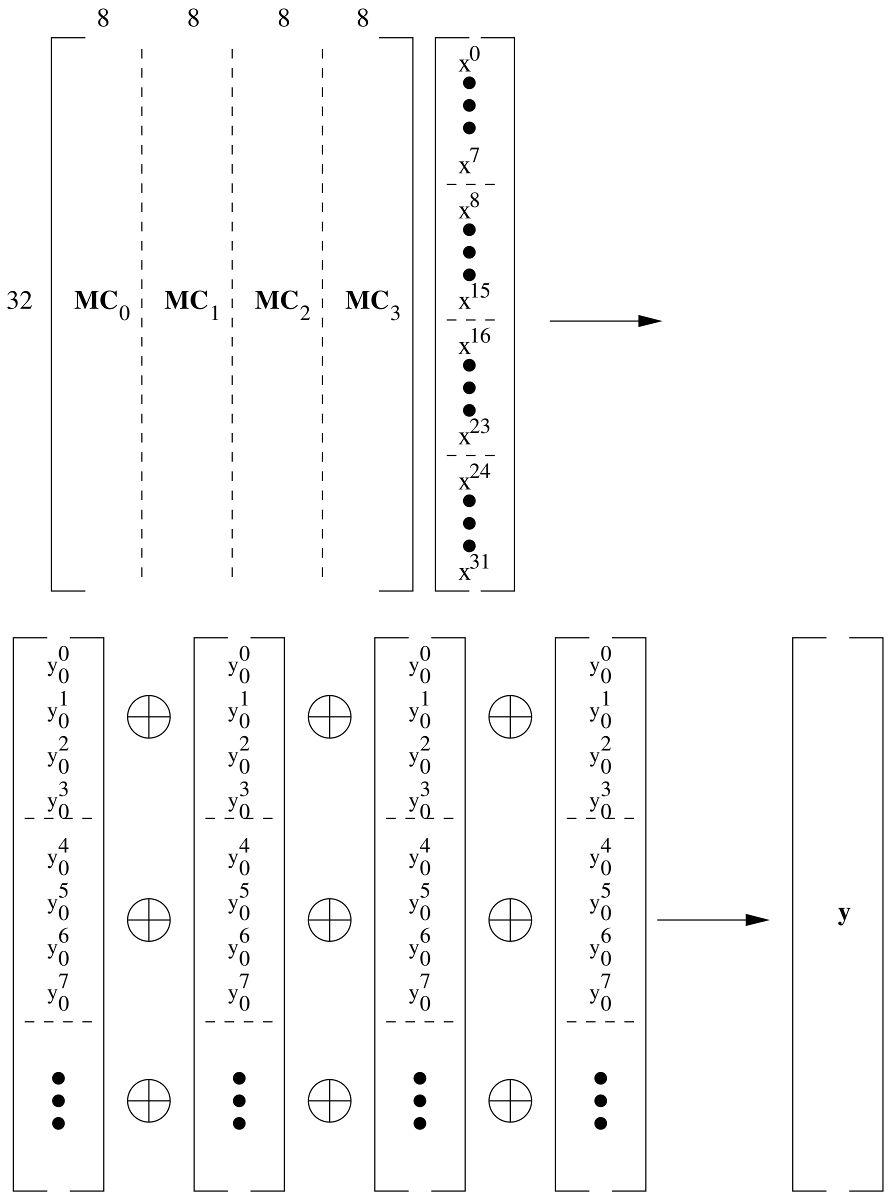

Fig. 2. Multiplication by MC

as part of *MixColumns*. This saves both space (to store the T-boxes) and time (to perform the table lookups).

With this composition, our implementation now resembles the implementation in section 5 of the Rijndael proposal[12], with the added benefit of WBAC-protection (and the added detriment of being much larger and slower).

3.4 Mixing Bijections

So far, we have used only encodings which are non-linear with high probability. Considering encodings as a encipherments of intermediate values in AES, such encodings are confusion steps. We now introduce linear transformations as diffusion steps, to further disguise the underlying operations.

We use the term *mixing bijection* to describe a linear bijection, especially one used in the above sense. Generally, we think of mixing bijections as matrices over GF(2), and use corresponding terminology.

