What can software developers do about side-channel and fault injection attacks?

An introduction to physical security for microcontroller devices.

By Joseph Yiu, August-2023

Background

Today, many embedded systems are based on low-cost microcontrollers and many of them are connected to the Internet or have some form of connectivity. As you would expect, security is an important requirement for these systems. Because many embedded systems are built on chips containing Arm® processors, such as processors from the Cortex®-A and Cortex-M families, Arm have introduced TrustZone® Technology and the Trusted Firmware project to help system designers and software developers create secure products.

While TrustZone and Trusted Firmware can protect assets from a wide range of threats, such as software-based attacks or attacks via the debug infrastructure, there are several forms of attack that TrustZone cannot address. These include and are not limited to:

- Fault injection attacks: A form of attack that introduces fault(s) to the chip's operations. These attacks cause incorrect operations in the processor(s); for example, effectively skipping some of the instructions being executed. Some of the known methods include:
    - Voltage glitching – creating a glitch in the power supply to disturb the operation of the processor system.
    - Clock glitching – creating a glitch in the clock signal to disturb the operation of the processor system.
    - Using a Laser/Electromagnetic-probe to inject faults into the chips.
- Side channel attacks: Information about the software being executed could be leaked by means of patterns in the power consumed and/or by electromagnetic radiation. From the information collected, an attacker could guess the confidential information that the device is processing.

In general, these attacks are called *physical attacks* because they are dependent on the physical properties of the microcontrollers/SoC devices. Some chips are designed with physical protection features (often known as anti-tampering) to address such threats.

An overview of common side channel and physical attacks is available in the Low-Level Software Security Book [1]. This book is an on-going open-source project. To avoid duplication of effort I will not describe the details relating to side channel and physical attacks in this document.

This document provides an overview of security measures that software developers can use to make physical attacks more difficult. But first of all, I will explain the role of TrustZone technology and Trusted Firmware because there are often misunderstandings about the differences between physical protection and TrustZone.

What TrustZone is and isn't

In recent years, with increasing numbers of Cortex-M based chips being built with TrustZone technology, a number of researchers are demonstrating the use of physical attacks to bypass/circumvent TrustZone security.

TrustZone Technologies are available on Cortex-A and recent Cortex-M processors. Although at the detail level, TrustZone technologies in Cortex-A and Cortex-M are different, the following high-level concepts are similar:

- The software running on the processor is divided into Normal Application(s) and secure firmware.
- The processor executes secure firmware in a Secure state and Normal Application(s) executes in the Non-secure state.
- TrustZone provides a controlled way for the Secure firmware and Normal Applications to interact.
- Secure resources including secure memories can be accessed by secure software only.
- Secure firmware contains security sensitive codes and assets such as key management, authentication and firmware update support.
- The system starts in Secure state. Typically, Secure boot would be implemented in such systems to ensure that the program image being used is intact and hasn't been tampered with.

The advantages of having TrustZone is that if the software running in the Non-secure world has been attacked and compromised, the attacker would not have full control of the system because:

- The attacker cannot access secure information in the secure memories.
- The attacker cannot change the firmware because the firmware-update mechanism is protected.
- TrustZone prevents the attacker from bypassing the product's life cycle management (Only applies when the device supports device life-cycle management features). For example, the attacker cannot downgrade the firmware with an older version with vulnerability, or re-enable the debug access.

To develop secure systems, Secure firmware developers need to correctly use the TrustZone features. To assist software developers, Arm provides software guidelines. For example:

- Secure software guidelines for Armv8-M [2]
- Stack sealing and why it is needed in TrustZone for Armv8-M [3]
- KAN355 – Stack Sealing on Armv8-M [4]

Please note that in addition to TrustZone, there are additional security features in Arm processors. For example, the Armv8-M architecture, which Cortex-M23 and Cortex-M33 processors are based on, also supports stack limit detection and Memory Protection Units (MPUs), which are useful for the separation of privileged and unprivileged software.

TrustZone is well suited for protecting systems from remote attacks, which is where the majority of the attacks on IoT devices are from (e.g. via Internet or connectivity). In most cases, before an attacker can attack the secure firmware, the person must first attack the normal applications running in the Non-secure world. As a result, a successful compromise of a TrustZone enabled device needs to involve multiple types of attack, which means it is more difficult to attack a TrustZone enabled system than one without.

For systems that do not have connectivity, TrustZone can still be useful as follows:

- TrustZone can be used to protect the firmware update mechanism. When used together with a secure boot loader, users can only update the system with firmware images that are correctly signed with a cryptography operation.
- A device vendor can include software libraries in the Secure world of a TrustZone enabled device without releasing the source code. This arrangement allows their customers to use the libraries in the normal applications running in the Non-secure world.
- TrustZone supports debug authentication mechanisms, and this can be used to determine whether a person is allowed to set up a debug connection to the hardware. Because the debug permission for the Secure and Non-secure worlds are controlled separately, it is possible to enable debug access for normal applications but disallow debug access to the secure firmware.

