Leakage resistant encryption and decryption

Introduction

Data encryption and decryption operations are basic building blocks for most security applications. For this purpose, most systems use block ciphers, such as the public AES standard. It is well known, however, that implementations of block ciphers such as AES, as well as other cryptographic algorithms, are subject to side-channel attacks [1]. These attacks allow adversaries to extract secret keys from devices by passively monitoring power consumption, EM emissions, or other "side channels". Differential power analysis (DPA) is a common side channel attack that leverages power measurements.

Countermeasures are required for applications where side-channel attacks are a threat. These include several military and aerospace applications where program information, classified data, algorithms, and secret keys reside on assets that may not always be physically protected.

There are several classes of countermeasures against side-channel analysis (see [1]). However, creating a block cipher implementation that verifiably resists all side-channel attacks in all usage scenarios is challenging. Some countermeasures introduce substantial overheads in terms of gates, code-size and performance. In addition, it is difficult to be certain that advanced side channel analysis techniques, such as high-order differential power analysis, will not recover enough information to find a key. For example, if the implementation has to protect a 256-bit key, and the same key could be used to encrypt 2^{40} distinct data blocks, one has to ensure that the side-channel leaks significantly less than 2^{-32} (2.3×10^{-10}) bits of information about the key during each operation. Implementing countermeasures to meet such extreme requirements would require substantial overhead, and be quite hard to test and validate.

In this design note, we explain an alternative approach to secure data encryption and decryption operations that can use existing unprotected hardware block cipher implementations, while achieving provable security against side channel attacks. The approach is a "protocol-level" countermeasure [1] whose security is based on realistic and testable assumptions about side-channel leakage from implementations of block ciphers and hash functions. In addition, the construction allows a significant safety margin, so security does not depend on controlling and modeling devices' precise analog properties. The approach is well suited to military and aerospace applications where the implementer can control how cryptographic primitives are used to solve a particular security problem.

Encryption and decryption: side-channel exposures

In a typical data encryption scenario, the encrypting and decrypting devices begin with a shared secret key K. The encrypting device uses K to encrypt a message M using a block cipher such as AES, to produce ciphertext C. In practice, since the block cipher operates on a fixed sized data block (128-bits for AES), the message M is padded and split in block-sized chunks. The encrypting device also chooses an initialization vector (IV) and uses a standard chaining mode such as CBC (cipher block chaining) or counter-mode to encrypt the data blocks to create the corresponding blocks of ciphertext. The ciphertext C together with the IV is stored or transmitted to the recipient. The decrypting device uses the key K, the IV, and the ciphertext C to recover the message M. In some scenarios, such as storage encryption, the encrypting and decrypting devices are the same. In other scenarios, such as packet level encryption, the same key K can be used for multiple packets. Figure 1 shows a typical encryption process using AES in CBC mode. Figure 2 shows a power trace from an FPGA implementing the AES-CBC mode encryption in Figure 1.