We use 8×8 mixing bijections to diffuse the inputs of the T-boxes (technically, the inputs to the combined T-box/*MixColumns* step). By combining the inversion of the mixing bijections with MC through ordinary matrix multiplication, we diffuse the inversion step over several table lookups, making it hard to remove.

Moreover, we further multiply MC by a 32×32 mixing bijection, MB. We choose MB as a non-singular matrix with 4×4 submatrices of full rank (see Xiao and Zhou[25] for a way to generate such matrices), ensuring that the encoded components of MB' will carry maximum information and maximizing information diffusion among those components.

Of course, we must invert MB. This requires an extra set of tables, similar in form to the ones used to calculate MC, in order to calculate MB^{-1}. While this is detrimental to size and speed, it appears to be outweighed by MB's security benefits.

Thus, a round of AES is implemented by encoding four pairs of two tables. The first table combines the T-boxes with input mixing bijections, MC, and MB. The second combines MB^{-1} with the inverse of the input mixing bijections for the subsequent round's T-boxes.

3.5 Input and Output Data Manipulations

As described in subsection 2.2, our implementation takes input in a manipulated form, and produces output in a differently manipulated form, thereby making the WBAC-resistant AES difficult to separate from its containing application. In the previous sections, we have described techniques to handle both small non-linear steps and large linear steps securely, and so our manipulation can combine linear and non-linear components in a variety of ways.

The idea is to have the first steps of the implementation undo the manipulation which is performed elsewhere in the program. Thus, while it is more straightforward to describe what these first steps of AES might look like, keep in mind that it is actually the inverse of steps done earlier in the program, and similarly the last steps will be undone at a later stage. The net result is an effectively standard, and WBAC-resistant, AES computation, obtained by embedding a non-standard AES in a correspondingly non-standard usage context.

The following is one suggestion for data manipulation. We insert a 128×128 mixing bijection *IDM* prior to the first T-box calculation, and another 128×128 mixing bijection *ODM* after the last T-box calculation. We compose *IDM* with the inverted input mixing bijections for $\mathbf{T}^1$. We can then block and encode these matrices in the usual way, where we ensure that the output encodings on the *IDM* addition tables are matched by the input encodings of the first round transformation. (This also determines the input encodings for the prior manipulation, which are matched by the output encodings on *IDM* addition.) We also compose $\mathbf{T}^{10}$ with the tables for computing the *ODM* strips.

3.6 Size and Performance

We now have an implementation of AES which consists entirely of table lookups. The only operation left unchanged is *ShiftRows*, which is performed simply by providing the appropriate shifted inputs to the tables. We begin with a mixing bijection over the full state, then perform a series of lookups computing the eight matrices necessary for the round transformation. We end with a combined final round and output mixing bijection. Every step is input- and output-encoded using simple or concatenated 4-bit encodings in a networked fashion, with min-loss encoding employed to hide excess information loss for components suffering from this weakness prior to encoding.

The total size of the lookup tables is 770048 bytes[4], and there are 3104 lookups during each execution[5]. A fair comparison is the AES implementation of Daemen and Rijmen[12], which requires 4352 bytes for lookup tables, and requires approximately 300 operations (lookups and *xor*s) in total. We feel that WBAC-protection is vital in potentially hostile environments, but it does come at quite a substantial price. Thus careful choices must be made as to where, and how, to employ white-box AES (see section 1).

4 Preliminary Security Comments

4.1 White-Box Analogs of Keyspace

Keyspace provides an upper bound on the security of a cryptographic algorithm. Analogously, if encodings 'encrypt' implementation steps, we can count the pos-

[4] T-box/*MixColumns* strips: $9 \times 2 \times 4 \times 4 = 288$ 8×32 tables = 294912 bytes, *MixColumns* additions: $9 \times 2 \times 4 \times 8 \times 3 = 1728$ 8×4 tables = 221184 bytes, *IDM* and T-box/*ODM* strips: $2 \times 16 = 32$ 8×128 tables = 131072 bytes, *IDM* and *ODM* additions: $2 \times 32 \times 15 = 960$ 8×4 tables = 122880 bytes.

[5] T-box/*MixColumns* strips: 288 lookups, *MixColumns* additions: 1728 lookups, *IDM* and T-box/*ODM* strips: $32 \times 4 = 128$ lookups, *IDM* and *ODM* additions: 960 lookups.

sible encoded steps. We call this metric *white-box diversity.* The white-box diversity of a given table type (see Fig. 1) counts how many distinct *constructions* exist for a table of that type. (This exceeds the number of distinct *tables.*)

A far more important metric is the *white-box ambiguity* of a table, which counts the number of distinct *constructions* which produce *exactly the same table* of that type.