However, TrustZone does not prevent physical attacks. For example:

- The execution of certain cryptography codes could have side channel leakage. Potentially, secure information being processed could be leaked through execution timing, power signature and electromagnetic signals radiated from the chip(s).
- With Secure and Non-secure codes sharing the same caches, it is possible, even though Non-secure code cannot directly see the Secure data in the caches, to exploit cache-based side channel leakage based on execution timing. If a remote attacker manages to compromise the Non-secure world and gains access to a high resolution timer, it is possible for the attacker to guess the Secure information.
- If a remote attacker manages to compromise the Non-secure world, they could be in a position to inject faults, as has been demonstrated with the RowHammer[5] and VoltJockey[6] attacks.
- Depending on the system design, some of the hardware resources in the chip can be shared between Secure and Non-secure worlds (e.g. some SRAM blocks can be shared). If there are other bus managers/bus controllers in the system that generate bus transactions at the same time as the processors, some of the bus transactions would be delayed by arbitration logic in the bus system, and this can lead to a timing side channel leakage. This type of timing delay is also known as resource contention.

TrustZone by itself does not protect the processor from fault injection attacks. For example, glitches in the supply voltage or in the clock source could result in incorrect operation(s) in the processor, which could lead to a security vulnerability. It is important to understand that TrustZone is not designed to handle physical attacks and thus additional security measures are needed. This does not mean that TrustZone is ineffective, it is just that we need different types of security measures to address different types of security threats. Some microcontrollers include certain levels of physical protection features, but there is always a trade-off between security and other product requirements, e.g. power and cost. TrustZone can be used in conjunction with other security features and software measures to provide a holistic range of protection capabilities.

Is it possible to add physical protection features to an Arm processor? The answer is yes. Arm SecurCore® and Cortex-M35P processors provide a range of physical protection features (see "Additional

Information" at the end of this paper). However, many of these features can increase the silicon area, the power consumption and can impact performance. At the same time, many applications do not require physical protection and many chip vendors will not be happy to pay for the design and implementation cost for those physical protection features. As a result, physical protection features are often available only in specialized secure processors.

Many recent Arm embedded processors, such as the Cortex-M55 and Cortex-M85, offer optional features designed for automotive functional safety, such as Dual Core Lockstep, Transient Fault Protection and SRAM ECC (Error Correction Code). These features provide strong protection against physical fault injection attacks, but also have an impact on silicon area and power consumption.

About Trusted Firmware-M

To utilize TrustZone Technology, you need to have secure firmware running in the Secure world. Secure firmware provides security services to the applications as well as handling the secure boot, the firmware update process and the debug authentication. Creating secure firmware from scratch is a massive undertaking and requires expertise in many fields. There are commercial solutions available in the market, but due to price pressure, some IoT product developers might find it challenging to adopt some of these commercial solutions.

To enable the ecosystem to utilize TrustZone and to encourage the deployment of security best practices as outlined by the Platform Security Architecture (PSA), Arm started the Trusted Firmware project which provides reference implementation of secure firmware based on APIs defined by PSA Certified™ (https://www.psacertified.org/).

The Trusted Firmware project supports both Cortex-A processors (Trusted Firmware-A) and Cortex-M processors (Trusted Firmware-M). The project is open source, and the source codes can be accessed from the following locations:

Project	Location
Trusted Firmware-A	https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/tree/
Trusted Firmware-M	https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/

Because many IoT devices are based on low-cost Cortex-M based microcontrollers, Trusted Firmware-M (TF-M) is one of the key enablement activities enabling the IoT industry to create security solutions. TF-M supports a range of features including:

- Cryptography APIs
- Initial Attestation APIs
- Secure Storage APIs
- Firmware Update APIs
- Secure Debug based on PSA Authenticated Debug Access Control.

(Note: There are additional APIs involved in the TF-M software framework)

TF-M provides limited physical attack mitigation – a part of the mitigation is a small Fault-Injection Hardening (FIH) function inside the TF-M. More information can be found here:

FIH resource	Location
Document	https://tf-m-user-guide.trustedfirmware.org/design_docs/tfm_physical_attack_mitigation.html https://www.trustedfirmware.org/docs/TF-M_fault_injection_mitigation.pdf
Document for FIH	https://tf-m-user-guide.trustedfirmware.org/design_docs/tfm_physical_attack_mitigation.html#tf-m-countermeasures-against-physical-attacks
FIH library code	https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/lib/fih/
Mbed™TLS crypto services	Scope of the MbedTLS crypto (which is used as Crypto services within TF-M) is documented here: https://github.com/Mbed-TLS/mbedtls/blob/development/SECURITY.md Currently MbedTLS provides limited protection on timing attacks and no guarantees on non-timing side channel or fault injection attacks.

The FIH library code provides mitigation against simple single fault injection attacks. However, if the attacker was able to inject multiple faults during the execution of the software, then the FIH mitigation by itself is insufficient and additional countermeasures are required.

Defining the security needs of your project

When creating an embedded system for a product that would be widely used or is security critical, it is important to have a clear scope of what security features are required. This affects many aspects of the project, for example:

- the microcontroller device being used,
- the security setup in the software,
- the security certification requirement(s).

