A Cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards

Suresh Chari * Charanjit Jutla * Josyula R. Rao * Pankaj Rohatgi *

March 31, 1999

Abstract

NIST has considered the performance of AES candidates on smart-cards as an important selection criterion and many submitters have highlighted the compactness and efficiency of their submission on low end smart cards. However, in light of recently discovered power based attacks, we strongly argue that evaluating smart-card suitability of AES candidates requires a very cautious approach. We demonstrate that straightforward implementations of AES candidates on smart cards, are highly vulnerable to power analysis and readily leak away all secret keys.

To illustrate our point, we describe a power based attack on the Twofish Reference 6805 code which we implemented on a ST16 smart card. The attack required power samples from only 100 independent block encryptions to fully recover the 128-bit secret key. We also describe how *all* other AES candidates are susceptible to similar attacks.

We review the basis of power attacks and suggest countermeasures for a secure implementation. Unfortunately, it appears that these software countermeasures result in significant memory and efficiency overhead and therefore the *most efficient* smart card implementation cannot serve as a guide in evaluating AES candidates.

Keywords: Smart-Cards, Power Analysis, AES Candidates, AES Evaluation Criterion.

1 Introduction

The official call for AES candidate algorithms [2], outlined three evaluation criteria for AES submissions: "security", "cost" and "algorithm and implementation characteristics". Included in "cost" were *Computational Efficiency* and *Memory Requirements* and "algorithm and implementation characteristics" included the criterion of *Flexibility*. The following was stated as an example of flexibility.

> b. The algorithm can be implemented securely and efficiently in a wide variety of platforms and applications (e.g., 8-bit processors, ATM networks, voice & satellite communications, HDTV, B-ISDN, etc.)

*I.B.M. T.J.Watson Research Center P.O.Box 704, Yorktown Heights, NY 10598, U.S.A. Email: schari,csjutla,jrrao,rohatgi@watson.ibm.com

NIST has justified the importance of smart-card implementation feasibility for AES candidates on the basis of these evaluation criteria , i.e., candidates should be flexible enough to have a reasonably efficient implementation on smart-card CPUs with a small RAM/EEPROM/ROM footprint. For example, in [3], NIST lists smart-cards under the Cost/Efficiency criterion. Most AES submissions have also discussed 8-bit CPU or smart-card performance [1]. This aspect has also been investigated in depth in [6], as part of an effort to compare the performance of all AES candidates on a variety of platforms. It is stated there that for widespread smart-card use, the algorithm must not only fit in high-end smart-cards but also in the most primitive smart-cards, i.e, those which have at most 128 bytes of RAM, of which only 64 bytes would be available to the encryption algorithm. On that basis, the argument is made that only CAST-256, Crypton, DEAL, Rijndael, SAFER+, Serpent and Twofish are suitable for widespread smart-card use[1].

In this paper, we strongly argue that the evaluation of AES candidates on smart-cards cannot be done simply by comparing the most efficient implementation on a typical smart-card CPU, because these implementations are very likely to be insecure. This is especially true for low end smart-cards which are highly vulnerable to Power Analysis based attacks [5] due to their simplicity. Therefore, any claim that an AES candidate is superior because it runs on a low end smart card (with 128 bytes of RAM) would be hollow if, in fact, the implementation leaks away its key in a power based attack. In its foresight, NIST phrased the *Flexibility* evaluation criterion as being applicable to *secure and efficient* implementations and not just to *efficient* implementations. We therefore assert that when evaluating smart-card implementations of AES algorithms, power based attacks on the implementation cannot be ignored and wished away, since these are inherent in simple, low cost devices. Instead, a true comparison can only be made with respect to power attack resistant implementations, which, in many cases, would be less efficient and more resource intensive than the fastest implementation.

To illustrate our point, we describe the vulnerability of naive smart-card implementations of AES algorithms to power attacks. Taking advantage of the publicly available 6805 Reference Code of one of the AES candidates, Twofish, and a 6805 based smart card available to us (a ST16 [2]) we mounted a power-analysis attack on the implementation. It should be noted that the same attack will work on most commonly available 6805 smart-cards.

Our investigations yielded startling results: The Twofish Reference 6805 code when implemented on a ST16 smart card together with a 128-bit "secret" key, requires power samples of the input whitening process from only 100 independent block encryptions to completely extract the key. With experience, the number of samples can be reduced to 50.

The attack uses Differential Power Analysis (DPA) [5] on the input whitening process to extract the input whitening subkeys and uses them to reverse-engineer the master key. Note that we did not have to attack the rounds of Twofish, since for 128-bit keys, the key-expansion step to derive input whitening keys is "almost reversible".

This attack has severe practical implications: An *honest* smart-card user who accidentally does 50 secure transactions with another *trustworthy* entity via a rigged smart-card reader risks exposing her key to the party who controls the reader. Major smart-card ap-

[1] This is highly debatable since other candidates can also fit in such cards with subkeys kept in EEPROM, or with EEPROM used as RAM swap space for limited transaction applications.

[2] ST16 is manufactured by SGS-Thomson Microelectronics.

plications such as e-cash and Pay-TV require much higher security assurance, i.e., even malicious smart-card users who can resort to physical attacks should not be able to extract the smart-card's internal keys. Clearly this implementation, even when protected by simple hardware countermeasures [3] does not meet this requirement.

