White-Box Cryptography and an AES Implementation

Stanley Chow, Philip Eisen, Harold Johnson, and Paul C. Van Oorschot

Cloakware Corporation, Ottawa, Canada
{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. We partially justify our AES implementation, and motivate its design, by showing how removal of parts of the recommended implementation allows specified attacks, including one utilizing a pattern in the AES *SubBytes* table.

1 Introduction and Overview

There has been tremendous progress in the uptake of cryptography within computer and network 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. Among several reasons is the 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 [15], 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 [26]. These keys can then be e-mailed at will to other addresses, as illustrated by the Sircam virus-worm [7]. An even easier attack in our context is to use a simple debugger to directly observe 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 (§2). This paper discusses methods developed and deployed for doing so.

A natural question is: if an attacker has access to executing decryption software, why worry about secret-key extraction — the attacker could simply use the software at hand to decrypt ciphertext. Protection methods in §2.2 make this hard, and even if such protections were compromised, our techniques are targeted at applications such as software-based cryptographic content protection for Internet media, rather than more traditional communications security. In such applications, the damage is often relatively small if an attacker can make continued use of an already-compromised platform, but cannot extract keying material allowing software protection goals to be bypassed on other machines, or publish keys or software sub-components allowing 'global cracks' to 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 software use 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. In *all* environments, however, our white-box implementations provide *at least* as much security as a typical black-box implementation (see §2.1). Moreover on hostile platforms, for conventional (black-box) implementations of even the theoretically strongest possible algorithms, typically-claimed "crypto" levels of security (e.g. 2^{128} operations, 10^{20} years, etc.) fall essentially to zero (0) as the key is directly observable by an attacker. Therefore when considering white-box 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 added security at relatively low cost (*vs.* crypto hardware solutions), and a practical compromise among cost, convenience, and security. (Such tradeoffs have long been recognized: e.g., see Cohen [11].) 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 software obfuscation appears somewhat contradictory. The NP-hardness results of Wang [27] and PSPACE-hardness results of Chow *et al.* [10] provide theoretical evidence that code transformations can massively increase the difficulty of reverse-engineering. In contrast, the impossibility results of Barak *et al.* [3], essentially 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.

It seems safe to conjecture that no perfect long-term defense against white-box attacks exists. We therefore distinguish our goals from typical cryptographic goals: we seek neither perfect protection nor long-term guarantees, but rather a practical level of protection in suitable applications, sufficient to make use of cryptography viable under the constraints of the WBAC. The theoretical results cited above leave room for software protection of significant practical value.

Overview of White-Box AES Approach. This paper describes generation and composition of WBAC-resistant AES components, analogous in some ways to the encrypted-composed-function approach (see Sander and Tschudin [24,25], and Algesheimer et al. [1] for an update and comments). 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 algorithm itself further out into the containing application, forcing attackers (reverse engineers) to understand significantly larger code segments.

Organization. We discuss white-box cryptography and the white-box attack context (WBAC) in §2. §3 outlines a strategy and provides details for generating white-box AES implementations, including key-embedding, construction and composition of lookup tables implementing AES steps, insertion of random bijections, and size-performance issues. §4 includes security comments on white-box AES, with partial justification showing how removing portions of our design allows specified attacks. One of the attacks described in §4.4 employs patterns in the AES *SubBytes* table which may be of independent interest. Concluding remarks are in §5.

2 White-Box Cryptography and Attack Context

Hosts may be untrusted for various reasons, including the economics and logistics of the Internet. Often software is distributed to servers where access control enforcement cannot be guaranteed, or sites beyond the control of the distributor. This happens for mobile code [24,25], and where software tries to constrain what end-users may do with content — as in digital rights management for software-based web distribution of books, periodicals, 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 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 related approaches.

2.1 Black-Box, Gray-Box, and White-Box Attack Contexts

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 (e.g. known-plaintext attacks, exhaustive key search) are restricted to observation only; active attacks (e.g. chosen-plaintext attacks) may involve direct interaction; adaptive attacks (e.g. chosen plaintext-ciphertext attacks) may involve interaction which depends upon the outcome of previous interactions.

True black-box attacks are generic and do not rely on knowing internal details of an algorithm. More advanced attacks appear to be 'black-box' at the time of execution, but in fact exploit knowledge of an algorithm's internal details. Examples include linear and differential cryptanalysis (e.g. see [15]). These have remnants of a *gray-box attack*. Other classes of cryptographic attacks that have a 'gray' aspect are so-called *side-channel attacks* or *partial-access attacks*, including timing, power, and fault analysis attacks [2,4,5,6,12,13,18,19]. These clearly illustrate that even partial access or visibility into the inner workings, side-effects, or execution of an algorithm can greatly weaken security.

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 algorithm details are 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 [24,25] and the hazards of unwittingly importing malicious software (e.g., see Forrest *et al.* [16]). The black-box attack model and its gray-box variations are far too optimistic for software implementations 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, rendering typical smartcard defenses insufficient, and typical smartcard attacks obsolete. 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 highlights that when the attacker has internal information about a cryptographic implementation, *choice of implementation is the sole remaining line of defense* [5,6,8,12,13,20,22] — and this is precisely what white-box cryptography pursues.

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 (cf. Jakobsson and Reiter [17]). When appropriate, such a design requires that protection need only withstand attacks for a limited period of time — thus counterbalancing the extreme threats faced, and the resulting limits on the level of protection possible.

2.2 White-Box Attack-Resistance at the Cryptographic Interface

Any key input by a cryptographic implementation is completely exposed to privileged attack software sharing its host. Two ways to avoid this follow. (1) *Dynamic key approach*: input encrypted and/or otherwise encoded key(s); this is the subject of ongoing white-box research. (2) *Fixed key approach*: embed the key(s) in the implementation by partial evaluation with respect to the key(s), so that key input is unnecessary. Since such key-customized software implementations can be transmitted wherever bits can, keys may still be changed with reasonable frequency. This approach is appropriate in selected applications (see §1), and is the subject of the remainder of this paper. We begin with a definition.