A common approach to start the definition of security needs is to define a threat model. Example documents of threat model can be found in the PSA Certified website:
https://www.psacertified.org/development-resources/building-in-security/threat-models/

After defining a threat model, you can then decide if physical protection features are required. Please note that many physical protection features can increase the power consumption of the system, the product cost and potentially reduce the performance of the processor system. To get the right balance, system designers and software developers need to know how to define the security needs of their products/projects from the threat model. This is dependent upon a range of factors such as:

- What is the value of the secret information stored on the system. E.g. if the product does not contain confidential information, or only confidential information of limited value (e.g. a Bluetooth pairing key for a low-cost consumer device), then physical protection could be overkill.
- Whether the attackers have physical access or can be in close proximity to the system.
- What are the impacts if the device is non-responsive.

- What are the impacts if the device is compromised.
- Other factors like legal requirements, potential reputational damage.

While many side channel attacks require physical access or close proximity to the device, it should be noted that timing side channel leakages could be monitored remotely if the attackers have managed to inject codes and then execute them on the device(s).

During software development, we need to define what should be protected. For example:

- What information is secret that needs to be protected from side channel leakage.
- What operations need to be protected from fault injection attacks.

Most of the data used by the secure code might not need protection from side channel leakage (e.g. address pointers), especially if the project is open source. In most cases, the crypto keys and confidential user data are the main targets. For example, in a data processing loop inside a crypto operation involving the access of crypto keys:

- The address of the key might not be a secret.
- The value of the key is a secret.
- Assuming that the number of iterations in the loop is fixed, then the loop counter values during the operation are not secret.
- The data being processed in the crypto operations may or may not be secret - This is application dependent.

The requirement for fault injection protection can be different from the requirement for information leakage protection. For example, although the loop-counter's value is not a secret, it needs to be protected to ensure that the crypto operation runs through a fixed number of iterations. As a result, you might need to declare an extra loop-counter variable in the source code to ensure the integrity of the program loop flow.

Side channel leakage

The general approach for addressing timing side channel risk(s)

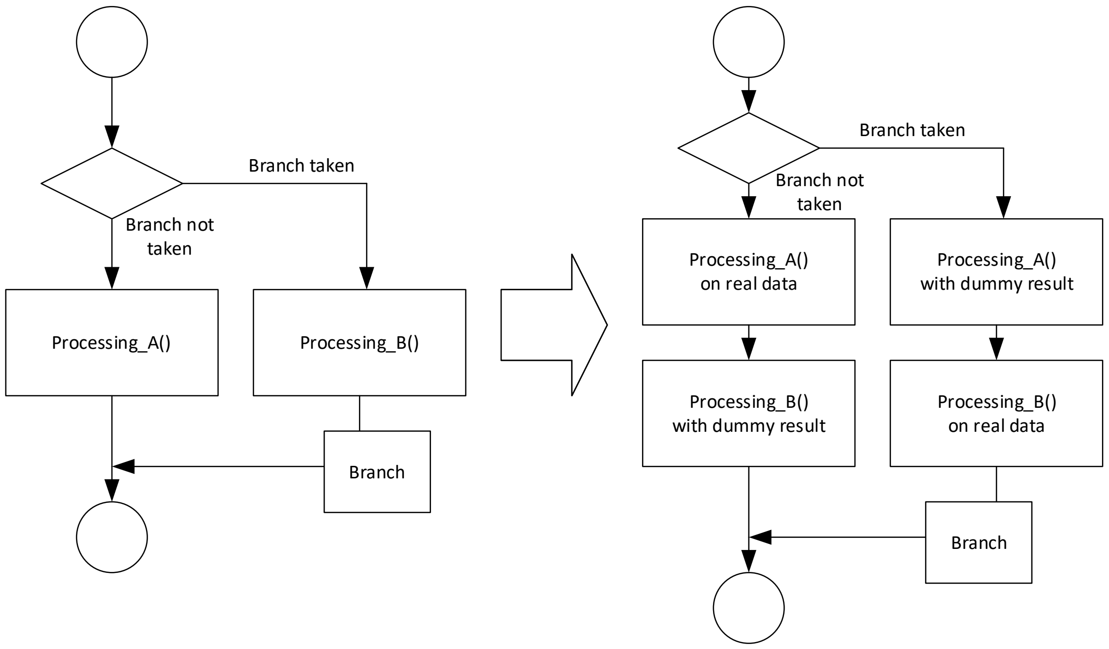
When creating secure firmware, if the device contains suitable crypto accelerators, it is usually better, instead of using software, to use the crypto hardware to handle the cryptography operations. This is because:

- Crypto accelerators can deliver a higher cryptography performance.
- Crypto accelerators can deliver better energy efficiency.
- Cryptography hardware usually has lower timing and power side channel leakage than running cryptography algorithms on the processor.
- Some crypto accelerators have built-in secure key storage which cannot be read by software. So even when the secure software has been compromised due to vulnerabilities, the keys are still protected.