This problem and attack is hardly limited to Twofish: based on our practical experience with Twofish and with the ST16, we analyzed the vulnerability of all AES candidates, without actually implementing them. We discovered that straight-forward implementations of almost all candidates would have similar vulnerabilities, although many would require more effort than Twofish. This is because many candidates explicitly or implicitly employed the design principle that deriving the master key or other subkeys from a subset (or a small subset) of subkeys should be computationally infeasible. Hence, power attacks on these algorithms are proportionally harder since several rounds have to be attacked.

The rest of the paper is organized as follows: Section 2 describes a simple model for smart card power consumption. Using this, we motivate our attack and explain the basis of all smart card power attacks. This model is also key in designing and understanding the effectiveness of countermeasures. In Section 3, we describe in detail the attack on the Twofish reference code. In Section 4 we investigate possible vulnerabilities of straight-forward implementations of all other AES candidates. Section 4.15 summarizes the vulnerabilities. In Section 5 we discuss possible countermeasures against power attacks including a general technique that can be employed to create "practically secure" smart-card implementations, i.e., implementations that require an impractically large number of power-samples to be break. However all these countermeasures come at the cost of reduced performance and increased code size and memory requirements.

2 A Simple Power Model for Smart-Cards

Most smart-cards use CMOS technology which consumes power only when some change occurs in the logic state of the chip. No significant power is needed to maintain a state. Such changes could include changes in the contents of the RAM, internal registers, bus-lines, states of gates and transistors etc. Smart-card chips are clock driven, i.e., almost all activity is triggered by an internal or external clock edge and usually all activity ceases before the next clock edge is due. A few processes, such as on-chip noise generators etc, operate independently of the clock and consume a small, possibly random amount of power continuously.

Each clock edge triggers a sequence of power consuming events (such as transistor switching, charging of internal and external lines etc) within the chip which brings it to the next state. This sequence of events is determined by the microcode executing within the chip and depends on parts of the current state of the processor and parts of the state of other subsystems that processor is accessing in that cycle. We use the term *relevant state* to denote parts of the overall state of the chip which determine the sequence of events during a clock cycle. Depending on the cycle, the relevant state could include values of the bits of some internal registers, bits on internal and external buses, address and contents of the external memory location being accessed etc. The smart-card being a digital circuit, its

[3] E.g, on chip filters etc., which can be easily bypassed by physical attacks

relevant state can be represented by a binary string.

The instantaneous power consumption of the chip, shortly after a clock edge, is a combination of the instantaneous power consumption components from each of the events that have occurred thus far. The power component from each event depends on several factors such as the electrical properties of the chip substrate and layout, at the current operating conditions (temperature, voltage, etc), as well as coupling effects between events occurring in close proximity. The exact timing of each event is also likely to be equally complex, even though the microcode will dictate a certain ordering between events. As a first approximation, we ignore coupling effects and create a linear model, i.e., we assume that the power consumption function of the chip is the *sum* of the power consumption functions of all the events that have taken place. This model may be applicable to somewhat larger chips as well.

Consider a particular cycle of a particular instruction in the execution path of some fixed code. At the start of the cycle, the smart card can be in one of several relevant states depending on the input and processing done in earlier cycles. Let $\mathcal{S}$ denote the set of possible relevant states when control reaches this cycle and let $\mathcal{E}$ be the space of all possible events that can occur in that cycle. For each $s \in \mathcal{S}$, and each $e \in \mathcal{E}$, let $occurs(e, s)$ be the binary function which is 1 if e occurs when the relevant state is s and 0 otherwise. Let $delay(e, s)$ be the time delay of the occurence of event e in state s from the clock edge and let $f(e, t)$ denote the power consumption impulse function of event e with respect to time t (here $t = 0$ when the event occurs and $f(e, t) = 0$ for $t < 0$). Then in our model, $P(s, t)$, the power consumption function of the chip in that cycle with state s and time t after the clock edge can be written as

$$P(s,t) = \sum_{e \in \mathcal{E}} f(e, t - delay(e,s)) * occurs(e,s)$$

In reality, due to random asynchronous power consuming components in the chip and noise introduced within the chip itself (e.g., due to local variations in operating conditions and minor coupling effects) the actual power is better modeled by adding noise components to it, i.e.,

