

# Secure Application Programming in the Presence of Side Channel Attacks

Marc Witteman and Martijn Oostdijk

Riscure, The Netherlands

{witteman, oostdijk}@riscure.com

**Abstract:** Side channel attacks can reveal secrets during program execution, or change the behaviour of a program. Without profound knowledge of these attacks it is hard to defend code effectively. Whereas traditional secure programming methods focus mostly on input validation and output control, side channel security requires pervasive protection throughout the code. This paper introduces a collection of secure programming patterns for security critical devices. These patterns help developers to mitigate the risk of side channel attacks.

## 1 Introduction

While patterns and guidelines for programming, and especially secure programming, have existed for a long time [11,12], most of the patterns used in practice focus on logical vulnerabilities such as lack of input checking, or promote maintainable code writing [3]. This paper discusses secure programming patterns (or: secure programming guidelines) intended to harden embedded software against side channel attacks.

Side channel threats are different from logical threats. They are more pervasive, and hardening code against side channel attacks requires a different way of viewing one's source code. At any point in time during program execution, a side channel attack can succeed in changing program behaviour or in identifying secret information. Defensive measures therefore need to be implemented throughout the code.

Smart cards are typically designed to run in a hostile environment and these have been threatened by side channel attacks for over ten years. Smart cards use effective countermeasures against these attacks. Protection against side channel threats also becomes relevant for other devices with the increase of secure functions in other devices. Examples are mobile phones, set-top boxes, printers, payment terminals and medical equipment. In addition, with the level of logical security rising in most (consumer) devices, we expect that adversaries will more and more start using side channels to attack a device.

So far, there has been little guidance to application developers in this field. Effective protection requires a mix of countermeasures at hardware, operating system and application level. In this paper, we focus on the application level.

Programming patterns provide solutions to common problems rather than defining a methodology for solving all problems in one domain. We formulate our patterns in a generic manner so that they hold for any language or operating system. In the accompanying coding examples, we use C and Java to illustrate the pattern.

Manufacturers in the smart card field as well as technology-licensing companies have developed countermeasures for hardware and cryptographic implementations. Some countermeasures are proprietary to these companies and are not publicly known. All information and concepts in this paper are based on information that is commonly and publicly known amongst side channel specialists.

The remainder of this paper consists of the following. First, an introduction to side channel attacks is given in Section 2. Then, a number of leakage patterns are presented, dealing with the aspect of *confidentiality*. Next, a number of fault injection patterns are presented dealing with the aspect of *integrity*. Finally, the paper discusses application of the patterns in Section 5 and draws some conclusions in Section 6.

## 2 Side channel attacks

Side channel attacks are well-known in the smart card community, and less known outside of it. Most developers have however heard of timing attacks. The report in 2003 by Boneh and Brumley of a timing attack on SSL over the network [1] is an example of an attack that received widespread attention in the developer and security community. The principle of a side channel attack is as follows. Rather than attacking the system via the regular input/output, an adversary abuses an unintended communication channel to attack the system. Hence the term side channel is used. In most cases, this requires that the adversary has physical access to the system. Consumer devices with a security function, such as smart cards, set-top-boxes, mobile phones and access control tokens, are therefore ideal candidates for an adversary with side channel attack skills.

Examples of commonly abused side channels include:

- Time: the time needed to complete certain operations.
- Power: the power available to and used by a device.
- Electro-magnetic radiation: EM radiation produced by a device.

These channels can be abused via two methods:

- Side channel analysis (i.e. data leakage): the attacker passively listens in on one of the side channels in the hope that some sensitive data leaks.

- Fault injection: the attacker introduces faults to change the behaviour of the device by actively manipulating the side channel.

## 2.1 Data leakage

Side channel analysis involves several techniques that may be applicable depending on the nature of the program running in the device. The two most common techniques are:

- Simple analysis: The attacker observes the side channel to see if confidential information is directly visible from the measurements. It may be possible to distinguish instructions or even a representation of data values. Optionally several traces (i.e. measurements) are averaged to get a better signal to noise ratio. Depending on the measured side channel, this is called Simple Power Analysis (SPA) or Simple Electro-Magnetic Analysis (SEMA).
- Differential analysis: The attacker typically collects a large number of traces (i.e. >1000) and uses statistical analysis to demonstrate internal relationships through correlation. This information is subsequently used to derive secrets. Depending on the measured side channel, this is called Differential Power Analysis (DPA) or Differential Electro-Magnetic Analysis (DEMA).