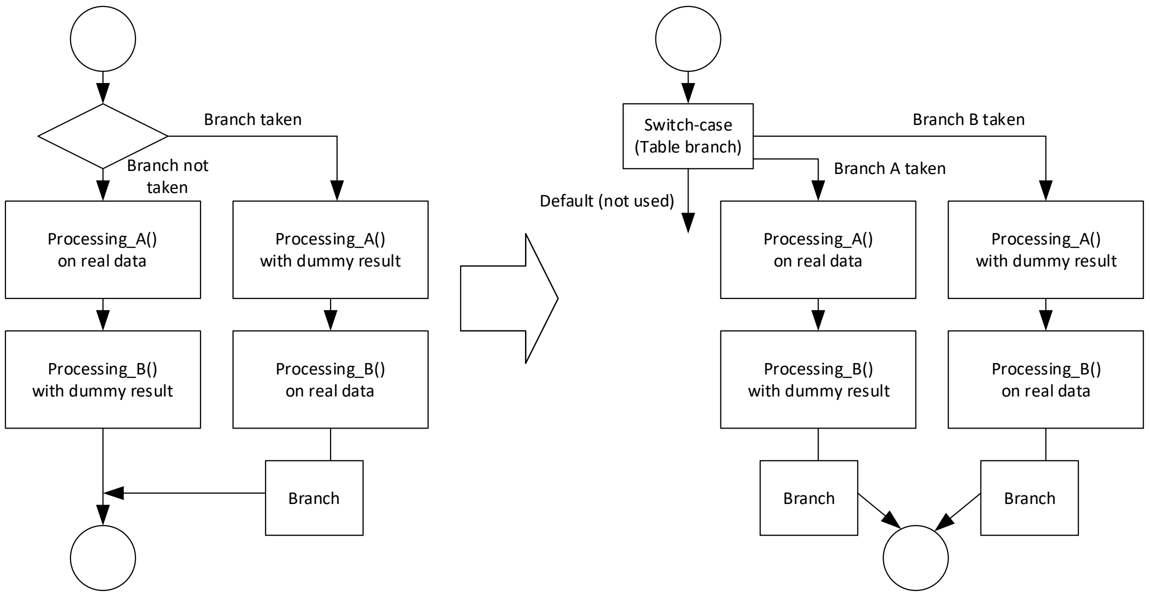
Some modern microcontrollers contain cryptography hardware that comes with side channel protection.

If it is necessary to use software to handle cryptography, there are some techniques that software developers can utilize to reduce the timing side channel leakage. The key areas that require special attentions are:

- Loop structures – From the side-channel leakage point of view, it is risky to have a program loop where the number of iterations is derived from a secret data value because the number of iterations could easily be observed from the timing/power signature(s). If a data processing function needs to be repeated by "X" number of times and "X" is a secret value with a maximum value of "K", one way to reduce the side channel leakage is to always execute the loop by "K" iterations. When using this arrangement, after the X'th iteration of the loop, dummy data processing is carried out instead of real processing for the remaining iterations. The dummy data processing does not affect the data processing results required by the application.
- Conditional branches - If there is a conditional branch where the condition is derived from secret information, the different execution paths (branch taken and not taken) can lead to observable timing/power signature(s) and this can result in leakage of the secret information. One way to reduce leakage is to carry out the same data processing operations in both paths (as shown in the diagram below) but execute the redundant operations with dummy results so that both paths have similar timing and power signatures.


Branch taken
Branch not taken
Processing_A()
Processing_B()
Branch
Branch taken
Branch not taken
Processing_A() on real data
Processing_A() with dummy result
Processing_B() with dummy result
Processing_B() on real data
Branch

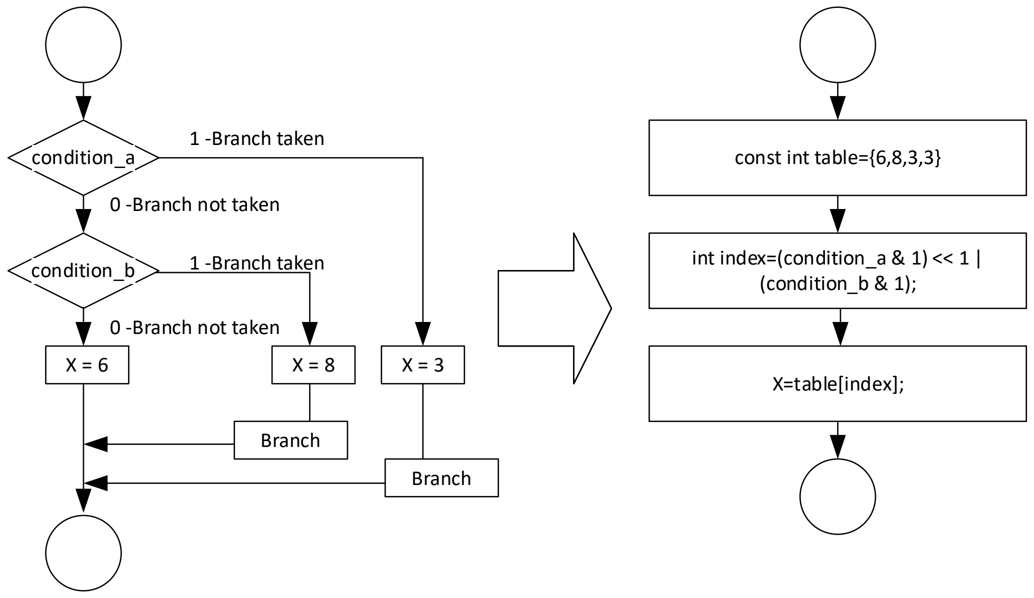
- Further enhancements for conditional branches – The conditional branch in the above example can still result in observable timing differences due to the fact the number of clock cycles needed for the conditional branch are different between the two execution paths. To overcome this issue, some of the conditional branches could therefore be converted into switch-case statements (see the diagram in the next page) so that a branch is needed in both cases, eliminating the cycle timing differences.