White-box diversity measures variability among implementations, useful in foiling pre-packaged attacks[14], whereas *white-box ambiguity* is a measure of the number of alternative interpretations or meanings of a specific table, among which an attacker must disambiguate in cracking one of our AES instances.

We will assume that the encodings are random and independent, except for the ones which are inverses of each other. (Choosing the encodings is one-time per implementation work, and thus quality should outweigh efficiency in the selection of a source of randomness.)

White-Box Diversity There are four table types in our implementation, with the functionality shown in Fig. 1. Type I tables compute the strips in *IDM* and *ODM*, and represent a pair of 4-bit decodings, followed by an 128×8 matrix, then thirty-two 4-bit encodings. Type II tables compute the strips in the first half of a round transformation. These represent two 4-bit decodings, then an 8×8 mixing bijection, followed by a T-box, then an 32×8 matrix, and finally eight 4-bit encodings. Type III tables compute the strips in the second half of a round transformation, representing two 4-bit decodings, followed by a 32×8 matrix, and then eight 4-bit encodings. Lastly, type IV tables compute the GF(2) additions, representing two 4-bit decodings, the known *xor* operation, and a 4-bit output encodings.

The *white-box diversity* for the various kinds of tables is:

- Type I - $(16!)^2 \times 20160^{64} \times (16!)^{32} \approx 2^{2419.7}$, where there are 20160 nonsingular 4×4 matrices over GF(2).
- Type II - $(16!)^2 \times 256 \times 2^{62.2} \times 2^{256} \times (16!)^8 \approx 2^{768.7}$. The number of 8×8 mixing bijections is roughly $2^{62.2}$. The actual number of type II tables is slightly lower, because not every 8×32 matrix can be produced as a product of *MC* and mixing bijections.
- Type III - $(16!)^2 \times 2^{256} \times (16!)^8 \approx 2^{698.5}$.
- Type IV - $(16!)^2 \times 16! \approx 2^{132.8}$.

White-Box Ambiguity The *white-box ambiguity* for a given table is the number of distinct constructions which could produce exactly that same table. By this measure, type IV tables are by far the weakest point.

Finding a rigorous way to compute white-box ambiguity remains a central open problem. Meanwhile, we have made estimates by extrapolating from smaller tables preserving our basic structure. (Also, we are assuming that the constructions are equiprobable, which is only approximately true.)

For type I tables, we considered both 4×4 and 6×4 matrices, with 2-bit input/output encodings. As we see later, the white-box ambiguity appears to be connected with the ranks of the matrix blocks, which would enforce the need to consider larger matrices (or to use min-loss encoding: see subsection 3.2). However, by construction, all matrix blocks for type I tables have full rank, which results in a more regular behavior, hopefully captured using 2×2 blocks.

We enumerated by simply exhausting over every possibility, counting distinct results. We conclude: given a type I table, its two input encodings, and one of the of the two 4×4 blocks in each set of four rows, are unconstrained, but these choices uniquely determine the other block in each set of rows and all of the output encodings. Therefore, the number of components which can generate a given type I table is $(16!)^2 \times 20160^{32} \approx 2^{546.1}$.

Type II and III tables are more complex: the matrix blocks may not have full rank. We can ameliorate this problem using min-loss encoding, but to ensure conservative estimation, we ignore such encodings.

With this modification, the rank of each block can be directly determined from the table. Consider the components needed to compute a single, fixed nibble of each entry in a type III table. We have two 4-bit input decodings, each of which feeds into a 4×4 block, and finally a 4-bit output encoding. If we consider the outputs in a 16×16 array, we find the effect of the first input decoding is to permute the rows of the array, while the second input decoding permutes the columns of the array. Likewise, the rank of the first block can be determined by counting the number of distinct entries in any column, and the rank of the second block by counting distinct entries in a row, where the rank is the base-2 logarithm of the resulting count.