Definition 1 (encoding) *Let X be a transformation from m to n bits. Choose an m-bit bijection F and an n-bit bijection G. Call $X' = G \circ X \circ F^{-1}$ an* encoded *version of X. F is an* input encoding *and G is an* output encoding.

A potential problem with the fixed-key approach is that a key-specific implementation might 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 to have this containing system provide the input to the cryptographic component in a manipulated or encoded form (see §3.4 for AES-specific details) for which the component is designed, but which an adversary will find difficult to remove. Further protection is provided by producing the output in another such form, replacing a key-customized encryption function E_K by the composition $E'_K = G \circ E_K \circ F^{-1}$. Here F and G are (external) input and output encodings, both randomly selected bijections independent of K. E'_K no longer corresponds to encryption with key K; this protects against key-extraction as no combination of implementation components computes E_K in isolation.

The recommended implementation makes G^{-1} and F available only on a computing platform separate from the platform running E'_K (the attack platform). Additionally, when possible, some prior and subsequent computational steps (e.g. *xor*s, binary shifts, and bit-field extractions and insertions, conveniently representable as linear operations on vectors over GF(2)) of the host system are composed with the initial and final operations implementing E'_K. This adds further protection by arranging that no precise boundary for E'_K exists within the containing system (boundaries lie in the 'middle' of compositions represented as table lookups).

2.3 Concatenated Encoding and Networked Encoding

In what follows, to avoid huge tables, we can construct an input or output encoding as the *concatenation* of smaller bijections. Consider bijections F_i of size n_i, where $n_1 + n_2 + \ldots + n_k = n$. Let $\|$ denote vector concatenation.

Definition 2 *The* function concatenation $F_1 \| F_2 \| \ldots \| F_k$ *is the 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,* $F^{-1} = F_1^{-1} \| F_2^{-1} \| \ldots \| F_k^{-1}$.

Encodings generally make use of random bijections (cf. Definition 1). To make results meaningful, the output encoding of one encoded transformation will generally be matched with the input encoding of the next, as follows.

Definition 3 *A* networked encoding *for computing* $Y \circ X$ *(i.e. transformation* X *followed by transformation* Y*) is an encoding of the form*

$$Y' \circ X' \;=\; (H \circ Y \circ G^{-1}) \circ (G \circ X \circ F^{-1}) \;=\; H \circ (Y \circ X) \circ F^{-1}\ .$$

Note that internally, $Y \circ X$ is computed. By separately representing the steps as tables corresponding to Y' and X', the bijections F, G, and H may be hidden.

3 Constructing White-Box AES Implementations

In the white-box attack context, each cryptographic step might leak information. Our strategy is to break each AES round into a number of steps, and compose the steps after inserting randomly chosen bijections serving as *internal* encodings (in addition to the *external* encodings enveloping an overall cipher, as in $G \circ E_K \circ F^{-1}$ above). This randomness, intended to be difficult to separate from the step itself, introduces 'ambiguity' (see §4.2) — many possible $\langle$*key,bijection*$\rangle$ combinations might correspond to the same encoded step (composed function); cf. the 'diversity' which it also introduces (see §4.1).

To facilitate such encoding, we represent each AES component as a lookup table (an array of 2^m n-bit vectors, mapping m-bit inputs to n-bit outputs). An AES *implementation generator* program takes as input an AES key and a random seed, and outputs a key-customized WBAC-resistant AES implementation. Composition of lookup tables is straightforward, and done by the implementation generator. The resulting implementation consists entirely of encoded lookup tables, with functionalities shown in Fig. 1 (to be discussed).

Taken to an unrealistic extreme, one could use a single lookup table of about 5.4×10^{39} bytes representing the 128×128 bit AES bijection from plaintext to ciphertext for a given key. This could be attacked only as a black box. We attempt to approximate this with tables of very much smaller size, as follows.

3.1 Partial Evaluation with Respect to the AES Key

Using standard terminology [15,23], AES consists of N_r rounds; $N_r = 10$ for AES-128. A basic round has four parts: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. An *AddRoundKey* operation occurs before the first round; the *MixColumns* operation is omitted from the final round. Blocks of 128 bits are processed, and each round updates a set of 16 8-bit AES cells. To generate key-customized instances of AES-128, we integrate the key into the *SubBytes* transformation by creating 160 8×8 (i.e. 8-bit in, 8-bit out) lookup tables $\mathbf{T}^r_{i,j}$ defined as follows (one per cell per round):

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

Here S is the AES S-box (an invertible 8-bit mapping), and $k^r_{i,j}$ is the AES subkey byte in position i, j at round r. These 'T-boxes' compose the *SubBytes* step with the previous round's *AddRoundKey* step.

The round 10 T-boxes also absorb the post-whitening key as follows:

$$\mathbf{T}^{10}_{i,j}(x) = S(x \oplus k^9_{i,j}) \oplus k^{10}_{sr(i,j)} \qquad i = 0,\ldots,3,\ j = 0,\ldots,3\ , \qquad (2)$$

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

Remark. Of itself, partial evaluation provides little security: the key is easily recovered from T-boxes because the S-box is publicly known. Further (networked) encoding is used to make partial evaluation useful.

3.2 Applying Encodings to Large Linear Transformations

To efficiently implement the wide function (32-bit) *MixColumns* step, we use standard matrix blocking, combined with concatenated and networked encodings