Branch taken
Branch not taken
Processing_A() on real data
Processing_A() with dummy result
Processing_B() with dummy result
Processing_B() on real data
Branch
Switch-case (Table branch)
Branch B taken
Branch A taken
Default (not used)
Processing_A() on real data
Processing_A() with dummy result
Processing_B() with dummy result
Processing_B() on real data
Branch
Branch

- Conditional branches for error handling - In most cases, conditional branches for dealing with error conditions (e.g. invalid input data format) do not need to be changed because such an event is unlikely to be treated as secret.
- Using instructions where the cycle-timings are independent of the data values - When processing secret data, avoid using instructions where the cycle timing is dependent on the data value. The number of clock cycles needed by some instructions like integer divide (as well as 64-bit multiply, and multiply-accumulate in the Cortex-M3) are dependent on the input data value. Newer Arm architectures introduced a Data Independent Timing (DIT) extension, and this is supported in new processors like the Cortex-M55 and Cortex-M85. In these processors, a range of data processing instructions are classified as DIT instructions, meaning that the number of clock cycles needed are independent of the data values. With the DIT extension, cryptography developers know which instructions are safe to use for processing secret data values, and therefore are able to avoid timing side channel leakage. The DIT extension does not imply that the cycle count is always constant and does not protect the system from power side channel leakage.
- Using a linear code sequence – reducing conditional branches that are dependent on secret data (excluding loop of fixed number of iterations) can help prevent timing side channel leakage. For example, some of the conditional operations in data processing sequences can be replaced with

look up tables. An example of this:



What I have shown here are just a few simple examples that reduce side channel leakage. There are other coding techniques that can also help prevent timing side channel leakage.

In addition, there are some commercial crypto libraries that are hardened against side channel attacks. Some literature about side channel hardening highlights the method of inserting random delays in specific codes to mask the timing signatures. This can be useful to make fault injection attacks harder. But from side channel leakage point of view, some researchers claim that if the attacker collects enough samples to analyze, the random delays can be averaged out, making the defense less effective than it would appear to be.

It is possible that some software countermeasures to reduce timing side channel leakages could end up increasing the power side channel leakage or could reduce the fault injection resistance. For example, when adding dummy data for processing, the choice of the dummy data value could affect the power signature. This is because the power consumption is affected by toggling of signals - If the usual value of a data path is 0xFF and the dummy data is 0x00, the toggling of signals could result in a noticeable power spike and result in power side channel leakage. To avoid this problem, real data can be used for the input of the dummy processing, with the processing results written back to dummy data locations. In addition, care must be taken to ensure that C/C++ compilers are not able to optimize away the dummy processing.

Power side channel leakage

In general, it is difficult for most software engineers to work out how to reduce power side channel leakage in their codes as the analysis requires dedicated tools. As a result, if you believe that your applications could be at risk of power side channel attacks, it is best to choose a microcontroller product that supports a crypto engine with side channel protection. If that is not possible, using microcontrollers with internal DC-DC converters might reduce the risk of power side channel attacks (providing that the generated core voltage is not measurable from the pins on the chip package).

When creating security applications, software developers should focus on the handling and accessing of secret information where those operations could lead to power side channel leakages. There are techniques that can help reduce the risk in handling secret data and some of these are as follows:

- When a processor accesses secret data (e.g. crypto keys), instead of accessing the data as sequential bytes, it is better to use wider memory accesses (e.g. word, doubleword) to help protect against templating of the power signature. To further reduce the risk, the memory access order should be randomized when accessing the secret data.
- When a secret data overwrites a memory location with known values, or when the secret data is overwritten by data known values, the memory state changes could lead to information leakage via a power side channel. As a result, instead of using known values for these operations, it is best to use freshly generated random data.

Potentially, additional protection could be implemented at the product level (e.g. PCB and product enclosure) to make it more difficult for a hacker to access the power pins needed to perform power side channel attacks.

Cache-based side channel

Using hardware accelerators to handle cryptography can also avoid cache-based side channel attacks in some systems. This is because the data handling is not routing through the processor’s internal cache(s). However, it might still be possible for a cache-based side channel attack to happen if the crypto engine and the processor shared a system level cache. Using non-cacheable memories in the system for crypto operations could reduce such risk (but look out for potential resource contention side channel issue(s), which is covered next).

If a cryptography algorithm is executed on a Cortex-M/Cortex-R processor, one possible arrangement to avoid cache-based side channel issue(s) is to use Tightly Coupled Memories (TCMs) which bypass the cache controllers. Because the code and data is not cached when running using I-TCMs and D-TCMs, the cache-based side channel issue can be avoided.

Resource contention side channel

Resource contention side channel issues were until recently rarely discussed. These issues are actually quite common in SoC and high-end microcontrollers where there could be multiple bus managers accessing memories and peripherals at the same time. When two bus managers access the same hardware resource at the same time, one of the transfers will be delayed and this could result in an observable timing side channel.

To avoid the side channel leakage of the secret information, secure firmware could set up the processor system so that when running Secure codes, the resources that the code uses are private (i.e. no contention). The setup could have a dedicated memory-bank that is private to the secure processor, or use other methods to prevent other bus managers (e.g. a DMA controller) from accessing the shared resource while the security sensitive operation is ongoing. However, such an arrangement could be problematic in certain applications. For example:

- The cost of implementing dedicated memory resources could be too expensive for low-cost microcontrollers.