The simplest sub-case to analyze is one where both blocks have rank 0. Here, the resulting table reveals no information about the input encodings (as any row and column permutation of a single entry table will look identical) and also reveals no information about the output encoding of any value except 0. Therefore, the number of components which could have produced such a table is $(16!)^2 \times 15! \approx 2^{128.8}$. Of course, it is entirely possible that the other blocks in the 32×8 matrix reveal more information about the encodings.

For full-rank blocks, the blocks uniquely determine the output encodings. Since there are at most 20160^2 possible such blocks, an upper bound for the number of components which could produce a given table is $(16!)^2 \times 20160^2 \approx 2^{117}$. In other cases, we could construct similar upper bounds (taking into account parts of the output encoding that cannot be determined).

The type IV tables have the smallest white-box ambiguity by far. Given one input decoding and the value to which 0 decodes for the other input decoding, we uniquely determine the remaining encodings. Therefore, the number of components which generate a given type IV table is $16! \times 16 \approx 2^{48.2}$. Exhaustion of such tables is certainly feasible, which is a threat if the attacker can find a way to decide whether an alternative is correct. It is not yet clear how to do this.

However, assume that an efficient means of performing this disambiguation on type IV tables exists. We still argue that: (1) our defense is intended only for a period of time, and (2) malware performing work anywhere near 2^{48} steps (about three days work at 1 GHz) cannot be stealthy. So for *malware*-resistance, it is likely to be adequate. Even a *malicious host* would need significant resources to perform a crack: and each crack would be only good for an interval (say, for part of a book or movie).

Of course, keyspace-like security measures are appropriate only in the absence of efficient attacks which bypass much of the search space. We now consider what form such an attack might take.

4.2 A Generic Square-Like Attack

We first describe a generic attack which assumes that the input decodings to a set of four type II tables, corresponding to a round transformation for a single column, have been removed. Later, we will show how this assumption can be realized for certain weakened variants.

We can send chosen texts through these tables, in the sense that we can consider the inputs to be direct (unencoded) inputs to the *AddRoundKey* and *SubBytes* steps. However, we cannot consider the outputs in the same way, because of the *MB* transformation, as well as the output encoding.

Consider the value of a cell after the two-part round transformation. Its encoding consists of an 8-bit mixing bijection and two concatenated 4-bit random bijections. Such an encoding is local to the cell, and therefore has an important property — *two texts which have the same encoded value in that cell, have the same unencoded value in that cell.* In other words, while we cannot in general determine the unencoded *xor* difference of two texts, we *can* identify when it is zero.

Our goal is to find two 32-bit texts which have a non-zero input difference in each cell, and have a zero output difference in all but one cell. This can be recognized as the strategy in the first round of the Square attack on AES[12]. Suppose we find such texts, denoted $w = (w_0, w_1, w_2, w_3)$ and $x = (x_0, x_1, x_2, x_3)$. Let the key which is embedded in the type II tables be $k = (k_0, k_1, k_2, k_3)$. Let $y = (y_0, y_1, y_2, y_3)$ be the mod 2 difference between the two texts after the *SubBytes* transformation, i.e.

$$y_i = S(w_i \oplus k_i) \oplus S(x_i \oplus k_i).$$

Then we have

$$\begin{aligned}
\mathtt{01}(y_0) \oplus \mathtt{02}(y_1) \oplus \mathtt{03}(y_2) \oplus \mathtt{01}(y_3) &= \mathtt{00} \\
\mathtt{01}(y_0) \oplus \mathtt{01}(y_1) \oplus \mathtt{02}(y_2) \oplus \mathtt{03}(y_3) &= \mathtt{00} \\
\mathtt{03}(y_0) \oplus \mathtt{01}(y_1) \oplus \mathtt{01}(y_2) \oplus \mathtt{02}(y_3) &= \mathtt{00}
\end{aligned}$$

or some variant thereof, depending on which cells match and which cell differs. The above system has the solution

$$\begin{aligned}
y_0 &= \mathtt{ec}(y_3) \\
y_1 &= \mathtt{9a}(y_3) \\
y_2 &= \mathtt{b7}(y_3)
\end{aligned}$$

Thus, the choice-count for k has been significantly reduced, to an upper bound of 256. At this point, we could simply exhaust to determine k. Alternatively, we could make a guess for k and choose texts as in the Square attack, and look for three-cell collisions as above.

Based on collision probability (about 2^{-22}), expected work to find an entire round key is about 2^{13} one-round encryptions for this weakened variant.