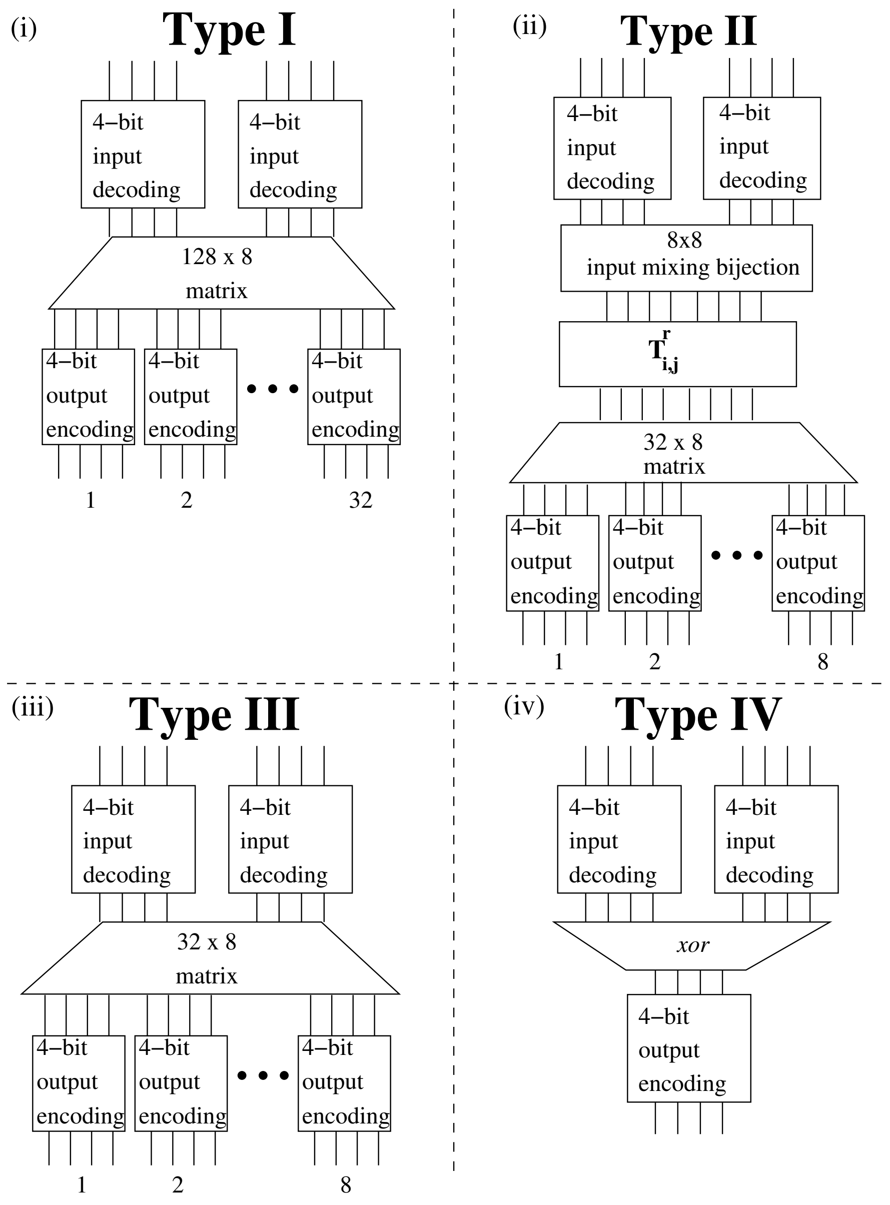

Fig. 1. Functionality of four table types in white-box AES implementation. Type I tables compute strips corresponding to the external encoding (§3.4). Tables of types II and III respectively compute strips in the first and second half of AES round transformations. Type IV tables are used to compute certain GF(2) additions (*xor*s).

for white-box protection. *MixColumns*, which operates on the AES state a column (four 8-bit cells) at a time, can be implemented [23] by multiplying a 4×4 matrix over GF(2^8) and a 4×1 vector. We use a 32×32 matrix *MC* times a 32×1 vector over GF(2), using four copies of *MC* to operate on the full 128-bit state.

We consider *MC* 'strips' (see Fig. 2): we block *MC* into four 32×8 sections, MC_0, MC_1, MC_2, MC_3. Multiplication of a 32-bit vector $x = (x^0, \ldots, x^{31})$ by *MC* is considered as four separate multiplications of the 8-bit $(x^{4i}, \ldots, x^{4i+7})$ by MC_i (yielding four 32-bit vectors y_0, y_1, y_2, y_3), followed by three 32-bit binary additions (*xor*s) giving the final 32-bit result y. We further subdivide the additions into twenty-four 4-bit *xor*s with appropriate concatenation (e.g. $((y_0^0, y_0^1, y_0^2, y_0^3) + (y_1^0, y_1^1, y_1^2, y_1^3)) \| ((y_0^4, y_0^5, y_0^6, y_0^7) + (y_1^4, y_1^5, y_1^6, y_1^7)) \| \ldots)$. By using these strips and subdivided *xor*s, each step is represented by a small lookup table. In particular, for $i = 0 \ldots 3$, the y_i are computed using 8×32 tables Ty_i (Fig. 1(ii)), while the 4-bit *xor*s become twenty-four 8×4 tables (Fig. 1(iv)).

Note that the *xor* tables, regardless of their order of use, take in 4 bits from each of two previous (e.g. partial y_i) computations. The output encodings of those computations must be matched by the input encodings for the *xor* tables. It turns out that as a consequence of pulling in 4-bit pieces from two separate computations (lookups), we require the use of concatenated 4-bit encodings for type IV tables; this imposes similar limitations on all other types, i.e. the use of 4-bit bijections. 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 T-boxes and 8×32 Ty_i's could be represented as separate lookup tables. Instead, we compose them creating new Ty_i's computing the *SubBytes* and *AddRoundKey* transformations as well as part of *MixColumns*. This saves both space (to store the T-boxes) and time (to perform the table lookups). Following our broad strategy, we insert input and output encodings around the *xor* tables (Fig. 1(iv)), and after the 32×8 matrix incorporating MC_i (Fig. 1(ii)).

ShiftRows is implemented by providing appropriately shifted input data (plaintext) to the generated tables, i.e. by run-time code during each round. The composition of *SubBytes* and *MixColumns*, and use of 8×32 lookup tables, resembles a proposed Rijndael implementation (§5 in [14]). Here we have composed also the *AddRoundKey* part and inserted encodings for added benefits of WBAC-protection, offset by the cost of being much larger and slower.

Remark. Ideally for security, we would explicitly avoid linear transformations. But randomly choosing bijections, essentially all will be non-linear: of the $2^n!$ n-bit bijections, $2^n \prod_{i=0}^{n-1}(2^n - 2^i)$ are affine — so for $n = 4$, of $16! \approx 2.09 \times 10^{13}$, only 322 560 (less than .000 002%) are affine (i.e. linear or translations thereof).