The picture below shows the power profile of a weak RSA implementation. The short and long intervals correspond to modular square and multiply operations. The distinction of these operations allows full key retrieval. This attack is an instance of SPA.

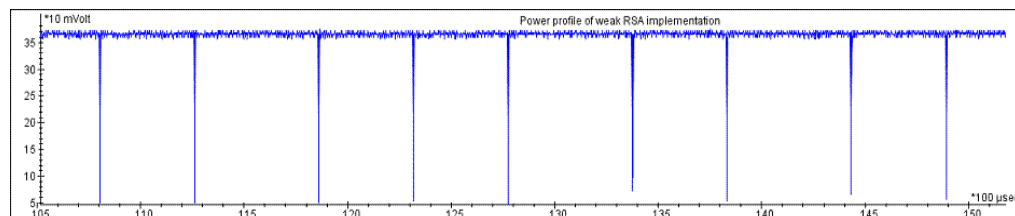


Figure 1: Example of a trace measured during an RSA execution

In differential analysis [6,8] the attacker uses statistical differences between large numbers of samples in order to retrieve secrets. This approach can be successful when the individual samples leak too little information to allow SPA. The method requires the acquisition of a number of traces. Subsequently the traces are partitioned in two different groups, depending on one bit in the expected intermediate data. Taking the average of the traces within each group and subtracting them yields a differential trace. This differential trace will show a peak where the intermediate data is processed. If the intermediate data is known it will show where and how the data is used. The attack gets more interesting when applied to unknown intermediate data that is related to a few key bits. In that case it may be possible to guess the intermediate data if the number of involved key bits is

relatively small. By computation and comparison of various differential traces (for each value of the key bits) it is possible to identify the correct value of the key bits.

The picture below contains a few differential traces demonstrating how input data bits correlate to power traces before (i.e. the first positive peak), and after mixing with the key (i.e. the second peak). The inversion of the second peak for bit 40 and 42 indicates that the corresponding key bits are set to 1. Key bits 41 and 43 must be 0 as the peak is not inverted. This attack is an instance of DPA.

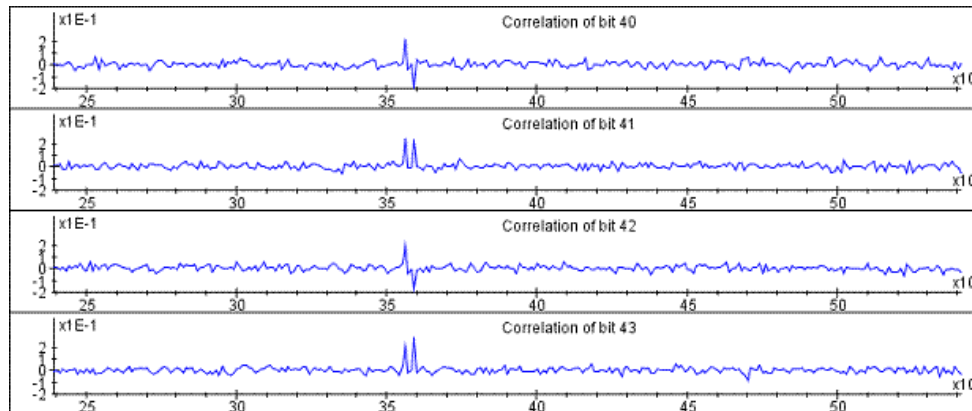


Figure 2 Example of multiple traces in DPA which reveal the key bits of an RSA secret key

## 2.2 Fault injection

With fault injection, the attacker's objective is to change a critical value or to change the flow of a program. This then allows him to skip a digital signature verification, load unauthorised firmware or jump over the increase of a security counter. The "unlooper" device that is used in cases of Pay-TV piracy makes use of a fault injection technique (Figure 3).