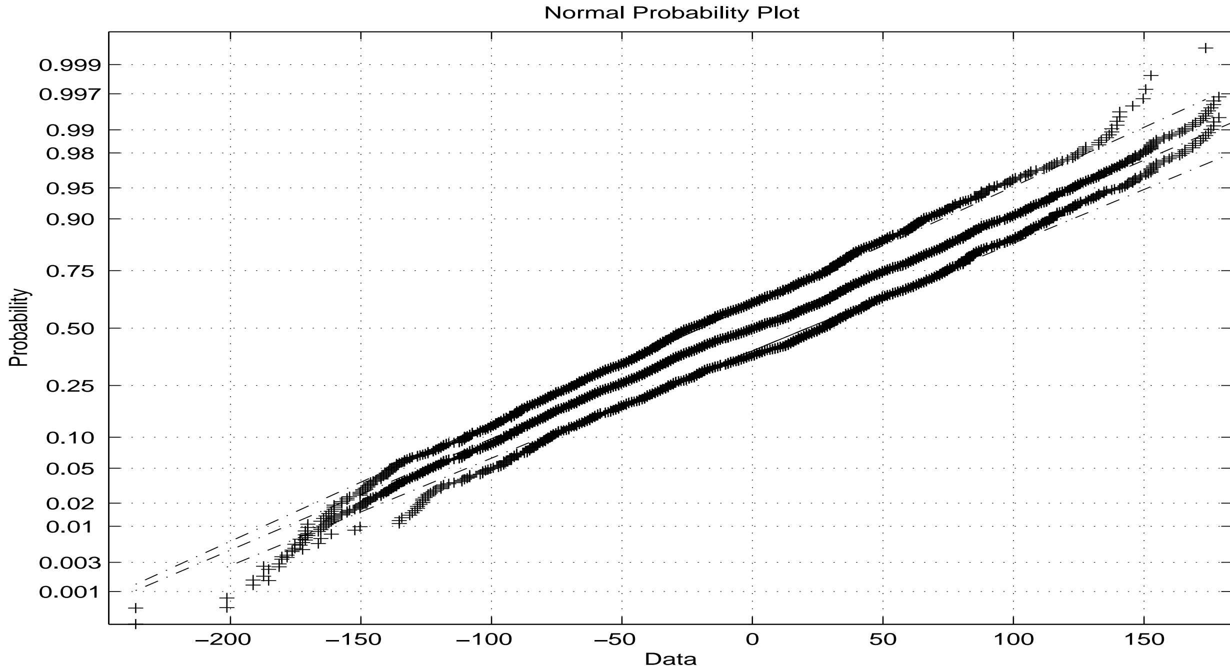
$$P(s,t) = \mathcal{N}_c(t) + \sum_{e \in \mathcal{E}} (\; f(e, t - delay(e,s) + \mathcal{N}_d(e,s)) + \mathcal{N}(e,t)) * occurs(e,s); \qquad (1)$$

where $\mathcal{N}(e, t)$ is a (small) Gaussian noise component associated with the power consumption function of event e, $\mathcal{N}_d(e, s))$ is a small Gaussian noise component affecting the delay function and $\mathcal{N}_c$ is the (small) Gaussian external noise component.

Equation 1 shows that there is a strong dependence between the power consumption function and the relevant state s at that cycle. This is because different events occur in different states and even if the same set of events were to occur in two different states, their timing could be different. This dependence is at the core of all differential power analysis attacks which seek to exploit asymmetries in the power consumption function with respect to state. These asymmetries are particularly pronounced in low end smart cards where the relevant state space is quite small.

We discuss these attacks in detail in the next section.

Figure 1: Power distributions when loading random bytes from RAM



2.1 Differential Power Attacks

Consider two different probability distributions $D1$ and $D2$ on the relevant state s before the clock edge of a certain cycle. From equation 1, it is very likely that the distribution of the instantaneous power when the state is drawn from $D1$ is going to be different from the distribution of the instantaneous power when the state is drawn from $D2$ and these cases can be distinguished by statistical tests on power samples. This difference and distinguishability between the two distributions is the basis for differential power attacks. In most well known attacks [5], the distributions $D1$ and $D2$ are very simple, e.g., $D1$ is the uniform distribution on the set of all states which have a particular state bit 1 and $D2$ is the uniform distribution on the set of all states which have that bit 0. The difference in the power distribution for these two cases represents the effect of that particular state bit on the net power consumption.

As an example, Figure 1 shows three distinct distributions of the instantaneous power consumption of the ST16 chip in the middle of a cycle which loads the value of a RAM byte into the accumulator. These three distributions correspond to three different distributions on the value of that particular RAM byte. All three power distributions are plotted on a "normal scale" and each distribution shows up as a thick line in this plot, which means all these three power distributions are close to normal. The middle line corresponds to the power distribution when the RAM byte is drawn uniformly at random. It has a mean of 0 (we have shifted all power readings by an additive constant to enforce this). The top line corresponds to the power distribution when the RAM byte is uniformly chosen from all bytes with MSB of 1. It has a mean of -25. The bottom line corresponds to the power distribution when the RAM byte is uniformly chosen from all bytes with MSB 0. This has

a mean of +25.

3 Power based attack on Twofish implementation

3.1 Target implementation

The target implementation was on an ST16 smart card by loading publicly available Twofish Reference code for the 6805 [8] onto the EEPROM of a ST16 smart card. The default implementation options in the code were retained. We created an interface which allowed us to send a 128-bit Twofish key and a 128-bit plaintext as a command to the ST16 and the card would return the ciphertext.

Although, a real implementation of Twofish would go into the ST16's ROM, we are quite confident that the behavior of such an implementation with respect to our attack would be identical. This is because, the fact that the code is coming from EEPROM instead of ROM only affects the processor cycles which fetch opcodes and operand addresses and not the cycles dealing with the data and the key since these will always be in RAM. Therefore our attack against the handling of plaintext data and key will be equally effective in both situations.

3.2 Power Attack Equipment

The power attack equipment consisted of a special smart-card reader with a current sensor attached to the Vcc contact. The output of this current sensor was sampled using a PC-based oscilloscope and data acquisition board and software from Gage Applied Sciences Inc. This equipment could acquire and record 12-bit power samples at 100Mhz sampling rate on to the PC's hard disk. The board could be programmed to trigger and record a specified number of samples upon receiving an external signal, thus minimizing the amount of sample data collected for each encryption. A PC with a 2GB disk was enough to collect and store samples from thousands of encryptions. Another PC was used to send commands to the smart card and triggering information to the board.