3.3 Insertion of Mixing Bijections

So far, we have used only (internal) encodings which are non-linear (or very likely so). Considering encodings as encipherments of AES intermediate values, such encodings are confusion steps. To further disguise the underlying operations, we now introduce linear transformations as diffusion steps, and for this reason

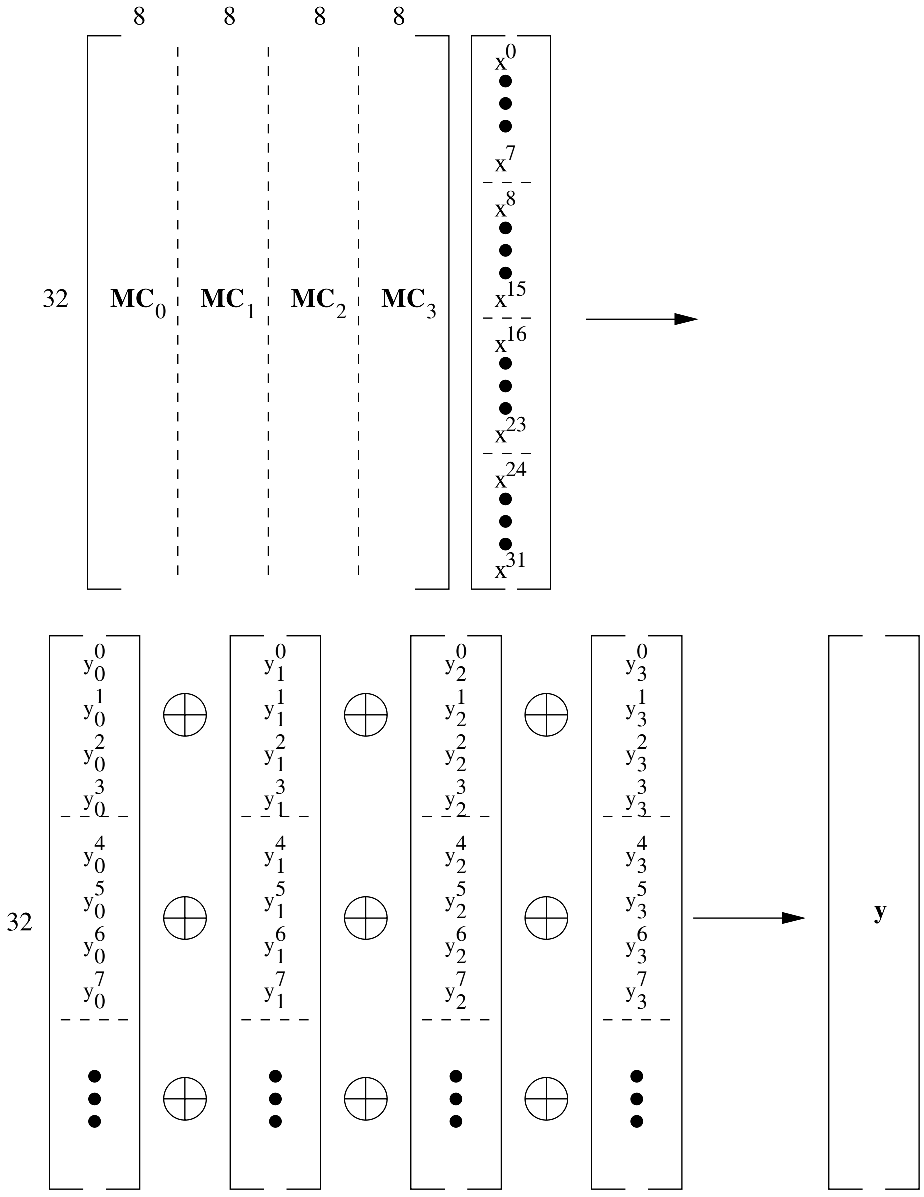

Fig. 2. Multiplication by *MC* (AES *MixColumns* operation)

refer to a linear bijection as a *mixing bijection*. Since linear transformations are representable as matrices, we think of mixing bijections as matrices over GF(2).

We use 8×8 (input) mixing bijections (Fig. 1(ii)) to diffuse the T-box inputs (technically, the inputs to the combined T-box/*MixColumns* step); these are inverted by an earlier computation. Moreover, before splitting MC into MC_i as above, we pre-multiply MC (i.e., left-multiply, yielding $MB{\cdot}MC$, so MC operates on data first) by a 32×32 mixing bijection MB chosen as a non-singular matrix with 4×4 submatrices of full rank. (See Xiao and Zhou [28] for a way to generate such matrices.) This design decision is related to using 4-bit encodings above.

To invert MB, an extra set of tables is used to calculate MB^{-1}, similar in form to those calculating MC. In type III tables, MB^{-1} is pre-multiplied by the inverses of the appropriate four input mixing bijections, and split into four 32×8 blocks. Implementing these via 8×32 tables[1] diffuses the 8×8 mixing bijections over several lookup tables. Corresponding *xor* tables are also needed; for appropriate applications, the detrimental size and speed implications are offset by MB's security benefits.

Summary. A set of tables is used for each of four 32-bit strips of state in an AES round. For each strip, a type II table combines a T-box (a partial evaluation of the *SubBytes* function with respect to a key byte) with three transformations: input mixing bijections, MC, MB; type IV *xor* (GF(2) addition) tables follow. Then a type III table combines MB^{-1} with the inverse of the input mixing bijections for the next round's T-boxes; again type IV *xor* tables follow. The type II, III and IV tables are all encoded using concatenated 4-bit (internal) input- and output-encodings in a networked fashion (see §2.2, §2.3).

3.4 Input and Output Data Manipulations (External Encoding)

As described in §2.2, our implementation takes input in a manipulated form, and produces output in another manipulated form, making the WBAC-resistant AES harder to separate from its containing application. The techniques described in previous sections, intended to securely handle both small non-linear steps (e.g. S-boxes) and large linear steps (e.g. *MixColumns*), are again used here to combine linear and non-linear components in the external encoding.