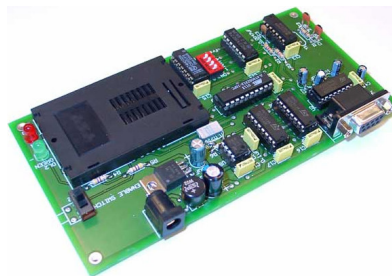


Figure 3: An "unlooper" device makes use of fault injection (US\$ 100)

Faults can be injected in several ways:

- Power glitches can disturb the power supply to the processor, resulting in wrong values read from memory.

- Optical glitches with laser light can force any elementary circuit to switch, conceptually enabling the attacker to achieve a very specific change of data values or behaviour.
- Clock manipulation by introducing a few very short clock cycles which may lead to the device misinterpreting a value read from memory.
- Interruption by cutting the power to the processor while it is performing important computations, hoping to either prevent the system taking measures against a detected attack or get the system into a vulnerable state when the power is back.

A slightly different technique which falls in the same attack family is Differential Fault Analysis (DFA). In DFA, the attacker reveals a cryptographic key by comparing the results of several runs of the cryptographic algorithm with injected faults and without injected faults.

### ***2.3 Used in combination***

Combinations of side channel analysis and fault injection attacks are possible and sometimes necessary to effectively exploit a vulnerability. For example, one may use time analysis to gain information about the time the processor takes to verify the digits of a PIN and interruption to prevent the processor from decreasing the “tries counter” in case of a wrong PIN. Sometimes it is possible to get the cryptographic key by passively listening in during a normal encryption in combination with listening in on an encryption where some random faults were injected during the crypto algorithm.

### ***2.4 In perspective***

Some side channel attacks are straightforward and cheap in equipment (< \$2,000) to perform. Others are complex and require expensive equipment (> \$50,000). This has an influence on the business case of the attacker; a device may need protection against the easy attacks, but may not require protection against the high-end attacks. It is outside the scope of this paper to discuss the differences between basic device protection and advanced device protection.

Further, although based on the security-by-obscurity principle, the complexity of many devices make an attack that is easy in concept still hard to execute, even when no side channel countermeasures are present. This aspect should also be taken into account when determining how much side channel protection a specific device requires.

## **3 Patterns to defend against data leakage**

The patterns in this section assist to prevent application program code from leaking sensitive information when this information is processed. Application developers can use

these patterns to protect confidential data like keys and passwords, but also to hide sensitive decisions. We silently assume that sufficient attention is given to protection measures against side channel analysis in the hardware and operating system. The seven patterns for leakage are shown in Figure 4. Using two examples, the arrows represent the following relationships:

- The pattern BRANCH serves to prevent leakage. We refer to it as LEAKAGE.BRANCH.
- The pattern KEY.INTEGRITY is a specific pattern that is derived from the broader KEY.ACCESS pattern. Both patterns serve to prevent leakage.

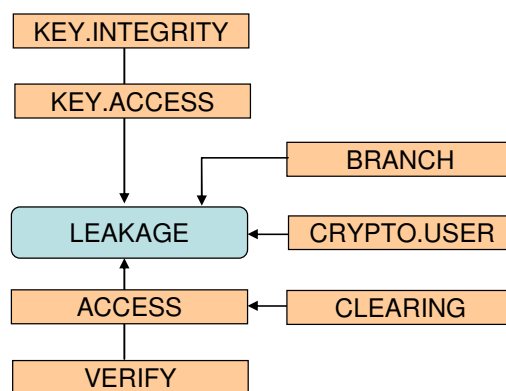


Figure 4: Data leakage patterns

### 3.1 LEAKAGE.KEY.ACCESS

**Context** Keys consist of key material which, when handled by the application (e.g. reading / copying / verifying), is prone to leakage.

**Solution** The application should at most access pointers to keys, but not their actual values. Handling of key values shall be done by dedicated crypto libraries that have proven side channel resistance (also see LEAKAGE.CRYPTO.USER).

### 3.2 LEAKAGE.KEY.INTEGRITY

**Context** Application access to plaintext key values with the purpose of checking integrity (e.g. parity) of the key material for integrity purposes is prone to leakage.