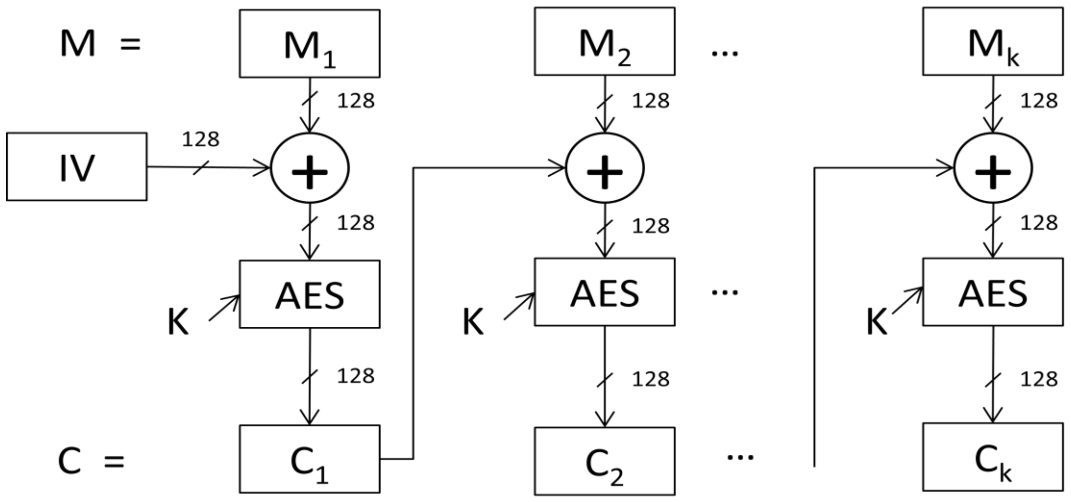
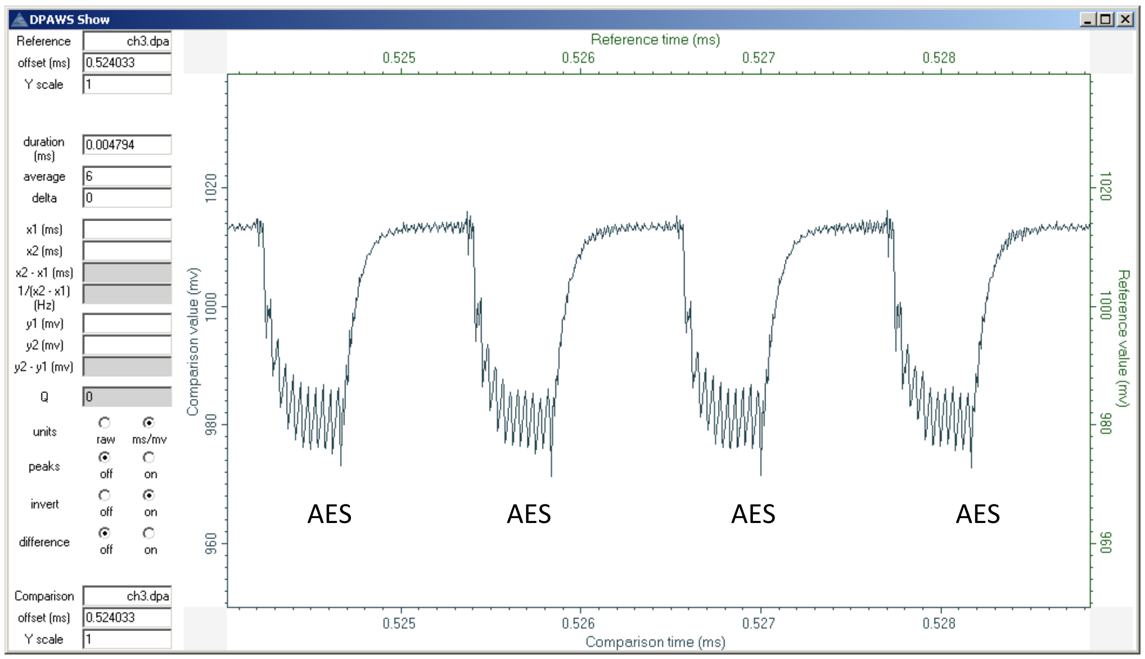
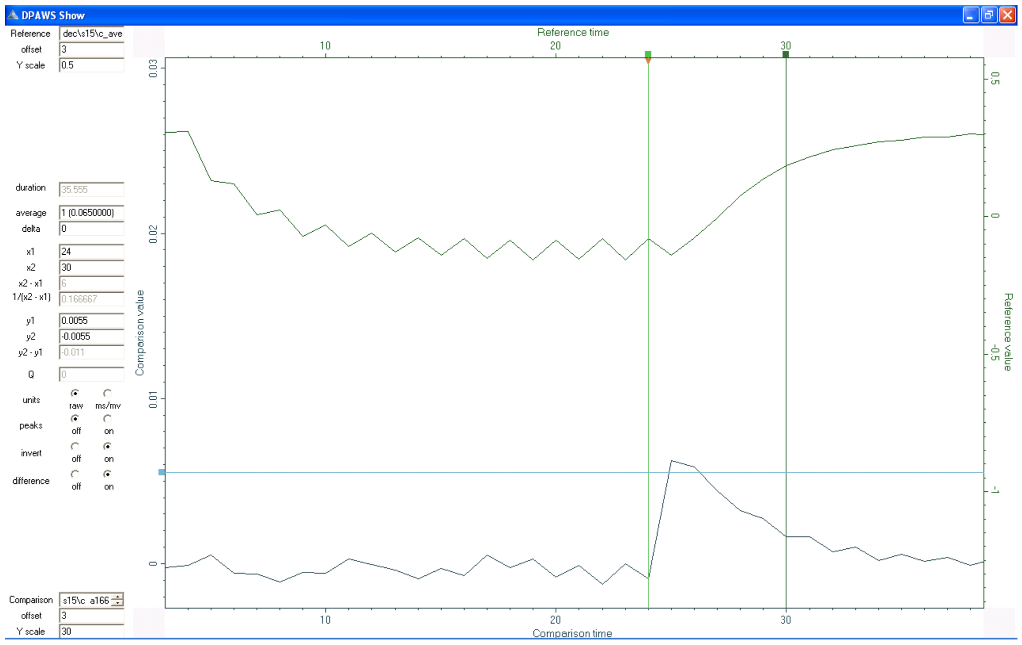