4.3 Applying the Square-Like Attack to a Weakened Variant

Let us consider the variant of our WBAC-resistant AES without input/output data manipulation. As well as vulnerability to direct capture during transmission, such a variant has no input encoding for the first set of Type II tables, and is vulnerable to the attack in subsection 4.2.

Unless more efficient attacks are found, removal of data manipulation requires exhaustion of a type I table. Their high white-box ambiguity makes this infeasible.

The mixing bijections serve a critical role in the intermediate steps of the cipher. Consider an implementation in which the T-boxes were not pre-mixed, and in which *MB* was not used (meaning that it would not have to be inverted). Such an implementation would have modified type II tables (and no type III tables), consisting of two 4-bit decodings, followed by a T-box, then a known 32×8 matrix, and finally eight 4-bit encodings. In particular, we note that two of the 8×8 blocks are simply the identity matrix (corresponding to multiplication by 01), and thus by considering only a portion of the table output, we can ignore the matrix altogether.

Partial evaluation of the key and the 4-bit input/output encodings, do not alter the distributions of nibbles in the rows and columns of the AES S-box. E.g., the high-order nibble of column 3 has six values appearing twice, four values appearing once, and six values which do not appear, and is the only column with such a distribution. Arranging the appropriate portion (corresponding to the identity matrix) of the modified type II table, we find an identical distribution in the high-order nibble. Continuing in this way, we find a 1-1 mapping between the rows and columns of the S-box and T-box.

Since the key rearranges the rows and columns of the S-box in an unknown fashion, the above mapping only determines the input encodings up to a guess of key. However, every choice of key yields an identical result for the *output* encoding (i.e., the output encoding is key-independent). Thus, we can uniquely determine a set of output encodings, which are inverted by the input decodings of the next round's tables. We have thus met the conditions necessary to launch the Square-like attack.

The mixing bijections thwart this attack by diffusing the distribution over the nibbles and over the set of T-boxes, and min-loss encoding foils attacks based on efficiently guessing mixing bijections by locating tables with excess bit-loss.

5 Conclusions and Future Work

We consider the white-box attack context (WBAC), reflecting both the capabilities of an adversary who can introduce malicious code, and the demands on mobile or commodity software cryptography due to the economics and logistics of the Internet. Traditional implementations of cryptographic algorithms, including AES, are completely WBAC-insecure. As with smartcards, the security of a cipher is as dependent on its environment and implementation as on its mathematical underpinnings.

As a proposal for pragmatically acceptable WBAC-resistance, we present a new way of implementing AES using lookup tables representing encoded compositions. Such implementations are far larger and slower than reference code, but arguably allow cryptographic computation to take place with a useful degree of security, for a period of time, even in the presence of an adversary who can observe and modify every step. (Aside from white-box strength, generated AES instances provide a diversity defense against pre-packaged attacks on particular instances of executable software.)

Combinatoric properties of white-box attacks are very incompletely understood. In this paper, we consider the difficulty of component exhaustion and a Square-like attack. The issues of attacks on multiple components at once, or on multiple implementations sharing a key, remain to be thoroughly investigated. For example, white-box ambiguity for a *sub-network* of tables in our implementation may differ depending on where its boundaries lie.

While we have found our generated AES instances viable in commercial practice, the large size and low speed of our implementations certainly limits general applicability. Efficiency improvements would be most valuable.

References

1. Ross J. Anderson, Markus G. Kuhn, *Low Cost Attacks on Tamper-Resistant Devices*, pp. 125-136, 5th International Workshop on Security Protocols (LNCS 1361), Springer-Verlag, 1997.
2. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, Ke Yang, *On the (Im)possibility of Obfuscating Programs*, pp. 1-18, Advances in Cryptology – Crypto 2001 (LNCS 2139) Springer-Verlag, 2001.
3. Eli Biham, Adi Shamir, *Differential Fault Analysis of Secret Key Cryptosystems*, pp. 513-525, Advances in Cryptology – Crypto '97 (LNCS 1294), Springer-Verlag, 1997. *Revised*: Technion - Computer Science Department - Technical Report CS0910-revised, 1997.
4. Eli Biham, Adi Shamir, *Power Analysis of the Key Scheduling of the AES Candidates*, presented at the Second AES Candidate Conference, Rome, Italy, March 22-23, 1999.
5. Dan Boneh, Richard A. DeMillo, Richard J. Lipton, *On the Importance of Eliminating Errors in Cryptographic Computations*, Journal of Cryptology 14(2), pp. 101-119, 2001.
6. CERT Advisory CA-2001-22 W32/Sircam Malicious Code, 25 July 2001 (revised 23 August 2001), `http://www.cert.org/advisories/CA-2001-22.html`.