**Solution** Use time-constant verification methods and use unpredictable ordering in key byte tests. Further, avoid key integrity checking when not strictly needed.

Negative example: The behaviour of the following code depends on the key value.

```
public static boolean checkParity ( byte[]key, int offset) {
    for (int i = 0; i < DES_KEY_LEN; i++) { // for all key bytes
        byte keyByte = key[i + offset];
        int count = 0;
        while (keyByte != 0) { // loop till no '1' bits left
            if ((keyByte & 0x01) != 0) {
                count++; // increment for every '1' bit
            }
            keyByte >>= 1; // shift right
        }
        if ((count & 1) == 0) { // not odd
            return false; // parity not adjusted
        }
    }
    return true; // all bytes were odd
}
```

The conditional statements are vulnerable to SPA and reveal the individual key bits. The picture below shows the power profile. One can see that the key bits are verified LSB to MSB and that testing bits set to 1 takes more time that testing bits set to 0.

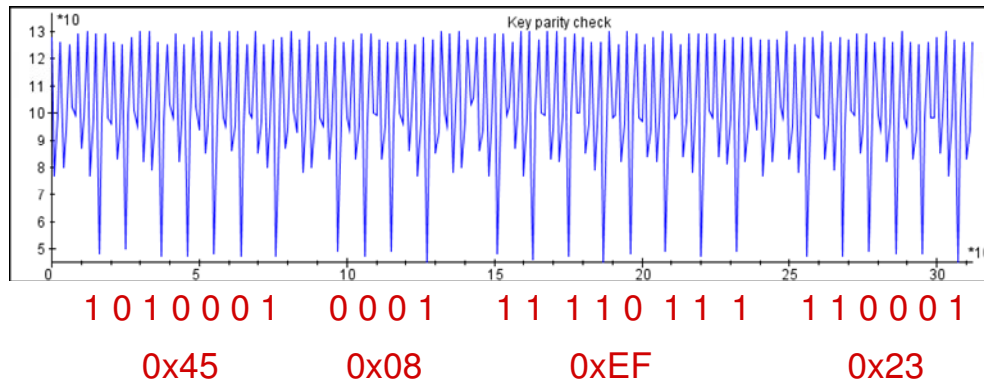


Figure 5: Power trace<sup>1</sup> of a vulnerable key parity check

Positive example: use a parity table that is consulted in an unpredictable order.

```
static byte odd_parity[]= {
    // each table entry represents odd parity of index
    1, 1, 2, 2, 4, 4, 7, 7, 8, 8, 11, 11, ..
    .. // intermediate data removed
    .., 248, 248, 251, 251, 253, 253, 254, 254};

public static boolean checkParity ( byte[]key, int offset) {
    int r = random.nextInt() & 7; // random number 0..7
    for ( int i=0, j = r; i<8; i++, j = (j+1)&7 ) {
        // for all key bytes
        if (key[j] != odd_parity[key[j+offset]]) {
            return false; // parity incorrect
        }
    }
}
```

<sup>1</sup> Generated with our side channel test platform called Inspector

```
}  
    return true; // all bytes were odd  
}
```

### 3.3 LEAKAGE.BRANCH

**Context** Program branches (e.g. if statements or switches) may be prone to side channel leakage.

**Solution** Avoid using branches for confidential decisions. Confidential program alternatives should be coded in a time-constant manner, preferably in identical instruction sequence.

This pattern can for instance be applied when an application performs multiple access checks but, in case of failure, does not want to inform a user why access was denied. In that case the program flow should not leak the results of the individual checks.

The example code below shows how one sensitive condition (i.e. the result of `checkPin()`) can be transformed into two conditions, each of which individually is non-sensitive. None of the branches directly relates to the `checkPin` result.

```
Random.generateData( randomByte, (short)0, (short)1 );  
boolean randomChoice = randomByte[0] < 0;  
byte mask = 0;  
if (randomChoice) // unpredictable  
    mask = 0x4;  
else  
    mask = 0x20;  
if (checkPin() == randomChoice)  
    // depends on checkPin and randomChoice  
    mask ^= (byte)0x4;  
else  
    mask ^= (byte)0x20;
```

When the result of `checkPin()` is true the variable `mask` is set to `0x00`, otherwise it is set to `0x24`. Following the `checkPin` step several other checks could be done similarly without SPA leakage. Eventually a branch is needed to compare this variable, yet to an attacker this branch is much harder to correlate to the individual checks performed.