- If bus transactions from other bus managers to the shared resource is blocked for too long, it can degrade real-time performance.

There can also be software workarounds to avoid side channel leakage. For example, using the following code as a starting point:

```
int    var_a, var_b; // variables in memory
int    cond_x;  // A secret with value of 0 or 1
  …
  if (cond_x) {
      var_a = 3;
  } else {
      var_b = 5;
  }
  …
```

The code snippet contains two execution paths, with updates to different variables in the memory in each path. Because "branch not taken" would usually be quicker than "branch taken", the execution cycle for the memory writes of both paths would also be different. If an attacker sets up a DMA transfer to the same SRAM block at the same time the processor writes to the SRAM in one of the execution paths, the attacker might be able to observe "cond_x", which is a secret.

To avoid the problem, we can modify the code to avoid the conditional branch and, instead, use data arrays to access the data variables and constants as detailed below.

```
int   var_array[2]; // var_a = array[0], var_b = array[1]
const int const_array[2] = {3,5};
int    cond_x;  // A secret with value of 0 or 1
  …
  var_array[i] = const_array[i];
  …
```

With the modified code, because there is only one execution path, there is no timing difference regardless of whether "cond_x" is 0 or 1.

In addition to bus arbitrations at the system level bus interconnect, similar bus arbitrations can take place inside the processor. For example, some processors provide a TCM access bus interface port to allow bus managers such as a DMA controller to access the TCM. If the DMA controller and the processor both access a TCM memory bank at the same time, the bus arbitration logic will delay one of the transfers and could result in an observable side channel.

There are other forms of resource contentions. For example, an application running in the normal world of a TrustZone enabled processor can send a request to the secure firmware to access a cryptography resource. If the access is denied, the deny response could imply that the secure software is handling some cryptography workload, indirectly showing that certain security event(s) are occurring.

Conceptually, a resource contention channel can exist without the need for the attacker to directly control a bus manager. For example:

1) In a TrustZone based system, if a secure service carries out access on behalf of a Non-secure software component, it is possible to create resource contention. Non-secure software could call a crypto API that programs a hardware crypto engine that generates an access pattern to a Non-secure memory region in an SRAM. At the same time, the same SRAM could also be used by Secure code for processing secret information.
2) In Armv8.1-M processors, scatter gather memory access instructions could access multiple addresses during the same clock cycle. If two of the accesses are targeting the same TCM interface, a contention could occur. As a result, if the addresses used are based on a secret, there could be side channel leakage issues. It follows therefore that secure firmware should not use scatter gather memory access instructions where the address is derived from secret information.

Countermeasures for simple fault injections

Because processor systems are digital circuits that require specific conditions to operate reliably, e.g. the voltage, clock speed, temperature and the electromagnetic radiation of the surroundings all need to be within an acceptable range, it is in theory possible to introduce faults into the processor systems.

For some application scenarios, it can be assumed that the fault injection attacks will last for only a very short time and will only affect the processor's operation at an individual instruction level. For example, some of the common voltage and clock glitching attacks target a specific program execution cycle to achieve one of the following effects:

- Skipping an instruction (e.g. skipping a compare instruction so that a conditional branch branches into an incorrect execution path).
- Reading an incorrect value from the memory or a peripheral.
- Writing an incorrect value to the memory or a peripheral or skipping the write operation.

However, there are also fault injection attacks that could affect the chip on a wider scale. In such cases, it is not a given that the processor system will resume the correct operation after the fault injection attack. Therefore, when one is designing protection feature(s) against fault injection attacks, one should not assume that the processor system can continue to reliably execute the software. The key focus, therefore, when one is designing protection features should be:

- To detect the fault injection attacks,
- To prevent secure information from being leaked after an attack is detected.

The protection features could take the following actions:

- Abandon the operation(s).

- Because the system might not be able to resume reliable operation(s), it might reset automatically after abandoning the operation(s).
- In some instances, the device could erase all secret data from the chip (This is a non-recoverable operation).

Inside the chip's hardware, a processor might support redundant hardware resources to help detect errors introduced by fault injection. For example:

- Dual-Core Lock-Step (DCLS): In a processor system with DCLS implemented, the functional logic is duplicated and comparators are used to detect differences in the processor's output, flagging up an error if there is a mismatch. Usually, the redundant logic operates with a timing offset of 2 or 3 clock cycles to reduce the probability of common mode failure. The RAMs inside the processor might not be duplicated if they are already protected by other mechanisms such as Error Correction Code (ECC).
- Transient protection: In a processor system with transient protection, the register bank(s) and critical functional logic include parity bits so that if a fault injection attack triggers a bit flip in a register, or if incorrect data is written to a register, the parity logic can detect the error.

If an error is detected, either by DCLS or transient protection logic, it could mean that the processor is no longer in a reliable state to execute software and must be reset. As a result, such systems should also have other hardware to handle safe shutdown of the system. In addition, these systems can have a higher power consumption and silicon cost due to the introduction of the additional hardware.

For commercial microcontrollers, cost pressures are usually high and therefore it is unlikely that commercial microcontrollers will feature DCLS or transient protection features. However, some chips that are targeting markets with security needs do support monitoring hardware to detect voltage and/or clock glitching attacks.