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

The following is one suggestion for data manipulation, essentially corresponding to the input and output encodings F, G in §2.2. Select two 128×128 mixing

[1] For a 32×8 (8-bit in, 32-bit out) matrix A, the corresponding 8×32 (8-bit in, 32-bit out) lookup table is defined by $B[x] = A \cdot x$.

bijections U^{-1} and V in which all aligned 4×4 submatrices are of full rank. Insert U^{-1} prior to the first AES *AddRoundKey* operation; insert V after the last AES *AddRoundKey* operation. Pre-multiply U^{-1} by the inverted input mixing bijections for $\mathbf{T}^1$. Matrix-block the result into 128×8 strips; precede and follow these by concatenated 4-bit networked input and output encodings. These output encodings are inverted by the usual set of type IV *xor* tables. To complete the networking, the output encodings on the last stage of *xor* tables supporting U^{-1} invert the input decodings of round 1. We also compose tables representing $\mathbf{T}^{10}$ with tables computing the strips of V (cf. §3.3). This external encoding is implemented in type I tables (see Fig. 1(i)). Where the context of the generated implementation permits, we can further compose operations immediately preceding U^{-1} or following V into the U^{-1} and V tables as per §2.2.

3.5 Overall Implementation

The AES implementation now consists entirely of table lookups. Sixteen 8×128 type I tables implement an input mixing bijection over the full state, along with supporting type IV tables. Each of the first 9 rounds is then performed by a series of type II and III lookup tables, plus supporting type IV tables. Sixteen 8×128 type I tables are used to combine the final round with the output mixing bijection, again supported by corresponding type IV tables.

As shown in Fig. 1, type I tables represent two 4-bit decodings, a 128×8 matrix, and 32 4-bit encodings. Type II tables represent two 4-bit decodings, an 8×8 (input) mixing bijection, a T-box, a 32×8 matrix representing $(MB\cdot MC)_i$, and 8 4-bit encodings. Type III tables represent two 4-bit decodings, a 32×8 matrix, and 8 4-bit encodings. Type IV tables represent two 4-bit decodings, the known *xor* operation, and a 4-bit output encoding.

3.6 Size and Performance

The total size of lookup tables in the resulting implementation is 770 048 bytes[2], and there are 3 104 lookups during each execution.[3] A fair comparison is the AES implementation of Daemen and Rijmen [14], which requires 4 352 bytes for lookup tables, and approximately 300 operations (lookups and *xor*s) in total. The expected increase in the size of an implementation is thus about 177×. (The performance slowdown is not as easy to measure; our implementation showed a slowdown of 55×, but this is presumably sensitive to the layout of the tables in memory and the size of the cache.) While WBAC-protection is important in hostile environments, 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 §1).

[2] Type II and III: $9\times2\times4\times4=288$ 8×32 tables = 294 912 bytes;
type IV supporting II, III: $9\times2\times4\times8\times3=1\,728$ 8×4 tables = 221 184 bytes;
type I tables: $2\times16=32$ 8×128 tables = 131 072 bytes;
type IV supporting I: $2\times32\times15=960$ 8×4 tables = 122 880 bytes.

[3] Type II and III: 288 lookups; type IV supporting II, III: 1 728 lookups;
type I tables: $32\times4=128$ lookups; type IV supporting I: 960 lookups.

4 Preliminary Security Comments on White-Box AES

Some immediate security observations are made in §1. Beyond these, an obvious question is: *can use of (external) encodings F and G as per §2.2 weaken the ordinary black-box security of E_K?* This seems unlikely — if with any significant probability, these key-independent, random bijections render $G \circ E_K \circ F^{-1}$ weaker than E_K, then intuitively one expects the cipher E itself is seriously flawed.

4.1 White-Box Diversity

We assume that encodings are random and independent, except for those which are inverses of each other; choosing the encodings is one-time per implementation work. Keyspace provides an upper bound on the security of a cryptographic algorithm. Analogously, if encodings 'encrypt' implementation steps, we can count the possible encoded steps, and call this metric *white-box diversity.* The white-box diversity of table types in Fig. 1 is the number of distinct *constructions* (or equivalently, distinct *decompositions*) — for type II tables, this includes varying the key — which exist for all possible tables of that type; this exceeds the number of distinct tables. (E.g., if constructing a table requires n independent choices to be made, and the ith choice has c_i alternatives, then the white-box diversity of the table is $\prod_{i=1}^{n} c_i$.) White-box diversity measures variability among implementations, which is useful in foiling pre-packaged attacks against specific instances. Implementations may differ both in time at a single site, and in space across sites (cf. [11,16]). The *white-box diversity* for the table types of Fig. 1 (see also §3.5) is:

Type I:[4] $(16!)^2 \times 20\,160^{64} \times (16!)^{32} \approx 2^{2\,419.7}$
Type II:[5] $(16!)^2 \times 256 \times 2^{62.2} \times 2^{256} \times (16!)^8 \approx 2^{768.7}$
Type III: $(16!)^2 \times 2^{256} \times (16!)^8 \approx 2^{698.5}$
Type IV: $(16!)^2 \times 16! \approx 2^{132.8}$

This is vastly more than needed for a diversity defense (see Forrest *et al.* [16]), but does not measure the resistance to key extraction from a specific instance.

4.2 White-Box Ambiguity

A far more important metric is the *white-box ambiguity* of a table type, which estimates the number of distinct *constructions* which produce *exactly the same table* of that type, computed by dividing its white-box diversity (see §4.1) by the (usually much smaller) number of distinct tables of that type. This gives an average measure of how many alternative interpretations (meanings) exist for an instance of a specific table type; certain white-box attacks must disambiguate

[4] There are 20 160 nonsingular 4×4 matrices over GF(2).