### 3.4 LEAKAGE.CRYPTO.USER

**Context** The implementation of a cryptographic algorithm is sensitive to side channel leakage and complex to implement well.

**Solution** Do not implement encryption at the application level, but only use the crypto library from a lower layer. This assumes that the crypto library is resistant against side channel analysis. Although this is often not (yet) the case, the

developer of a crypto library should be better equipped to implement proper protection measures.

### 3.5 LEAKAGE.ACCESS

- Context** Accessing (i.e. reading or writing) confidential array values may expose confidential data through differential side channel analysis.
- Solution** Accessing confidential array values shall not be done in a zero-offset sequential manner (e.g. from left to right). Instead, choose (dynamically) an offset and traverse the array starting at that offset modulo the length of the array.

A negative example of this pattern is:

```
memcpy( pin, buffer, 4 ); // copy a PIN code to a buffer
```

The better way of doing this would be:

```
for (int i = 0, j = (rand() & 3); i < 4; i++, j = ((j+1) & 3))  
buffer[j] = pin[j]; // start at a random index in the range 0..3
```

### 3.6 LEAKAGE.VERIFY

- Context** Verification of secrets like passwords and PIN codes shall not use a zero-offset sequential comparison (e.g. from left to right) and fail with the detection of a wrong character or digit. The duration of the verification then reveals the index of the character. This applies for instance to verification of authentication data.
- Solution** Do not use sequential methods like `memcmp` or `strcmp` to verify secret array values. Use a mechanism that compares the entire data before completion. Furthermore, consider to choose an unpredictable offset and traverse the array starting at that offset modulo the length of the array (see LEAKAGE.ACCESS). Even better, the comparison could be done with an encrypted secret instead of a plaintext secret.

Negative example: the following code is sensitive to a timing attack as its duration reveals the index where the comparison fails, and sensitive to SPA as the power consumption is dependent on individual confidential array values.

```
if ( strcmp( givenPasswd, storedPasswd ) != 0 ) return -1;
```

A better solution that would return only after comparing the complete password follows.

```

char* c1 = givenPasswd;
char* c2 = storedPasswd;
char error = 0;
for ( ; *c1 != 0 && *c2 != 0; c1++, c2++ ) // loop
    error |= *c1 ^ *c2; // collect diff in error
if (error | *c1 | *c2) // fail if any not zero
    return -1;
return 0;

```

Note that this solution could be further improved with a randomised offset.

### 3.7 LEAKAGE.CLEARING

**Context** Clearing confidential array values shall not be done in a zero-offset sequential manner (e.g. from left to right) as clearing the data may expose the confidential data through differential side channel analysis.

**Solution** Do not use sequential methods to erase confidential array values. This threat is commonly known for storing confidential data, and equally applies to clearing this data. An optimal approach would be to overwrite the array with random data first, and to clear it afterwards.

## 4 Patterns to defend against fault injection

The patterns in this section assist in preventing fault injection in application program code and in responding to it. These patterns protect critical data or program flow. The arrows between the patterns have the same meaning as the ones between the leakage patterns described in the previous chapter.

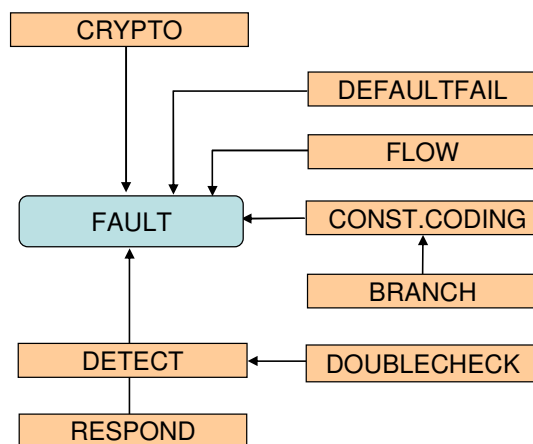


Figure 6: Fault injection patterns

## 4.1 **FAULT.CRYPTO**

- Context** Cryptographic algorithms are sensitive to fault injection. They may even reveal key data through the return of false encryptions due to fault injection. Differential fault analysis (DFA) is a good example of this.
- Solution** Check for fault injection during or after crypto. Verify any ciphered data before transmission by deciphering or repeated enciphering. If the deciphered data matches the original data to be ciphered, or the repeated enciphering matches the original result, it is most likely that the encryption was not corrupted.