7. S. Chari, C. Jutla, J.R. Rao, P. Rohatgi, *A Cautionary Note Regarding Evaluation of* AES *Candidates on Smart-Cards*, presented at the Second AES Candidate Conference, Rome, Italy, March 22-23, 1999.
8. S. Chow, Y. Gu, H. Johnson, V.A. Zakharov, *An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs*, pp.144-155, Proceedings of ISC 2001 – Information Security, 4th International Conference (Malaga, Spain, 1-3 October 2001), LNCS 2200, Springer-Verlag, 2001.
9. F. Cohen, *Operating System Protection Through Program Evolution*, Computers and Security vol.12 no.6 (1 Oct. 1993), pp.565-584.
10. Joan Daemen, Vincent Rijmen, *Resistance Against Implementation Attacks: A Comparative Study of the* AES *proposals*, presented at the Second AES Candidate Conference, Rome, Italy, March 22-23, 1999.
11. Joan Daemen, Michael Peeters, Gilles van Assche, *Bitslice Ciphers and Power Analysis Attacks*, pp. 134-149, 7th International Workshop on Fast Software Encryption – FSE 2000 (LNCS 1978), Springer-Verlag, 2000.
12. Joan Daemen, Vincent Rijmen, AES *Proposal: Rijndael* `http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf`, 1999.
13. Joan Daemen, Vincent Rijmen, *The Design of Rijndael: AES – The Advanced Encryption Standard*, Springer, 2001.
14. Stephanie Forrest, Anil Somayaji, David H. Ackley, *Building Diverse Computer Systems*, pp. 67-72, Proceedings of the 6th Workshop on Hot Topics in Operating Systems, IEEE Computer Society Press, 1997.
15. Paul C. Kocher, *Timing Attacks against Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, pp. 104-113, Advances in Cryptology – Crypto '96 (LNCS 1109), Springer-Verlag, 1996.
16. Paul Kocher, Joshua Jaffe, Benjamin Jun, *Differential Power Analysis*, pp. 388-397, Advances in Cryptology – Crypto '99 (LNCS 1666), Springer-Verlag, 1999.
17. Oliver Kömmerling, Markus G. Kuhn, *Design Principles for Tamper-Resistant Smartcard Processors*, pp. 9-20, Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99), USENIX Association, ISBN 1-880446-34-0, 1999.
18. National Institute of Standards and Technology, *Round 2 Discussion Issues for the* AES *Development Effort*, November 1, 1999. `http://csrc.nist.gov/encryption/aes/round2/Round2WhitePaper.htm`.
19. National Institute of Standards and Technology (NIST), *Advanced Encryption Standard* (AES), FIPS Publication 197, 26 Nov. 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`
20. Markus Jakobsson, Michael K. Reiter, *Discouraging Software Piracy Using Software Aging*, pp.1-12, Security and Privacy in Digital Rights Management – ACM CCS-8 Workshop DRM 2001 (LNCS 2320), Springer-Verlag, 2002.
21. Tomas Sander, Christian F. Tschudin, *Towards Mobile Cryptography*, pp. 215-224, Proceedings of the 1998 IEEE Symposium on Security and Privacy.
22. Tomas Sander, Christian F. Tschudin, *Protecting Mobile Agents Against Malicious Hosts*, pp.44-60, Mobile Agent Security (LNCS 1419), Springer-Verlag, 1998.
23. Nicko van Someren, Adi Shamir, *Playing Hide and Seek with Keys*, pp. 118-124, Financial Cryptography '99 (LNCS 1648), Springer-Verlag, 1999.
24. C. Wang, *A Security Architecture for Survivability Mechanisms*, Doctoral thesis, University of Virginia, October 2000.
25. James Xiao, Yongxin Zhou, *Generating Large Non-Singular Matrices over an Arbitrary Field with Blocks of Full Rank*, Cryptology ePrint Archive (`http://eprint.iacr.org`), no. 2002/096.