In addition, software developers can also introduce security measures in their applications. For example:

1) Introducing random timing delays in program flows to make it much harder to apply voltage or clock glitching attacks to a specific location of program code. To do this, the chip needs to have a random source (e.g. a True Random Number Generator) so that the execution timing of the security sensitive code cannot be predicted. In a software environment with TrustZone enabled, a random delay could be placed after each transition from the Non-secure state to the Secure state so that even if the Non-secure software provided a timing hint to an external observer that it was going to switch to the Secure code, the exact timing of the Secure operation(s) could not be determined.
2) Introducing duplicated instructions/operations in the program flow so that even if an attacker injected a fault "skip" one instruction, the software could still operate correctly; and potentially report that there was a potential attack. Some examples of these can be found in the FIH library in the Trusted Firmware-M repository as mentioned earlier.
3) Duplicating critical variables: For critical variables used to determine program executions or to select a policy (e.g. for security management), it is recommended that the variable is either duplicated or a shadowed version is created. Before the variable is used, for example, to make a decision, a comparison of both values should be carried out. This is helpful because it is

generally hard to attack both values in the same way. It is even better to use inverted logic for the shadow value so that the same fault won't cause the same effect on both values.

4) In conditional branches that are critical to the system being developed, add addition checking of the branching condition. If one of the branch directions is beneficial to an attacker, then double check the conditions for this direction before proceeding. For example, if the variable providing the branch condition is a Boolean value (i.e. either TRUE or FALSE), and if it is in the attacker's interest to make the software execution take the FALSE branch by attacking the comparison operation, then an additional comparison of the variable to FALSE should be carried out.
5) For critical values used for comparison and for the choosing of the execution path, use complex values with:
    a. A large Hamming distance between them and
    b. A variety of interleaved set and unset bits.

    When an attacker injects a fault, usually, a group of adjacent bits turns to 0 or 1. Therefore choosing values such as 0xaaaaaaaa and 0x55555555 to represent the TRUE and FALSE Boolean values would be the best arrangement.
6) Introducing program flow monitoring primitive functions to make sure that critical stages of the program execution have not been skipped. For example:
    a. re-verifying the number of executed loop iterations.
    b. counting the program execution expected checkpoints.
7) In functions that return a success/fail status value, setting the function's default return status variable to fail status and only setting it to success status after all the required processing is completed and verified. If the relevant function execution is attacked and the function is "glitched" to carry out a return earlier than expected (i.e., before all the computations are finalized), it will return a failure status by default. This arrangement is considered a general programming technique.
8) Ensuring that all redundancies of data/operations are checked by comparison operations. When there is an attack, the comparison should result in a comparison failure. Such a failure cannot happen in any other execution flow and is handled as a fatal error. The software developer should also define a failure handling policy; if an attack is detected, the device typically should be reset or shut down.

Introducing duplicated operations in software is a complex subject. For example, if we want to add duplicated operations and redundancy checking to the following simple code:

```
void test1(int * src, int * dest, int count)
{
  int i;
  for (i=0;i<count;i++) {
    dest[i] = src[i];
  }
  return;
}
```

First, we need to define what information/operations we need to protect. There are several types of attack targets that we would want to protect:

- Setting initial value of loop counter “i” to 0.
- Increment of loop counter.
- Comparison of “i” with “count”.
- Conditional-branch taken/not taken.
- Reading of source data for copying.
- Writing of destination data.

To keep things simple, the following illustration assumes that the input data is trusted (i.e. no need for range check).

Another assumption is that there is only one point of fault inject attack during the execution of this function. The code can thereby be rewritten as follows to address most of the fault injection risks:

```
void test2(int * src, int * dest, int count)
{
  volatile int i1=0, i2=0;  /* i1 and i2 are initialized to 0 */
  if (i1!=0) {error_detected();} /* If loop counter is not zero, a fault is detected */
  if (i2!=0) {error_detected();} /* If loop counter is not zero, a fault is detected */
  /* i1 and i2 are safe to use at this point */
  do{
    do{
      if (((volatile int) i1) >= count) {
        error_detected();
        break;
      }
      dest[i1] = ((volatile int) src[i1]); /* Copy data */
      if (((volatile int) dest[i1]) != ((volatile int) src[i1])) { /* copying failed */
        error_detected(); } /* Copy operation failed */
      i1=i1+1; /* Increment loop counter i1 */
      if (i1 != (i2+1)) { error_detected(); } /* loop counter increment failed */
      i2=i2+1; /* Increment loop counter i2 */
      if (i2 != i1) { error_detected(); }        /* loop counter increment failed */
    } while (i2<count);
    /* If not in the last iteration
       - if the branch backward fails, the outer while-loop will catch this.
       If in the last iteration
       - if the branch is incorrectly taken, it will reach the compare condition branch and
         the error will be detected. */
  } while (i1<count);
  /* If not in the last iteration, this is not normally reached because of the inner loop.
     If in the last iteration, and if the branch is incorrectly taken, it will reach the compare branch
     and the error will be detected */
  return;
}
```

Some concepts of the above code are explained below:

- The loop counter “i” is changed to two loop counters “i1” and “i2”. Both are incremented in each iteration.
- Several “volatile” casting operations are used to ensure the latest value of the variable is loaded from the memory instead of reusing a copy inside the processor from a previous load.
- An error_detected(void) function needs to be defined to handle the error.