## 4.2 **FAULT.DEFAULTFAIL**

- Context** If a parameter can only take a limited number of values it is tempting to use the default case in a switch (or the else part of a cascaded if-else-if construct) for dealing with the last possible (and valid) value without checking. This is prone to fault injection because this alternative is easily selected as a result of potential data manipulation.
- Solution** Do not make this assumption. When checking conditions (switch or if) check all possible cases and fail by default. Also a final else statement in an if-else construct should lead to a fail.

The positive example below does not assume a valid option for the default case.

```
switch (state) {
    case STATE_INIT: processInit(apdu); break;
    case STATE_PERSO: processPerso(apdu); break;
    case STATE_ISSUED: processIssued(apdu); break;
    case STATE_LOCKED: processLocked(apdu); break;
    default: fail();
}
```

## 4.3 **FAULT.FLOW**

- Context** Fault injection can hit the program counter or stack. This can result in “code hijacking”, i.e. unauthorized jumps to privileged code. Verification of the correct flow should be done during the execution of sensitive code.
- Solution** Use a shadow stack and program counter that keeps track of the correctness of the current execution path. Check these shadow parameters after branches.

The positive example below shows how a shadow stack could work.

```

void mySensitiveMethod( . . . ) {
    . . .
    // increase virtual program counter by one
    increaseShadow( MY_SENSITIVE_METHOD );
    . . .
    // increase virtual program counter by one
    increaseShadow( MY_SENSITIVE_METHOD );
    . . .
}

. . .
void someCode() {
    push( MY_SENSITIVE_METHOD ); // push the next method
    // on the shadow stack
    mySensitiveMethod( arguments );
    // pop the shadow stack and check that we indeed return from
    // the expected method and our virtual program counter
    // was increased twice
    pop( MY_SENSITIVE_METHOD, 2 );
}

```

#### 4.4 FAULT.CONSTANT.CODING

**Context** Sensitive data carrying a limited set of values (like phase and state variables) may be manipulated by fault injection attacks if they use trivial constant coding (e.g. 0, 1, 0xFF).

**Solution** Do not use trivial constants for sensitive data. These constants should use non-trivial values that are unlikely to be set through fault injection.

Positive example:

```

static final byte STATE_INIT = (byte)0x3C;
static final byte STATE_PERSO = (byte)0xC3;
static final byte STATE_ISSUED = (byte)0x5A;
static final byte STATE_LOCKED = (byte)0xA5;

```

Note that choosing the values at maximal hamming distance (number of bits that have different value) makes it difficult for an attacker to change one valid value to a different valid value.

#### 4.5 FAULT.BRANCH

**Context** Boolean values can be manipulated by fault injection attacks.

**Solution** Do not use booleans for sensitive decisions. Fault injection typically changes actual values to simple values, like 0 or 1. Non-trivial numerical values are more difficult to set by fault injection. Sensitive choices should therefore not be coded as a boolean value, but rather as a non-trivial

numerical value.

Positive example:

```
if (conditionalValue == (byte)0xA5)
    // then part
else if (conditionalValue == (byte)0xC3)
    // else part
```

#### 4.6 **FAULT.DETECT**

**Context** Sensitive data may be manipulated by a fault injection attack at any time during program execution.

**Solution** Verify sensitive data. Sensitive data can for instance be protected by a checksum. Data protected in this way should be verified at regular intervals. Ideally the integrity of sensitive data should be verified each time when used. For convenience, data can be encapsulated in so-called security controlled objects that have their own methods to preserve integrity.