3.3 Attacking the Twofish whitening process

In the Twofish whitening process, a 128-bit whitening key consisting of 4, 32-bit whitening key words K_0, K_1, K_2, K_3 are exor'ed into the input data block. The reference code to this is given below. Note that the xor4 operation which seems to do a word whitening in the code is actually a macro which expands to do byte-wise whitening since the 6805 is an 8-bit machine.

```
jsr  computeSubkey /* Function call to calculate K0, K1 in sk0, sk1 */
xor4 Text, sk0     /* Macro to whiten 1'st word */
xor4 Text+4,sk1    /* Macro to whiten 2'nd word */
jsr  computeSubkey /* Function call to calculate K2, K3 in sk0, sk1 */
xor4 Text+8, sk0   /* Macro to whiten 3'rd word */
xor4 Text+12,sk1   /* Macro to whiten 4'th word */
```

Upon macro expansion, the actual 6805 code for the whitening process is:

```
jsr computeSubkey /* Function call to calculate K0, K1 in sk0, sk1 */
lda Text          /* Load 1'st byte of 1'st word in accumulator */
eor sk0           /* Xor accum with contents of 1'st byte of sk0 */
sta Text          /* Store accum back as 1's byte of 1'st word */
lda Text+1        /* Load 2'nd byte of 1'st word in accum */
eor sk0+1         /* Xor accum with contents of 2'nd byte of sk0 */
sta Text+1        /* Store accum back as 2'nd byte of 1'st word */
...
```

As observed in [5], macro features such as a group of similar operations are clearly visible in power traces of most smart cards. This will be true for the computeSubkey subroutine which is invoked during whitening and in all rounds. Therefore, by inspection, the whitening process can be identified and the equipment can be set up to collect power samples of just the whitening process for each encryption.

Let us focus on the least significant bit (LSB) of the byte "Text" in RAM, a plaintext bit known to the attacker. The first instruction which accesses this bit in the whitening process is "lda Text". This brings Text, and in particular its LSB into the accumulator. The next instruction exor's the whitening key into the accumulator and the next instruction stores the result back into the RAM variable Text. If the LSB of the whitening key is 1, then Text LSB is negated by the exor, and if the whitening LSB is 0, Text LSB remains unchanged. These are the ONLY places where this bit is directly manipulated in the input whitening process. Note however that when this bit is manipulated, e.g., brought from RAM into the accumulator over the data bus, it value remains in some parts of the circuit (e.g., buses or internal lines, latches, registers, etc) until it is overwritten by activities performed in subsequent cycles. Therefore this bit will be part of the relevant state for the next few cycles. From the code, it is easy to verify that the same properties hold for all input and whitening key bits.

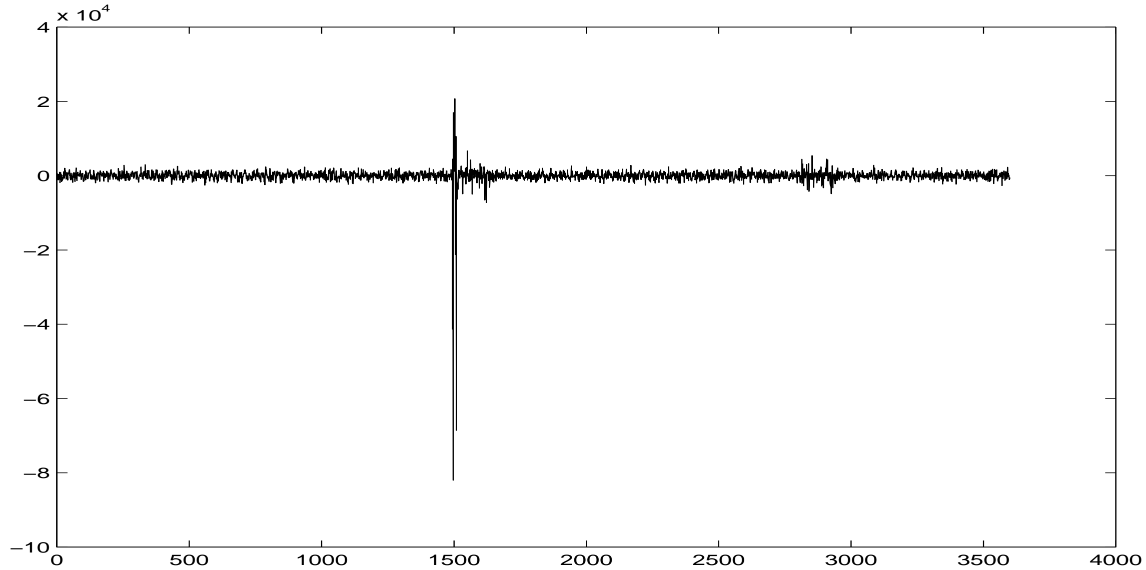
The DPA attack against Twofish is now easy to describe: Obtain power samples of the whitening process for several random encryptions. For each of the bits in the plaintext input to these encryptions, calculate the co-variance between the bit and power samples of the runs at every sample point.For each bit i, look at the co-variance plot for all the sample points. It should be flat except for a few strong peaks. Look at the first significant peak. That corresponds to the load of the plaintext. Look at the co-variance at a sample point corresponding to the store of the whitened plaintext. The co-variance here should also be a peak. If these peaks have the same sign then the i'th bit of the key is 0, else it is 1.[4]

3.4 Experimental Results

We performed the above experiment using 2000 random encryptions and then repeated it with 500, 100 and 50 samples to see how few samples are needed to attack.

[4]This assumes that the i'th bit's average contribution to the power in both a load from RAM and a store to RAM has the same sign, which was true in our experiment. In the worst case, depending on the smart-card, a subset of the 128 bits will have similar contributions and the rest will have dissimilar contributions. Once this subset is known, a similar attack can be mounted against that smart-card. This subset can easily be learned by experimenting with load, exor and store operations with known operands.