[5] The number of 8×8 mixing bijections is roughly $2^{62.2}$. The actual number of type II tables is slightly lower, as not every 8×32 matrix can be produced as a product of MC and mixing bijections.

among these. (E.g., since for a type II table, changing its T-box key-byte and changing its input encoding can have the same effect, the ambiguity of a type II table includes all possible key bytes.) Type IV tables have the lowest ambiguity. Ambiguity is intended as a defense against *disambiguation*, in the sense that greater ambiguity is likely to make disambiguation more difficult:[6]

Definition 4 (disambiguation) *To* disambiguate *a table is to narrow its set of possible constructions below the cardinality given by the* white-box ambiguity *for its table type, restricting its potential decompositions. Disambiguation is* total *if the set of possible constructions contains one element; otherwise, it is* partial.

Finding a rigorous and tractable way to compute white-box ambiguity appears difficult. We have made estimates by extrapolating from smaller tables preserving our basic structure, and assuming that the constructions are equiprobable (which is only approximately true).

For estimation purposes, assume type I tables are built from 128×8 matrices having all aligned 4×4 blocks of full rank, and 2-bit input and output encodings. We modelled them by combinatorially more tractable 4×4 matrices with all aligned 2×2 blocks of full rank, computing the number of distinct constructions. We carried out these constructions and counted the number of distinct resulting tables. We observed that for a given scaled-down such table, its two input encodings together with one of its $b\times b$ (here, $b = 2$) blocks in each aligned set of b rows uniquely determines the other $b\times b$ block in the aligned set of b rows and the output encodings. This observation also held for $b = 2$ and 6×4 matrices, and we conjecture that it holds for the real $b = 4$, 128×8 matrices. By this reasoning, the ambiguity of a type I table is $(16!)^2 \times 20\,160^{32} \approx 2^{546.1}$.

Type II and III tables are more complex: the matrix blocks may not have full rank. We discuss only construction of type III tables here. The rank of each block is a function of the the table, as follows. Consider the components needed to compute a single, fixed nybble of each entry in a type III table. We have two 4-bit input decodings, each feeding into a 4×4 block of the 32×8 matrix, and finally a 4-bit output encoding. If we arrange the 256 32-bit outputs in a 16×16 array, the effect of the first input decoding is to permute the array rows, while the second input decoding permutes its columns. The rank of the first block can be determined by taking the base 2 logarithm of the number of distinct entries in any array column; the rank of the second by taking the base 2 logarithm of the number of distinct entries in any array row. (E.g., a rank 3 block preserves 3 bits of information; over all 16 inputs it yields 8 distinct outputs; $\log_2 8 = 3$.)

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.

[6] For example, the greatest possible ambiguity would be achieved by an implementation of AES-128, with input and output data manipulation, as a single (infeasibly) immense table. At this limiting point, the power of a white-box attacker is reduced to that of the black-box attacker.

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 $20\,160^2$ possible such blocks, an upper bound for the number of components which could produce a given table is $(16!)^2 \times 20\,160^2 \approx 2^{117}$. 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. One input decoding, together with the value to which 0 decodes for the other input decoding, uniquely determines the remaining encodings, so the number of constructions yielding a given type IV table is $16! \times 16 \approx 2^{48.2}$. It is not clear, however, how an attacker could determine which of these alternatives corresponds to the generated AES implementation. Total disambiguation of the eight type IV tables feeding into a set of type II tables (see §3.3 *Summary*) would permit removal of decodings for that set of type II tables, allowing the attack of §4.3.

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.

4.3 A Generic Square-like Attack

We describe a generic attack, possible only in the white-box context where the attacker has full control of the key-instantiated AES implementation, and when all input encoding is removed (i.e., $\mathbf{T}^r_{i,j}$ inputs are fully exposed) for the set of four type II tables performing a round transformation for one column. (§4.4 shows how to perform such removal for weakened variants of our AES implementation.) While we can send arbitrary texts through these simplified tables, whose inputs are unencoded inputs to the *AddRoundKey* and *SubBytes* steps, the outputs are still obscured by the *MB* transformation and the output encoding.

Consider the value of an AES cell after our two-part round transformation. It has undergone an 8-bit mixing bijection and two concatenated 4-bit random bijections. This encoding is local to the cell, and therefore has the following property: *two texts having the same encoded value in a 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* determine when it is zero.

Our goal is to find two 32-bit texts which have a non-zero input difference in each of the four cells, and a zero output difference in all but one cell (called a 'three-cell collision'). This can be recognized as the strategy in the first round of the Square attack on AES [14]. 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.

$$y_i = S(w_i \oplus k_i) \oplus S(x_i \oplus k_i) \ . \tag{3}$$

Then we have three equations in four unknowns y_i, with hexadecimal coefficients determined by the *MixColumns* matrix [23]:

$$\texttt{01}\cdot y_0 \oplus \texttt{02}\cdot y_1 \oplus \texttt{03}\cdot y_2 \oplus \texttt{01}\cdot y_3 = \texttt{00}\ , \tag{4}$$

$$\texttt{01}\cdot y_0 \oplus \texttt{01}\cdot y_1 \oplus \texttt{02}\cdot y_2 \oplus \texttt{03}\cdot y_3 = \texttt{00}\ , \tag{5}$$

$$\texttt{03}\cdot y_0 \oplus \texttt{01}\cdot y_1 \oplus \texttt{01}\cdot y_2 \oplus \texttt{02}\cdot y_3 = \texttt{00}\ , \tag{6}$$

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

$$y_0 = \texttt{ec}\cdot y_3\ , \quad y_1 = \texttt{9a}\cdot y_3\ , \quad y_2 = \texttt{b7}\cdot y_3\ .$$

Thus, the choice-count for k has been reduced to at most 256 (by equation (3)). Exhaustive search now finds k.

Based on the probability of a three-cell collision ($\binom{4}{1}$ cells in which to occur $\times$ probability $\frac{255}{256}$ of a non-collision $\times$ probability $(\frac{1}{256})^3$ of three collisions $\approx 2^{-22}$) and a birthday-paradox counting argument, the expected work to find an entire round key is approximately 2^{11} texts $\times$ four 32-bit columns $= 2^{13}$ one-round encryptions for this weakened variant.

4.4 Partial Design Justification by Examining Weakened Variants

We show here that removing certain aspects of our design destroys its security.