Positive example:

```
byte result = SOME_VALUE;
byte resultChecksum = ~SOME_VALUE; // create checksum
...
if ((result ^ resultChecksum) != 0xFF) fail(); // verify checksum
```

#### 4.7 **FAULT.DOUBLECHECK**

**Context** Sensitive data may be manipulated by fault injection attacks. When a decision is made upon a single test the decision may be corrupted.

**Solution** Double check sensitive conditions. A conditional process based on sensitive data should double check the data. Preferably these checks should not be identical, but complementary as the attacker will have to perform two different types of attack. An unpredictable time distance between the two tests would further increase the security.

Positive example:

```
// within then part
if (conditionalValue != (byte)0xA5)
    fail(); // fail if not equal to 0xA5
if (~conditionalValue != (byte)0x5A)
    fail(); // fail if complement not equal to 0x5A
```

## 4.8 FAULT.RESPOND

**Context** Immunity to fault injection attacks is hard to achieve and a good response to an attack can further protect the device.

**Solution** Monitor and respond to fault injection attacks. Monitoring can be done by repeatedly checking known data for changes (see FAULT.DETECT). The defence could consist of incident logging and temporarily or permanently disabling functionality. For instance, one can wipe the secret key upon detecting a fault attack.

Note that it can help to track changes to *known* data, not just changes to *sensitive* data as is stated in the FAULT.DETECT pattern. This way, it is possible to detect and respond to an attack before unauthorized actions are performed by the attacker.

An example that performs the checks:

```
static final byte[] CONSTANT = { 0x0F, 0x1E, 0x2D, 0x3C, 0x4B, 0x5A,
0x69, 0x78 };
static byte[] copied = new byte[CONSTANT.length];

public void main( Strings[] s ) { // copy at begin of program
    System.arraycopy( CONSTANT, 0, copied, 0, CONSTANT.LENGTH );
}

private void check() { // call this method repeatedly
    for (int i = 0; i < CONSTANT.LENGTH; i++) {
        if (CONSTANT [i] != copied[i]) {
            fail();
        }
    }
}
```

## 5 How to apply the patterns

Before using the patterns a developer should assess the risk of side channel attacks to his device. Some of the main aspects to consider are:

- Is there a business case for attacking the device?
- Are side channel threats the low hanging fruit, or are other areas in the device weaker and more likely to get attacked first?
- Can an attacker get physical access to the device and control or observe its relevant interfaces?

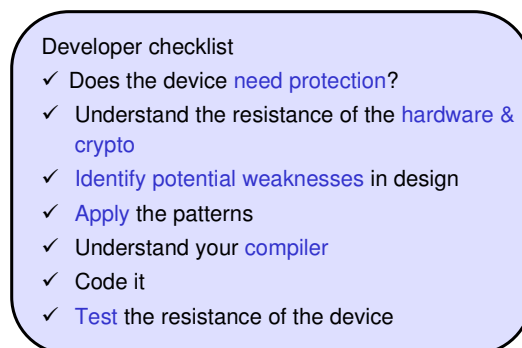
When these questions are answered positively, the next step is to gain understanding on the resistance of the hardware and the operating system, including the cryptography that you are planning on using in your application. When these lower layers have strong countermeasures implemented, the application will benefit from this. When these lower layers do not have countermeasures implemented, you need to prioritise what patterns are

best to apply. For instance, optimising a key-parity-check implementation in your application does not add much value if the underlying cryptographic algorithm already leaks the secret key. On the other hand, implementing the pattern for PIN code verification will still add value for many applications.

Next, you identify the potential weaknesses in the application design. This involves finding critical decision points in the application and operations that access sensitive information. The patterns can assist in finding the weaknesses. Potential weaknesses for data leakage are the places in which a secret - and data that depends on a secret - is accessed, used or verified. Potential weaknesses for fault injection are all places where you make a security-critical decision such as access control decisions and branches with if or switch statements, and where you manipulate sensitive data such as state and key-dependent data.