Figure 1: AES CBC mode encryption.



Reference	ch3.dpa
offset (ms)	0.524033
Y scale	1
duration (ms)	0.004794
average	6
delta	0
x1 (ms)	
x2 (ms)	
x2 - x1 (ms)	
1/(x2 - x1) (Hz)	
y1 (mv)	
y2 (mv)	
y2 - y1 (mv)	
Q	0
units	raw [✓ ms/mv]
peaks	[✓ off] on
invert	off [✓ on]
difference	[✓ off] on
Comparison	ch3.dpa
offset (ms)	0.524033
Y scale	1

Figure 2: Power trace collected from an FPGA performing the AES-CBC encryption on a 256KB message. The figure shows power consumption during four consecutive AES block encryptions, out of the 16K block encryptions used to encrypt the message.


DPAWS Show

Reference	dec\s15\c_ave
offset	3
Y scale	0.5
duration	35.555
average	1 (0.0650000)
delta	0
x1	24
x2	30
x2 - x1	6
1/(x2 - x1)	0.166667
y1	0.0055
y2	-0.0055
y2 - y1	-0.011
Q	0
units	[✓ raw] ms/mv
peaks	[✓ off] on
invert	off [✓ on]
difference	off [✓ on]
Comparison	s15\c a166
offset	3
Y scale	30

Figure 3: A successful differential power analysis attack using the individual encryption operations from the trace in Figure 2. The average trace is shown on top and the correlation trace for the correct guess (166) for byte 15 of the last round key is shown below.

A typical DPA attack on an implementation of a block cipher like AES involves performing statistical analysis on power traces from many block encryption/decryption operations with a fixed key. The attack typically involves a data collection process, during which side channel measurements and corresponding ciphertext blocks are recorded from a target device. The physical device is no longer needed; the rest of the attack involves analyzing the data set. The analysis process involves combining candidate values for small portions of the key (e.g., 8 bits at a time) with the ciphertext, then checking whether the results are correlated with the side channel measurements. The strongest correlations appear for correct values for key portions, allowing recovery of the entire key.

For the AES-CBC example, encrypting a longer message involves many AES operations with different ciphertexts. Measurements of the target device's side channels during a single encryption or decryption sequence can be sufficient to mount a DPA attack. Indeed, Figure 3 shows the result of a successful DPA attack on the AES-CBC FPGA implementation from Figure 2, from a single trace collecting during encryption of 256 kilobytes of data (16K block encryptions). The correlation shown in the figure shows the correct value of one byte of the secret key being identified using DPA. The analysis method can be repeated using the same power measurement to recover the remaining 15 bytes of the key. The entire process, including collecting the power data and recovering a complete 128-bit AES key, takes less than two minutes on a PC.

For DPA to be effective, the attacker must monitor many block encryption or decryption operations with same secret key, but many different plaintexts/ciphertexts. If a single encryption key was used to encrypt only a few data blocks, then statistical attacks such as DPA become impractical. In fact, updating the key (e.g., by using a hash function) after every few encryptions is a common protocol-level countermeasure [1], since it limits the amount of useful information that an attacker can obtain about any particular key.