Need for Input and Ouptut Data Manipulation. The weakened WBAC-resistant AES variant without input and output data manipulation (see §3.4), beyond suffering the problem discussed in §2.2, has no input encoding for the first set of type II tables, and is thus vulnerable to the attack in §4.3. Thus such manipulations are necessary for security.

Need for Internal Mixing Bijections. Diffusion (cf. §3.3) is crucial to the security of white-box implementations: without it, patterns in the underlying tables allow disambiguation, as we illustrate in the attack methods below. Hence, mixing bijections are essential in the intermediate steps of the cipher.

Consider an implementation in which T-boxes are not preceded by input mixing bijections (see Fig. 1(ii)) nor followed by *MB*, thus having no type III tables. Each of its type II tables implements two 4-bit decodings, followed by a T-box, then a known 32×8 matrix, and finally eight 4-bit encodings. Within such a 32×8 matrix, two 8×8 blocks are identities (multiplications by the `01` polynomial), so considering only parts of the output coming from such an identity, we can read an encoded version of the underlying T-box from the type II table, computing a function of the form $T' = (G_1\|G_2)\circ T\circ(F_1^{-1}\|F_2^{-1})$, where T is some $\mathbf{T}^r_{i,j}$ function *per* equation (1) or (2). We can rewrite this as $T' = (B_1\|B_2)\circ S\circ(A_1\|A_2)$ where S is the *SubBytes* function, each B_i is an output nybble encoding (rounds 1–9) or such an encoding composed with a post-whitening nybble *xor* (round 10), and each A_i is a nybble *xor* (from *AddRoundKey*) composed with an input decoding (rounds 1–10).

By ignoring nybble values *per se*, and considering only left and right nybble frequencies described below, we can recover output encodings for the T' for each type II table in rounds 1–9, where $B_1 = G_1$ and $B_2 = G_2$. Since the A_i and B_i are 4×4 bijections, a table for T' can be derived from the 16×16 *SubBytes* table (see [23] p. 16, Fig. 7) by row (due to A_1) and column (due to A_2) permutation, together with bijective left (due to B_1) and right (due to B_2) renumbering of entry nybbles. (Random choice of nybble swapping in table elements or the table input affords no further protection, as shown below.) We disambiguate as follows.

Definition 5 (frequency signature) *Let L be an $n \times n$ array of byte entries. L has n^2 cells $L[rc]$ where r is the row and c is the column. For a sequence of left (respectively, right) nybbles in a row (respectively, column) of L, its* frequency signature *is the sequence of n occurrence frequencies for values `0`–`F`, sorted into descending order, written as a string of n digits in a sufficiently large base.*

E.g., `F013AB073AF32126` has the frequency signature 3222221110000000.

Definition 6 (cell signature) *For an $n \times n$ array L of byte entries, the* cell signature *of a cell $L[rc]$ in row r and column c is the $4n$-digit concatenation of the frequency signatures for (a) row r left nybbles; (b) row r right nybbles; (c) column c left nybbles; and (d) column c right nybbles.*

For example, in the *SubBytes* table above, the *SubBytes*[`00`] cell signature is 4421111110000000 4311111111100000 3222221110000000 4222211110000000.

There are 192 distinct cell signatures in *SubBytes*. 160 apply to unique cells. Each of the remaining 32 applies to a 3-set of cells with indices $[rc]$ of the form $[xd]$, $[yd]$, and $[zd]$, where among 3-sets, d ranges over all hex digits and either (a) $x =$ `4`, $y =$ `6`, and $z =$ `9` or (b) $x =$ `5`, $y =$ `7`, and $z =$ `F`.[7]

This 3-set co-ordinates pattern is invariant under any combination of: (a) column permutation; (b) bijective renumbering of entry left nybbles or right nybbles; and (c) swapping left and right nybbles in all entries. A row permutation bijectively renumbers the rows and the corresponding alternatives for x, y, z above, but otherwise preserves the 3-set pattern above. Swapping input nybbles is easily identified: it interchanges rows and columns in the pattern. Entry nybble swapping is identified by the numerically highest signature, which has different values for swapped and unswapped.

Given a 16×16 T' table for a T' function, for any table cell $T'[r'c']$ there is a corresponding cell *SubBytes*$[rc]$ with the same cell signature. After correcting for entry or input nybble swapping, 160 T' cells are identified immediately with *SubBytes* cells having their signatures. Every left and right nybble occurs in a unique-signatured *SubBytes* (and hence T') table cell: comparing corresponding

[7] The *SubBytes* shared-cell-signature n-set pattern is atypical: of 10 000 16 × 16 S-boxes filled with pseudo-randomly selected permutations of the byte values from 0 to 255 using L'Ecuyer's recommended generator [21] with its default seeds, just 4.6% contained any, and just 1.7% contained only, such n-sets with n divisible by 3.

entries, we find B_1 and B_2. Then the T' table's row and column permutations (relative to *SubBytes*) define A_1 and A_2, and thus reveal the function T.

In rounds 1–9, B_1 and B_2 are the identity-block output encodings for T'. Non-identity blocks are known (corresponding to multiplications by `02` and `03` polynomials), so correcting for these known bijective blocks, we obtain output codings for the non-identity blocks. We can thus find a round's type II table output codings, from which we know all succeeding type IV tables input codings. Since these encode *xor*s, we can then determine their output codings. We proceed in a similar fashion until we reach the next round's type II table input codings, which are the inverses of the (now known) output codings of the immediately preceding type IV tables. We can then remove them, creating the conditions necessary to launch the Square-like attack of §4.3.