Figure 2: Pointwise Co-variance between 1'st data bit and Power during whitening



Each experiment yielded 128 co-variance plots, one for each plaintext/key bit. Using the above approach we were able to predict correctly all the 128 key bits when working with 2000, 500 and 100 samples. At 50 samples, there was difficulty defining "first significant peak", since the co-variance at other points was almost comparable to the real "first peak". However, if the adversary could identify where the "first peak" should be, e.g., by looking at the shapes of the actual power signal and the code to figure out the correct cycle number (something we, with experience with ST16, can easily do), then comparing the signs of the two peaks still worked.

For example, Figure 2 shows the covariance plot of the whole whitening process for the first bit. At higher resolution, i.e., zooming on to the region of the peaks, we see in Figure 3 that the first peak at position 1495 is -ve and the peak corresponding to the store at 1506 is -ve and hence the key bit is 0. Note the presence of a few peaks after position 1495 and after 1506. These represent cycles where the lingering value of the directly manipulated bit in the earlier cycle is overwritten from parts of the state. Figure 4 which is the covariance plot of the next input bit shows the first peak at 1495 is +ve and the peak corresponding to the store at 1506 is -ve which shows that the next key bit is 1.

3.5 From whitening keys to the 128-bit master key

We assume that the reader is familiar with the specification of Twofish and the notation used in [7]. After the attack we know 32-bit whitening keys K_0, K_1, K_2, and K_3. From the Twofish specification, K_0 and K_1 are derived from the quantities A_0 and B_0 using a Pseudo-Hadamard Transform (PHT) and K_2 and K_3 are similarly derived from A_1 and B_1. By inverting the PHT we derive the quantities A_0, A_1, B_0, B_1 from the whitening keys. From the specification of A_0, A_1, B_0, B_1 we then derive the value of the function $h(0, M_e)$, $h(\rho, M_o)$, $h(2\rho, M_e)$ and $h(3\rho, M_o)$, where if $M = (M_0, M_1, M_2, M_3)$ are the four words of

Figure 3: Closer look at peaks in Figure 2

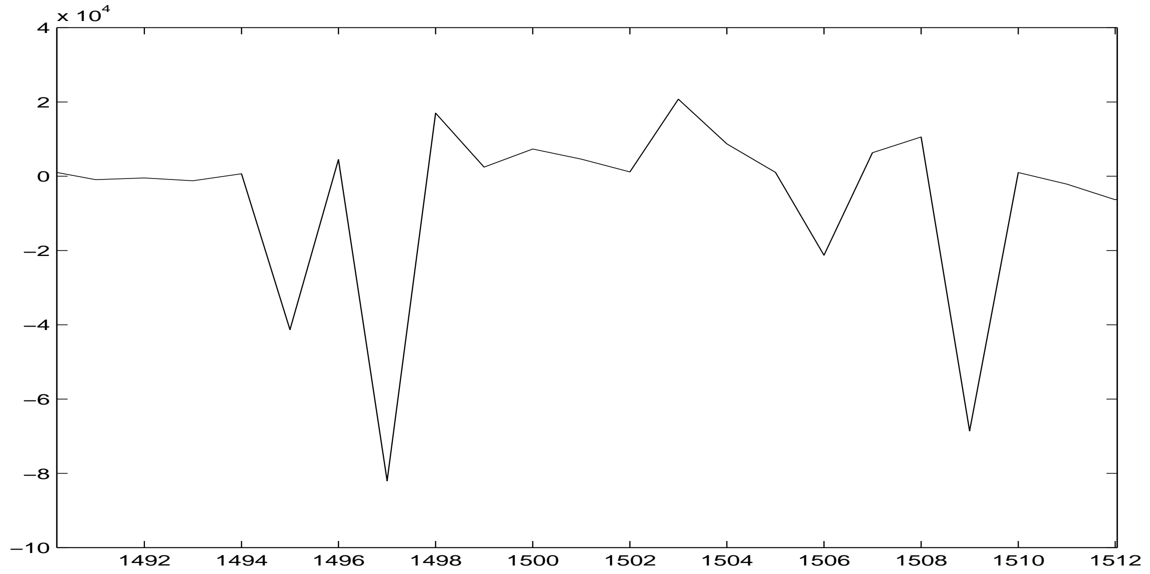
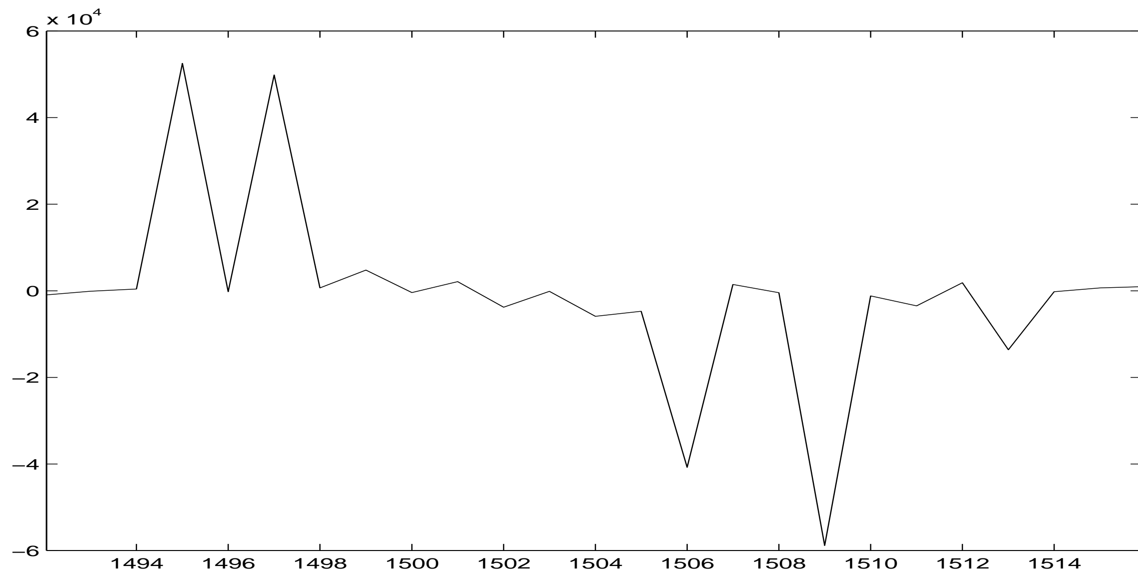