For many use cases, a general solution to leakage resistant encryption and decryption requires more than just updating the encryption key. For example, a group of devices that share a top-level key secret K_{ROOT} may need to exchange multiple messages between them. To prevent key reuse, there needs to be an efficient technique to derive a fresh key for each message. The key derivation mechanism itself must not expose K_{ROOT} to DPA. For example, it is not sufficient to create random message identifiers and encrypt these with K_{ROOT} to derive message keys since that exposes K_{ROOT} to side channel analysis. For other use cases, such loading encrypted firmware or secure storage, the decrypting entity may be required to decrypt the same ciphertext multiple times. In this setting, even if a key was originally used encrypt a few blocks of data, by modifying the ciphertext, the attacker can make the decryption device use that key in arbitrarily many decryption operations with different inputs, thus creating a side-channel exposure. This exposure is not mitigated by adding a standard message authentication code (MAC) to the ciphertext; the attacker can use modified ciphertexts and side-channel analysis to first attack the MAC key. After recovering the MAC key, the attacker can target the decryption key using modified ciphertexts with forged MACs. The solution to the security problem is to construct an efficient leakage-resistant message authentication construction that enables the decrypting device to authenticate the ciphertext before decrypting it, then pair this with a leakage-resistant decryption procedure.

Leakage resistant protocol for encryption and decryption

Our solution makes use of standard hash functions and block ciphers together with three major constructions: key updates, key trees, and hash chaining.

Hash function

For several steps, we use a cryptographic hash function hash(), which can be the SHA256 algorithm or any of the SHA-3 candidates. While we don't require the full strength of cryptographic hash functions (pseudorandomness and second-preimage resistance are sufficient), strong hash functions are efficient and make good choices. Let hashLen be the length of the hash's output in bytes. For SHA256, hashLen = 32.

Block Cipher

We will use a standard block cipher for core encryption and decryption operations. For this article, we assume the AES-256 block cipher. To achieve side-channel resistance, we require the block cipher implementation be such that, for repeated encryptions and decryptions of the *same* plaintext/ciphertext block with a fixed key, no more than a few bits of information about the key and plaintext blocks are revealed from the side-channel. A good hardware implementation of AES-256, where all the s-box lookups are done in parallel should have this property.

Key Updates

During the encryption process itself, the key is updated after encrypting every few blocks. The key update frequency (denoted `nblocks`) should be tuned for the worst-case of the side-channel leakage properties assumed for the block cipher implementation as described above. If the information leakage is very low, then a larger value of `nblocks` could be used. In practice, however, we normally use more conservative choices for `nblocks` such as 4 or 8.

The update is done by setting $key_{i+1} = g(key_i)$, where g() is a pseudorandom function that is independent of the encryption and decryption functions. In practice, it is generally simplest to implement the function g() using the hash function hash(), e.g. by hashing the key and a fixed constant. The side-channel resistance assumes that repeated application of g() on the same input provides at most a small number of bits of information about the input and the output.

Key Trees

To derive the initial keys and for ciphertext authentication, we use a **key tree** construction. Let $f_0()$ and $f_1()$ be pseudorandom functions which are independent of g() and the block cipher. These functions can also be based on the hash function hash(), by prepending the hash input with two different constants that differ from the constant used for creating g(). As with g(), the information leaked from repeated application of f0 and f1 to the same input should be limited. A function keyTree(K_{root}, path) can then be construed from $f_0()$ and $f_1()$, where:

- K_{root} is the 256-bit key shared between the encryption and decryption devices,
- path is a sequence of bits and path.length is the number of bits in path.

An algorithm for computing the keyTree() function is described below using psuedocode:

```
define keyTree(K_root, path):
    R = K_root
    for i = 0 upto path.len-1
        j = bit i of path     # i=0 is leftmost bit (MSB) of path
        R = f_j(R)
    endfor
    return R
```

The security of the keyTree construction follows from the fact that each non-final key in the key tree is involved in at most three operations: it is created from its parent key using either $f_0()$ or $f_1()$, and it can be an input to both $f_0()$ and $f_1()$. Since each of these operations can leak at most a small number of bits about the key, the overall entropy of the key remains high.