Round-Pair Attack. A simpler and faster attack on this weakened variant (without internal mixing bijections) is as follows. We perform the above disambiguation process on two successive rounds in the range 2–9. For the second round of the round-pair, as noted above, we then know the input encodings, and the previous disambiguation techniques give us the output encodings. We can write any round 2–9 T' function as

$$T' = (G_1||G_2) \circ S \circ (X_1||X_2) \circ (F_1^{-1}||F_2^{-1}) \ , \tag{7}$$

where the G_i are the output nybble encodings, S is the *SubBytes* function, the F_i are the input nybble encodings, and the X_i denote *xor*s with *AddRoundKey* nybbles. Given the corresponding type II table in the second round of the round-pair, we can now identify G_1, G_2, F_1 and F_2. Removing these encodings, we derive the function $T = S \circ (X_1||X_2)$, where $T(x) = S(x \oplus k)$ for any byte x by equation (1), and k is the key-byte hidden in T by partial evaluation (see §3.1). This allows key extraction directly from (weakened) type II tables, without any encryption operations (required in the Square-like attack). However, it requires absence of mixing in two successive rounds.

Summary. The mixing bijections thwart the above attacks by diffusing information over bytes instead of nybbles for both inputs and outputs of type II tables: they make it impossible to write their input encodings as concatenations of nybble-to-nybble bijection pairs, thereby causing them to have T' cell signatures unlike those of the *SubBytes* table, and they eliminate identity blocks in the 32×8 matrices of the type II tables, thereby making it hard for an attacker to separate the effects of their output encodings from those of the matrices.

5 Conclusions and Future Work

The white-box attack context (WBAC) reflects both the capabilities of an adversary who can introduce malicious code, and the reality of untrusted hosts. Traditional implementations of cryptographic algorithms, including AES, are completely insecure in the face of these threats. As is well-known with smartcards, in practice the security of a cipher is dependent on its environment and implementation, as well as its mathematical underpinnings.

As a proposal for pragmatically acceptable white-box attack-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. As a bonus feature, aside from white-box strength, generated AES implementation instances provide a diversity defense against pre-packaged attacks on particular instances of executable software.

Further security analysis is needed, and we encourage the wider cryptographic community to participate. In this paper, we consider the concept of component disambiguation, a Square-like attack, and attacks on weakened variants. The issues of attacks on multiple components at once, or on multiple implementations sharing a key, remain to be investigated. For example, white-box ambiguity for a *sub-network* of tables in our implementation may differ depending on where its boundaries lie.

Although AES implementations using optimized versions of the techniques in this paper have been found acceptable in several commercial applications, their large size and low speed limit general applicability. Efficiency improvements would be most welcome. We also encourage the extension of white-box techniques to other algorithms (cf. Chow *et al.* [9]).

Acknowledgements. We thank Alexander Shokurov for suggesting the ambiguity metric of §4.2 in another context, and anonymous reviewers.

References

1. J. Algesheimer, C. Cachin, J. Camenisch, G. Karjoth, *Cryptographic Security for Mobile Code*, pp. 2–11 in Proceedings of the 2001 IEEE Symposium on Security and Privacy, May 2001.
2. R.J. Anderson, M.G. Kuhn, *Low Cost Attacks on Tamper-Resistant Devices*, pp. 125–136, 5th International Workshop on Security Protocols (LNCS 1361), Springer 1997.
3. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang, *On the (Im)possibility of Obfuscating Programs*, pp. 1–18, Advances in Cryptology – Crypto 2001 (LNCS 2139), Springer-Verlag, 2001.
4. E. Biham, A. Shamir, *Differential Fault Analysis of Secret Key Cryptosystems*, pp. 513–525, Advances in Cryptology – Crypto '97 (LNCS 1294), Springer-Verlag, 1997. *Revised*: Technion - C.S. Dept. - Technical Report CS0910-revised, 1997.
5. E. Biham, A. Shamir, *Power Analysis of the Key Scheduling of the* AES *Candidates*, presented at the 2nd AES Candidate Conference, Rome, Mar. 22–23 1999.
6. D. Boneh, R.A. DeMillo, R.J. Lipton, *On the Importance of Eliminating Errors in Cryptographic Computations*, J. Cryptology 14(2), pp. 101–119, 2001.
7. 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`.
8. 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.

9. S. Chow, P. Eisen, H. Johnson, P.C. van Oorschot, *A White-Box* DES *Implementation for* DRM *Applications*, Proceedings of DRM 2002 – 2nd ACM Workshop on Digital Rights Management, Nov. 18, 2002 (Springer-Verlag LNCS, to appear).
10. 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.
11. F. Cohen, *Operating System Protection Through Program Evolution*, Computers and Security 12(6), 1 Oct. 1993, pp. 565–584.
12. J. Daemen, V. 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.
13. J. Daemen, M. Peeters, G. 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.
14. J. Daemen, V. Rijmen, AES *Proposal: Rijndael*, `http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf`, 1999.
15. J. Daemen, V. Rijmen, *The Design of Rijndael:* AES *– The Advanced Encryption Standard*, Springer, 2001.
16. S. Forrest, A. Somayaji, D. 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.
17. M. Jakobsson, M.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.
18. P.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.
19. P. Kocher, J. Jaffe, B. Jun, *Differential Power Analysis*, pp. 388–397, Advances in Cryptology – Crypto '99 (LNCS 1666), Springer-Verlag, 1999.
20. O. Kömmerling, M.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.
21. P. L'Ecuyer, *Efficient and Portable Combined Random Number Generators*, Communications of the ACM 31(6), pp. 742–749, 1988.
22. National Institute of Standards and Technology (NIST), *Round 2 Discussion Issues for the* AES *Development Effort*, November 1, 1999.
`http://csrc.nist.gov/encryption/aes/round2/Round2WhitePaper.htm`.
23. 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`
24. T. Sander, C.F. Tschudin, *Towards Mobile Cryptography*, pp. 215–224, Proceedings of the 1998 IEEE Symposium on Security and Privacy.
25. T. Sander, C.F. Tschudin, *Protecting Mobile Agents Against Malicious Hosts*, pp. 44–60, Mobile Agent Security (LNCS 1419), Springer-Verlag, 1998.
26. N. van Someren, A. Shamir, *Playing Hide and Seek with Keys*, pp. 118–124, Financial Cryptography '99 (LNCS 1648), Springer-Verlag, 1999.

27. C. Wang, *A Security Architecture for Survivability Mechanisms*, Doctoral thesis, University of Virginia, October 2000.
28. J. Xiao, Y. 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.