Figure 4: Co-variance Peaks for 2'nd data bit



the master key then $M_e = (M_0, M_2)$ and $M_o = (M_1, M_3)$;

We then use our knowledge of $h(0, M_e)$ and $h(2\rho, M_e)$ to create possible candidates for (M_0, M_2) and similarly our knowledge of $h(\rho, M_o)$ and $h(3\rho, M_o)$ to create possible candidates for (M_1, M_3). Since the process is identical we only describe how to derive very few candidates for (M_0, M_2) using $h(0, M_e)$ and $h(2\rho, M_e)$.

When considering $h(0, M_e)$ and $h(2\rho, M_e)$, $L_1 = M_2$ and $L_0 = M_0$ in the specification of the h function. Consider $h(0, M_e)$. From the specification, the four byte $h(0, M_e) = (z_3, z_2, z_1, z_0)$ is derived as

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} \cdot\ \cdot & & \cdot\ \cdot \\ \cdot\ \cdot & MDS & \cdot\ \cdot \\ \cdot\ \cdot & & \cdot\ \cdot \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

for some bytes (y_0, y_1, y_2, y_3). Since the MDS matrix is invertible, from $h(0, M_e) = (z_3, z_2, z_1, z_0)$ we derive (y_0, y_1, y_2, y_3). By definition of h,

$$\begin{aligned} y_0 &= q_1[q_0[q_0[0] \oplus l_{1,0}] \oplus l_{0,0}] \\ y_1 &= q_0[q_0[q_1[0] \oplus l_{1,1}] \oplus l_{0,1}] \\ y_2 &= q_1[q_1[q_0[0] \oplus l_{1,2}] \oplus l_{0,2}] \\ y_3 &= q_0[q_1[q_1[0] \oplus l_{1,3}] \oplus l_{0,3}] \end{aligned}$$

for fixed permutations q_0 and q_1 in the Twofish specification. Now note that if we know the value of y_0 and that

$$y_0 = q_1[q_0[q_0[0] \oplus l_{1,0}] \oplus l_{0,0}]$$

then we know that for any possible value of the byte $l_{1,0}$ there can be at most one value of the byte $l_{0,0}$. We can therefore create a 256-entry dependence table T_0 between $l_{1,0}$ and $l_{0,0}$. Similarly, knowing the values of y_1, y_2, y_3 allows us to create similar dependence tables T_1 between $l_{1,1}$ and $l_{0,1}$, T_2 between $l_{1,2}$ and $l_{0,2}$ and T_3 between $l_{1,3}$ and $l_{0,3}$.

Similarly, unrolling the definition of $h(2\rho, M_e)$, we get yet another set of dependence tables G_0, G_1, G_2, G_3 for the same quantities. The actual values of the bytes $l_{1,0}$ and $l_{0,0}$ can only be those values in which T_0 and G_0 coincide. Since T_0 and G_0 are derived from different starting points in the first q_0 lookup that defines y_0, T_0 and G_0 are likely to have very few co-incidences. In experiments typical coincidences have ranged from $2 - 9$.

Thus from $h(0, M_e)$ and $h(2\rho, M_e)$, we get on average between $2 - 9$ possibilities for each the 4 byte pairs in M_e. A similar analysis using $h(\rho, M_o)$ and $h(3\rho, M_o)$ yields $2 - 9$ possibilities for each the 4 byte pairs in M_o. So we are very likely to have less than 9^8 possible 128-bit master keys which are consistent with a single whitening key. This is a very small number of possibilities for the key and the right key can then be identified by encrypting known plaintext by all these possible keys to see which one yields the correct ciphertext.

If we had attacked the output whitening process and derived the output whitening keys, the attack and results would be very similar. Had we attacked both the input and output whitening, we would have very few candidate keys to try out.

4 Power Analysis Vulnerabilities of other AES Candidates

4.1 CAST-256

Unless protected, CAST-256 will leak all subkeys with DPA. The masking keys can be attacked as in Twofish and the rotation keys can be obtained by DPA based on the guess of the rotation amount and correlating with the predicted output of the S-boxes. However due to the strong key schedule design principle employed, it appears that all rounds will have to be attacked to extract all subkeys.

4.2 CRYPTON

Unless protected, the whitening keys can be extracted as in Twofish. Similarly the keys exored in at the end of the first round can be extracted. These keys are sufficient to fully reverse engineer the 128-bit master key.