To encrypt a messages, instead of directly encrypting with the root key, the encrypting device uses the keyTree() operation to derive a message key $K_{message}$. To do this, the encrypting device chooses a nonrepeating message ID for the message, e.g. as bits from a random number generator, a counter value, etc. $K_{messsage}$ is then computed as:

Set path = 0||messageID, where || denotes concatenation

$K_{message}$ = keyTree(K_{root}, path)

This is the only use of K_{root}, and since its used only in the KeyTree construction.

The process of authenticating the ciphertext must also be protected from DPA, otherwise an attacker could submit many invalid ciphertexts to the decryption device and collect enough data for a side channel attack. For the encrypted message, for efficiency purposes we first use a hash chaining technique (described in the next section) to protect the body of the encrypted message from modification attacks, and place the first hash of this (reverse) chain into a non-secret header. This header itself is protected from modification attacks using a DPA-resistant MAC, called the validator which is computed as:

path = 1|| hash(header)

validator = keyTree($K_{message}$, path).

This computation is protected from DPA attacks since the computation of hash(header) uses no secret data, and the structure of the key tree operation ensures that $K_{message}$ and all intermediate keys derived from it in the key tree remain protected, no matter how many times the keyTree() function is invoked with different, modified headers.

Hash Chaining

To encrypt the bulk contents of a message, we use an ordinary block cipher chaining mode, such as CBC, except that we change the key every `nblocks` cipher blocks. The encrypting device chose message ID, thereby ensuring that a given $K_{message}$ will only be used for one message. The decrypting device, however, must additionally make sure to validate each block of the message before decrypting it, since the attacker could modify the ciphertext. This verification must be done without storing the entire message, since the decrypting device may have limited memory. To enable this, the encrypting device divides the message into *stripes* of length stripeLen – hashLen, where stripeLen is the amount of data the decryption device can buffer and is a multiple the block size. After encrypting the stripes, the encrypting device adds to each stripe the hash of the *subsequent* stripe. (This is done in reverse order, i.e. first computing the hash of the last stripe for inclusion in the next-to-last stripe.) The hash of the first stripe is included in the message header.

Before decrypting a stripe, the decryption device can check that its hash matches the one specified in the previous stripe (or in the message header, for the first stripe). This computation is done on the ciphertext, so it uses no secret information. The hash check is done before the stripe data is decrypted, so that only authenticated messages ever reach the decryption stage. Because each authentic message has a different messageID and thus a different $K_{message}$, the attacker can only cause decryption of one message for a given messageID.

Message header

Each message has a header. The format of the header is an implementation detail, but typically contains at least the protocol version, the length of the message, the messageID and the hash of the first stripe.

Encryption and Decryption Processes

Figure 4 illustrates the encryption process and Figure 5 illustrates the decryption process using the building blocks described earlier:

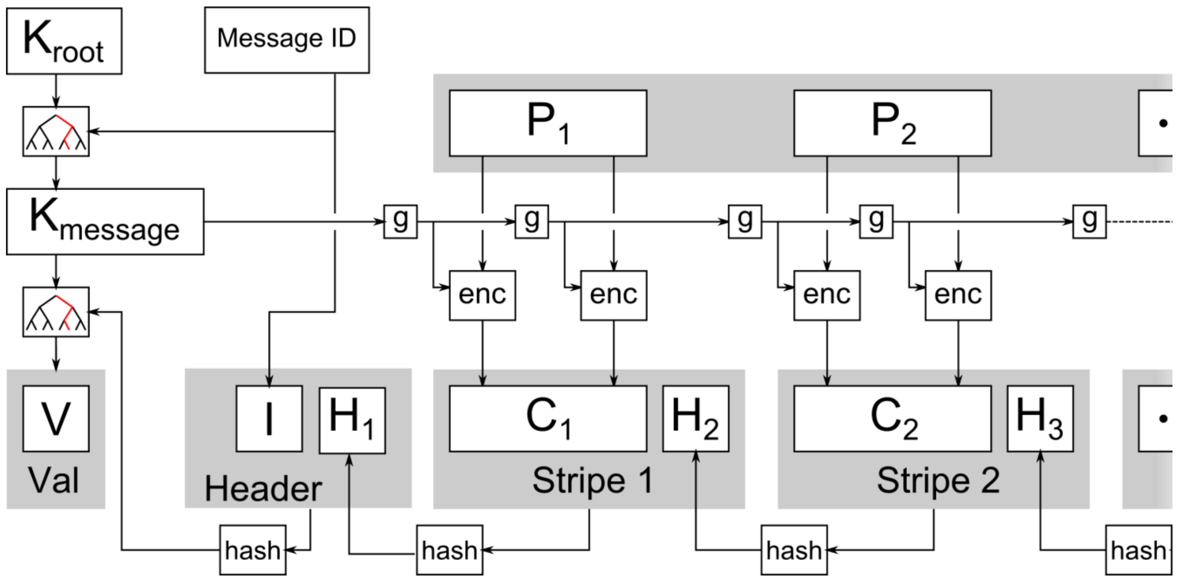


Figure 4: Leakage resistant encryption protocol

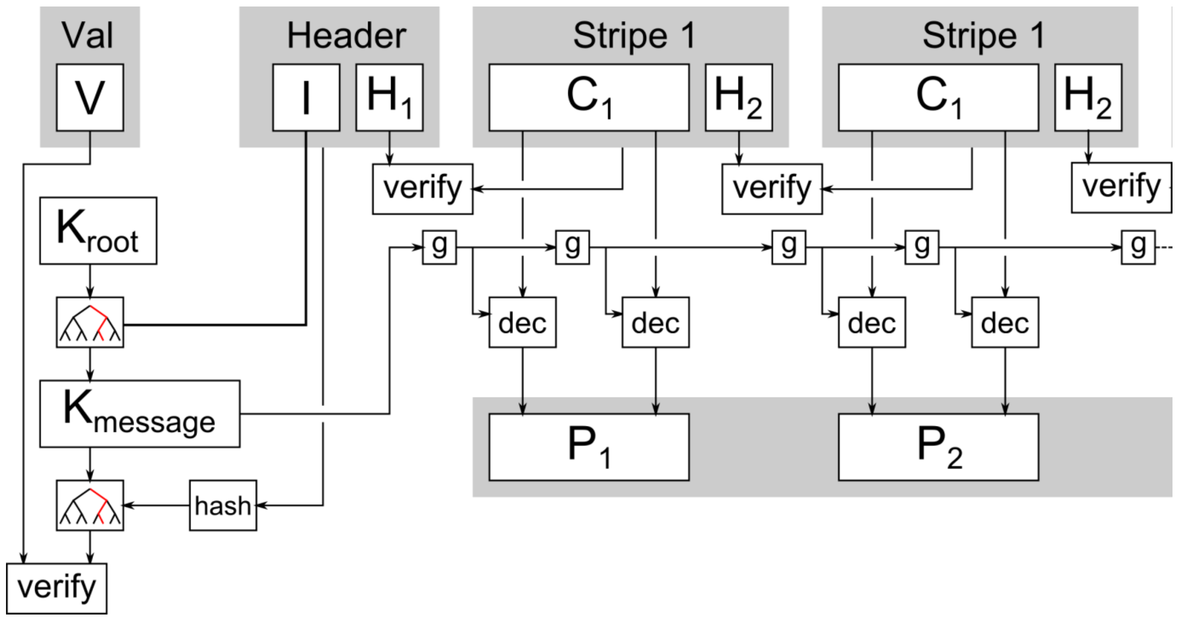


Figure 5: Leakage resistant decryption protocol

Conclusion

This construction illustrates how protocol level countermeasures can be used to provide efficient and low-overhead, side-channel resistant implementations of core security functions. Such constructions provide practical security against side-channel attacks and are well suited to military and aerospace applications, where high levels of assurance are required.

Patent Disclaimer

Cryptography Research discovered SPA and DPA in the mid 1990's and holds fundamental patents on countermeasures against such attacks, including protocol based methods such as those described in this article.

References

[1] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, Pankaj Rohatgi: *Introduction to differential power analysis*. Journal of Cryptographic Engineering 1(1): 5-27 (2011). Also available at: http://www.cryptography.com/public/pdf/IntroToDPA.pdf

Contact Information

For more information about this article contact:

Pankaj Rohatgi, Director, Hardware Security Solutions, Cryptography Research, Inc.

pankaj.rohatgi@cryptography.com