There are some limitations to consider:

- Most of the redundant coding techniques in the example are focused on protecting against single fault injection attacks and are not able to handle multiple fault injections. However, if the codes are carefully constructed, it is possible to address some forms of multiple attacks. Typically, such software would apply many of the aforementioned techniques around various parts of the code, e.g. using inverted logic for shadow values, multiple random delays during the software flow. Please note: While it is harder to apply multiple fault injections in a physical attack than a single fault injection, the availability of modern physical attack tools have made this practical and multi-glitch attacks on Arm processor system have already been demonstrated by a security researcher [7].
- During a memory write operation, if the processor writes to an incorrect memory address due to a fault injection attack, the protection techniques already illustrated in the example would not be able to detect or mitigate the fault. To mitigate this, it is better to write critical values alongside integrity protection values to allow the software to check the integrity of the data when the data is read back.
- A C/C++ compiler could transform the code and inject additional branch instructions. If that happens, the branch inserted by the C/C++ compiler might not be protected from fault injection attacks.
- Because C/C++ compilers could optimize away redundant code, it is necessary to examine the generated code.

If an application needs a high-level of protection against fault injection attacks, some of the critical functions might need to be coded in assembly to prevent the C/C++ Compiler from reducing the protection coverage. Please also note that depending on the application code, fault injections to some of the instructions might not lead to security vulnerabilities.

Key take-aways

Physical attacks can be a threat to a product, requiring specific expertise and actions for them to be mitigated. However, many products do not require physical protection features. This is dependent on many factors such as the usage and the value of the assets inside the product. System designers need to decide what is the right level of security features required for the products that they are designing.

There are some public domain materials that provide guidelines and coding examples on how to prevent/reduce timing side channel leakages and fault injection attacks. (e.g. Secure Application Programming in the presence of Side Channel Attacks, [8]). When creating program codes for systems that are at risk of physical attacks, it is desirable to use tools to identify where weaknesses are inside the codes and to decide where additional protections are needed. There are commercial tools available from various vendors (e.g. Riscure, eShard, …), and those vendors often offer their security expertise as a consultancy service for hardening codebases. There is also an open-source tool framework, called "PAF, the Physical Attack Framework" available on Arm Github [9]. Note: This tool requires access to Arm FastModels.

Protection schemes against different physical attacks (e.g. timing and power side channel, fault injection, etc) can be incompatible at times, so a trade-off will have to be found. However, most protection schemes are fragile and are often removed by optimizations in the compilers. It is thus essential to check that the resulting binaries have the desired level of protection. Ideally, software developers should be in a position to routinely undertake this during the development of their projects, e.g. handling the checking as part of the project's continuous integration testing. This is needed because the codebase and the compiler (and its versions) are likely to evolve over the lifetime of the product.

Summary

This document provides a quick introduction to side channel and fault injection attacks and their countermeasures. Please note that there are additional forms of side channel and fault injection attacks not mentioned in this paper.

This document also explains that different security technologies are required to address different security needs. While TrustZone technology is great for addressing security needs for connected devices where attacks are likely to originate remotely, systems that are at risk of physical attacks will require additional physical protection features.

Reference and Further readings

Reference	Document
1	Low level software security book. https://github.com/llsoftsec/llsoftsecbook
2	Secure software guidelines for Armv8-M https://developer.arm.com/documentation/100720/0300/
3	Stack sealing and why it is needed in TrustZone for Armv8-M https://developer.arm.com/documentation/102446/0100?lang=en&rev=03
4	KAN335 – Stack Sealing on Armv8-M https://developer.arm.com/documentation/kan335/latest/
5	Row Hammer https://en.wikipedia.org/wiki/Row_hammer
6	VoltJockey: Abusing the Processor Voltage to Break Arm TrustZone https://dl.acm.org/doi/10.1145/3427384.3427394
7	Arm security knowledge base article: Clarification of Multi-Fault-Injection Attacks on TrustZone enabled Cortex-M based systems. https://developer.arm.com/documentation/ka005159/latest
8	Secure Application Programming in the presence of Side Channel Attacks.

	https://riscureprodstorage.blob.core.windows.net/production/2017/08/Riscure_Whitepaper_Side_Channel_Patterns.pdf
9	Physical Attack Framework. https://github.com/ARM-software/PAF

Additional information

Several Arm processors are designed with physical protection features. These are:

Processor	Architecture	Product page
Cortex-M35P	Armv8-M	https://developer.arm.com/Processors/Cortex-M35P
SecurCore SC300	Armv7-M	https://developer.arm.com/en/dev2/Processors/SecurCore%20SC300
SecurCore SC000	Armv6-M	https://developer.arm.com/en/dev2/Processors/SecurCore%20SC000

These processors are used in specialized products such as smart cards. If you need further information on these processors including technical documentation, please contact Arm directly.

Acknowledgements

I would like to thank the following people for providing feedback and technical information during the preparation of this paper: Einat Luko, Arnaud De Grandmaison, Kristof Beyls, Peter Williams, Thomas Grocutt, Akim Layachi Daineche, Shebu Varghese Kuriakose.