4.3 DEAL

Unless protected, we can use the Kocher's DPA attack on the DES functions in the first two rounds of DEAL to extract the DES key used. Each DPA attack will be applied on the power samples on the first two rounds of DES. After the first two DES keys are extracted the entire DEAL key can be reconstructed. To its advantage, DEAL can reuse the DPA resistant implementations of DES that are currently in the market to make itself DPA resistant.

4.4 DFC

To break DFC we need to be able to extract the 128-bit key (a, b) when it is used for the operation $(ax + bmod(2^{64} + 13))mod(2^{64})$, which should be easy unless protected. Due to the pseudorandom nature of the key expansion process, it appears that these attacks have to be carried on each round to extract all subkeys.

4.5 E2

Unless protected, DPA will be needed to strip each application of keys in E2 to get the entire set of derived keys. This is because of the following sound design principle followed in E2.

Requirement 5. Deriving master key or other subkeys from some subkeys should be computationally infeasible.

Exor keys used in the IT can be extracted as in Twofish. Multiplication keys used in the IT can be extracted by a bit-by-bit DPA starting from the LSB. Round keys can be extracted by a DPA attack similar to the Kocher DPA attack against DES.

4.6 FROG

FROG has decent resistance to DPA. The bomb permutation and the xorBuf can be extracted by DPA using a reasonable number of samples. However, the use of a key dependent Subst Permutation makes DPA attack less efficient since a large number of samples will be

required to ferret out each of the elements of the Subst Permutation. This forces the adversary to have roughly 256 times the number of samples as compared to other algorithms on the same smart card.

4.7 Hasty Pudding Cipher

By design the key expansion process in HPC is supposed to be lossy so that it cannot be easily reversed nor can one subkey (or table) be used to derive another. This makes HPC have decent resistance to DPA, since a large number of samples of several steps of the cipher will be needed to ferret out each entry in each of the five key tables.

4.8 LOKI-97

Unless protected,a DPA attack similar to the DES DPA attack, i.e., prediction of S-box output based on guesses of few key bits can be used to derive the round subkeys and strip away rounds. A direct DPA attack may also work on the permutation sub-key. If the ciphertext is not available, then due to the non-linear key schedule based on an unbalanced Feistel network, all rounds have to be attacked to get all subkeys. On the other hand if ciphertext is available, then the attack should be mounted on the last 4 rounds, which will give away the master key.

4.9 Magenta

Unless protected DPA will be effective in stripping out all the key bytes. The best place to attack would be the calculation of $PE(x, y)$ when one byte is known data and the other is a key. Due to the simplistic nature of the key-schedule only 2 rounds need to be attacked for 128-bit keys.

4.10 Mars

Unless protected, the whitening keys used in Mars can be obtained as in Twofish. Due to complex key schedule at least a large number of core rounds also need to be attacked to get all subkeys. For each subsequent expansion box, the exor key should be attackable by DPA and the multiplication key by a bit-by-bit DPA.

4.11 RC6

Unless protected, the additive input and output whitening keys can be extracted by bit-by-bit DPA. The additive round keys can be extracted similarly. Due to complex key schedule it appears that all rounds need to be stripped away to get all subkeys.

4.12 Rijndael

Unless protected, DPA can extract the Round 0 keys which are exored in after which all subsequent subkeys can be derived for 128-bit Rijndael. The authors of Rijndael claim a 34 byte smart card implementation which they claim will be DPA resistant on "good

hardware". In reality, as of July 1998 [5], no such smart-card hardware existed and the authors need to be more forthcoming.

4.13 SAFER+

Unless protected, SAFER+ is vulnerable to key add/exor DPA attacks and DES like DPA attacks(i.e., table lookup after key exor/add). From the key schedule, it is clear that only a half-round needs to be attacked for a 128-bit key.

4.14 SERPENT

Unless protected, SERPENT would be vulnerable to same type of DPA attacks as DES (i.e., S-box lookup after key exor). From the key schedule, it is clear that the first 2 round subkeys should be sufficient to derive the master key.

4.15 Summary of Power Vulnerabilities of AES Candidates

Based on the above analysis, we can summarize the power analysis vulnerabilities of straightforward implementations of AES candidates by placing them in three categories.

Easiest to attack are ciphers like CRYPTON, DEAL, LOKI-97, Magenta, Rijndeal, SAFER+, SERPENT and Twofish, where DPA needs to be done only on very few rounds, since very few subkeys give enough information about the master key or all other subkeys.

Slightly harder to attack would be ciphers like CAST-256, DFC, E2, Mars and RC6, where DPA will have to be carried out on several, if not all rounds. These ciphers have the property that it is hard to get the master key or other subkeys from a several other subkeys.

Hardest to attack would be FROG and HPC which employ large key dependent tables, which would require a large number to samples to perform a DPA attack to ferret out each entry in these tables. However, it may be hard for these ciphers to maintain these large key dependent tables within the RAM constraints of even some advanced smart card architectures and they may need to use EEPROM to do so. However, EEPROM reads tend to leak a lot of information about the value being read, and secure implementations of these ciphers will need to exercise a lot of care about how this is done.

5 Countermeasures to Power Analysis

From a theoretical viewpoint, given current smart card architectures and their information leakage model, it may be impossible to guard against all forms of power analysis attacks. This is especially true if the adversary has a very good power consumption model for a particular smart card, access to the source code, extremely high precision and high frequency sampling equipment and a lot of time.