The next step is to apply the patterns. In this activity, it is important to understand the compiler. Nowadays, compilers are intelligent and they can undo side channel countermeasures for performance reasons. For instance, if a compiler identifies a double verification of a variable, it is likely to reduce this to one at machine code level. By understanding how your compiler deals with the patterns, you can adapt the coding if necessary. It is only at this stage, that the application should be coded. In practice, you are likely to iterate back and forth the previous steps a few times, also depending on the software development approach that is followed.

The final step is to test the resistance of the device against the side channel attacks that it should protect against. Since the overall resistance of the device depends on so many aspects, including the physical features of the hardware, it is important that the final product is tested rather than evaluating only the design and source code. Dedicated side channel test tools can assist developers and security evaluators in this.

- 
- Developer checklist
- ✓ Does the device **need protection**?
  - ✓ Understand the resistance of the **hardware & crypto**
  - ✓ **Identify potential weaknesses** in design
  - ✓ **Apply** the patterns
  - ✓ Understand your **compiler**
  - ✓ Code it
  - ✓ **Test** the resistance of the device

*Figure 7: Device developer checklist for application development*

With respect to the effectiveness of these patterns the following considerations apply:

- Be aware that side channel resistance needs to be addressed also at hardware and operating system level. Weaknesses at any level can result in exploitable vulnerabilities.
- Most patterns proposed in this paper introduce penalties in performance, size and maintainability of code. Also note that efficient compilers may undo some of the code improvements suggested by the patterns.
- Application of the patterns still requires a decent understanding of side channel issues, and can be rather error prone and laborious. For the far future, a tool for semi-automated inspection of source code and a compiler that adds side channel countermeasures to code rather than undoing these could be helpful. However, due to the complexity of the matter the development of these will take significant research effort.

We inspected some open source software, such as the key-parity checking in GnuCrypto (see Figure 5) and OpenSSL, to see to what extent the code is sensitive to side channel threats and whether it could be improved with our patterns. In all open source software that we have come across in the past years, side channel protection is generally absent, and several concepts could benefit from the patterns in this paper.

## 6 Conclusion

Side channel attacks form a well-defined domain which is understood and addressed in the smart card context, but which has received limited attention outside of this domain. For device application developers, there is little guidance available on how to mitigate side channel risks. The patterns presented in this paper are based on a pragmatic approach and they can assist developers in implementing side channel protection on embedded devices.

## References & further reading

1. Boneh and Brumley, Timing attack on unprotected SSL implementations, 2005, ISBN:1-59593-226-7
2. Differential Power Analysis, [www.cryptography.com/resources/whitepapers/DPA.pdf](http://www.cryptography.com/resources/whitepapers/DPA.pdf)
3. Gamma, Erich et al., *Design Patterns*, Addison Wesley, 1994, ISBN 0-201-63361-2
4. GNU Crypto, [www.gnu.org/software/gnu-crypto](http://www.gnu.org/software/gnu-crypto)
5. Hubbers, Engelbert and Poll, Erik, Reasoning about Card Tears and Transactions in Java Card, *Proc. Fundamental Software Approaches to Software Engineering 2004*, LNCS 2948, 114-128, 2004
6. Kocher, Paul and Jaffe, Joshua, and Jun, Benjamin, Differential Power Analysis, *Proc. Advances in Cryptology*, LNCS 1666, 388 – 397, 1999

7. Kocher, Paul, Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems, Whitepaper available from <http://www.cryptography.com/resources/whitepapers/TimingAttacks.pdf>
8. Mangard, Stefan and Oswald, Elisabeth, Popp, Thomas, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, Springer Verlag, 2006
9. OpenSSL, [www.openssl.org](http://www.openssl.org)
10. Smart Card Alliance, <http://www.smartcardalliance.org>
11. Sommerlad, Peter, *Security Patterns - Integrating Security and Systems Engineering* (Wiley Series in Software Design Patterns) by, 2005
12. Steel, Christopher and Nagappan, Ramesh, and Lai, Ray, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*, Prentice Hall, 2004
13. Witteman, Marc, Advances in Smart card security, *Information Security Bulletin*, CHI Publishing, 11-22, July 2002.
14. Witteman, Marc, Java Card Security, *Information Security Bulletin*, Volume 8, CHI Publishing Ltd., 291-298, October 2003