However, from a practical perspective, it is usually sufficient that an implementation be resistant only to well known generic attacks such as Simple Power Analysis (SPA), Differential Power Analysis (DPA) as well as esoteric Higher-Order Differential Power Analysis Attacks [5]. The internal organization of most smart cards are kept as closely guarded secrets by the smart card hardware vendors and card specific attacks can be mounted only by very few adversaries who have knowledge of the card internals or have the large resources to derive a cards power consumption model via experimentation. Generic attacks, on the

other hand, can be mounted by anyone and therefore pose a much more serious risk. Also, from a practical perspective, it is easy to limit the number of encryptions a smart card will perform and hence the number of samples available for generic attacks. E.g., a limit of 1 million encryptions can easily be enforced without affecting legitimate users.

With a reasonable upper bound on the number of power samples an adversary can collect, it is possible to guard against large number of generic power based attacks. To protect against SPA, the code execution path should be independent of the key and data. Also, all means to introduce noise in the power signal should be enabled to prevent an adversary from gaining enough information about the ongoing computation from a single power sample and force him to resort to statistical tests (such as DPA or higher order DPA) involving a large number of samples.

At an upper bound of 1 million, one should assume that the adversary can exploit every relevant state bit in any instruction to mount a DPA attack, provided he can efficiently predict that bit in a significant fraction of the runs based on the code specification, known inputs and small number of guesses for parts of the key. With a large number of samples, simple countermeasures involving "balancing" approaches (i.e., trying to negate the effects of one set of events by another "complementary" set) will probably fail because the power consumption functions and the timing of even two "complementary" events will be slightly different and the adversary can maximize these differences by adjusting the operating conditions (e.g., temperature, voltage, external clock) of the card.

Yet another approach involving random sequencing of operations and random delays may also be quite ineffective. Unless this random sequencing and delaying is done extensively, it can be undone and a canonical order re-created by applying signal processing tools on the power signal. Attacks can then be mounted on re-ordered signals. Even when full re-sequencing is not possible, it is enough to identify "corresponding" sample points in a large number of runs which having the property that a significant fraction of these points are samples from the same power function P for the same cycle, whereas other points are power samples from unrelated power functions for other cycles. All statistical attacks that work for P are also applicable on these "corresponding" points, although a lot more samples would be needed to get rid of the noise introduced by the unrelated samples.

A sure defense against DPA would be to modify the code to ensure that the adversary cannot predict any relevant bit in any part of the computation, without making several other run-specific assumptions. This makes statistical tests involving even several tens of runs impossible, since the chances of the adversary making the correct assumptions for each of those runs is extremely low. While this approach solves the problem of DPA, it is not clear how one can do serious computation if this requirement is to be satisfied since no bit that depends directly on the data and key can be manipulated at any cycle. Sometimes, the function being computed itself has algebraic properties that permits such an approach, e.g., for RSA one could use blinding [4, 9] to hide the actual values being manipulated. However such structure is unlikely to be present in block ciphers.

One general technique would be to randomly split every relevant bit into several (say k) shares. Each of the k shares and furthermore every collection of $k-1$ shares should be statistically independent of the bit required for computation. Computation can then be carried safely by performing computation on the shares. This share based technique needs to be applied for a sufficient number of steps into the computation until the adversary has

very low probability of predicting bits, i.e., till sufficient secret key dependent operations have been carried out. Similar splitting also has to be done at end of the computation if the adversary can get access to its output. With a splitting technique, the adversary needs is forced to mount k'th order differential attack, i.e., statistical information needs to be collected from at least k points in each run in order to relate it to a bit of the computation. At the same time, for simple power models or even for actual smart card implementations, one can empirically deduce the number of runs needed to distinguish whether a 1 bit is being coded or a 0 bit is being coded by using the information theoretic *likelihood ratio test*. Our experiments based on a very simple power consumption model show that the number of runs required grow exponentially in k and therefore very easily one can force an adversary to require more samples than will be available.

However, the above technique is also very demanding on the code and memory requirements. It therefore remains to be seen how large secure implementations of AES candidates on smart cards will be.

6 Acknowledgements

We thank Helmut Scherzer for porting the 6805 reference code to the ST16, Steve Weingart for setting the power attack hardware, and Don Coppersmith, Helmut Scherzer and Martin Witzel for help with the power model and countermeasures.

References

[1] Proceedings of the First Advanced Encryption Standard Candidate Conference. Aug 20-22, 1998.

[2] "Request for Candidate Algorithm Nominations for AES", Federal Register, Volume 62, Number 177, pp 48051–48058, Sept 1997.

[3] J. Foti. "Status of the Advanced Encryption Standard (AES) Development Effort". 1998 National Information Systems Security Conference.

[4] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems. Advances in Cryptology-Crypto '96, Lecture Notes in Computer Science # 1109, pp 104–113.

[5] P. Kocher, J. Jaffe and B. Jun. "Introduction to Differential Power Analysis and Related Attacks". http://www.cryptography.com/dpa/technical/index.html

[6] B. Schneier, J. Kesley, D. Whiting, D. Wagner, C. Hall and N. Ferguson. "Performance Comparison of the AES Submissions". http://www.counterpane.com/AES-performance.html

[7] B. Schneier, J. Kesley, D. Whiting, D. Wagner, C. Hall and N. Ferguson. "Twofish: A 128-Bit Block Cipher". http://www.counterpane.com/twofish-paper.html

[8] Twofish 6805 Reference code. Available at http://www.counterpane.com/download-twofish.html

[9] D. Chaum. "Blind Signatures for Untraceable Payments". Advances in Cryptology: Proceedings of Crypto '82, Plenum Press, 1983, pp 199-203.