Toward Better Fault Characterization Tests

Vincent Werner, Laurent Maingault, Marie-Laure Potet

Toward Better Fault Characterization Tests

***Abstract*—Although fault injection is a powerful technique to exploit implementation weaknesses, this is not without limitations. An important preliminary step, based on rigorous calibration of the fault injection equipment, greatly affects the accuracy and repeatability of the results. Even if equipment calibration can be performed directly with the target application, it can be more relevant to use fault characterization tests instead, which are small programs designed to propagate fault effects, in order to identify more easily faulty behaviors. However, designing tests that propagate most fault effects is not an easy task. In this study, we propose original metrics to evaluate fault characterization tests based on fault propagation and fault discrimination. These metrics are used to compare the performance of different fault characterization tests from the literature, so as to identify the most important design choices according to a wide range of fault models. From these observations, we detail a general approach to design more efficient tests. Finally, we propose a minimal test set based on these guidelines, to quickly find settings with high fault probability on a Cortex-M4 32-bits microcontroller, using voltage fault injection.**

***Index Terms*—Fault Injection, Characterization, Calibration**

I. Introduction

Fault injection is a powerful technique to bypass security features of embedded systems, such as code protection mechanisms [1]. Using electrical glitches [2], focused light [3], electromagnetic pulses [4] or even nanofocused X-rays [5], one can locally perturb the chip environment to alter its behavior and gain access to critical information. Although fault injection can lead to impressive results, this is not without limitation. One of the biggest challenges is the calibration of fault injection equipment. Each fault injection equipment has multiple specific parameters that must be adjusted precisely, such as the positions x, y, z of an electromagnetic probe tip.

This preliminary step is required in order to find accurate and repeatable faults. In academic literature, several optimization methods have been proposed to finely tune parameters of a given fault injection equipment [6]. In practice, these methods can be used to calibrate the equipment either directly on a critical section of the target application (e.g. the last rounds of an AES), or indirectly on an external program designed to propagate faults. This will be referred as direct and indirect calibration, respectively.

Direct Calibration consists in focusing on a small code chunk rather than the whole application; in order to calibrate the fault injection equipment for a specific attack scenario. Indeed, direct calibration on the entire application is often proved to be infeasible in reasonable time. The selected section is adjudged as critical and can potentially lead to security holes with fault injections. If necessary, the section can be isolated from the rest of the application and integrated into a test environment with built-in target synchronization mechanisms to help fault injection. Direct calibration implies 1) the evaluator has identified the critical section which is challenging, 2) the evaluator has an attack scenario and 3) the evaluator will not cover the whole application. The main advantage is that, once a successful fault is injected, the vulnerability is directly confirmed. However, keep in mind that only the assumed critical section is covered, so potential vulnerabilities can be easily missed. Moreover, depending on the application, lack of feedback makes direct calibration much more difficult [7]. Furthermore, because direct calibration focuses on attack success, and not on fault effect understanding, the results are not necessarily generalizable to different applications (or other critical sections) of the same microcontroller.

On the other hand, *Indirect Calibration* attempts to comprehend fault effects on the target microcontroller, and then to generalize the results to the entire target application. To this end, an external program, called *fault characterization test* in this study, is used to maximize the number of effective faults on the target microcontroller. A fault characterization test is not the application itself, or a section of the application, but rather a series of instructions, arranged in such a way as to highlight the assumed, or adjudged effects of fault injections, on the target. In other words, a fault characterization test is designed around one or more fault model assumptions. The main advantage of indirect calibration by comparison with direct calibration is that, once these assumptions have been validated experimentally, the fault models and the fault injection settings found can be used to evaluate different security mechanisms of the same microcontroller, thereby reducing the initial time investment. Moreover, a characterization test is often smaller than the target application, reducing even more the time required in the long run. Furthermore, a fault characterization test simplifies the equipment calibration by giving instant feedback on the effectiveness of the fault injection parameters, in comparison with direct calibration. Finally, indirect calibration can help in reducing complexity of multiple fault injections [8], as fault injection parameters can be observed independently of each other, in comparison with direct calibration.

Table I sums up the main points of both approaches. Depending on the objective and the means employed, one can choose to go either with direct or indirect calibration. Nevertheless, one question remains, what is a (good) fault characterization test ? Because unlike direct calibration, whose test vector is clearly defined (the critical section), indirect calibration strongly relies on the design of fault characterization tests. We will see in section III that depending on the design, outcomes can be very different, which can be a problem for indirect calibration. The main issue arises when

Calibration	Test Vector	Goal	Advantage	Drawback
Direct	Critical section	Calibrate equipment Validate attack scenarios	Effective fault $\Rightarrow$ vulnerability	Not generalizable Lake of feedback
Indirect	Fault characterization test	Calibrate equipement Understand Fault Effects	Generalizable Instant feedback	Effective fault $\not\Rightarrow$ vulnerability

TABLE I: Comparison between direct and indirect calibration.

actual fault effects on the microcontroller do not fit the fault models assumptions taken during the characterization test design. Therefore, it can be impossible to understand fault manifestation and propagation. For example, the fault effect can be entirely masked during execution, due to incorrect assumptions (see in section III). More generally, we will show that the characterization test design can be flawed if the propagation and discrimination properties (detailed in section II) are not verified w.r.t the fault model assumptions taken. There is no general framework for characterization test design yet, and many different designs have been proposed in the literature. Most of the time, the characterization tests are empirically designed, which takes time, and can be suboptimal due to design flaws.

A better way to design characterization tests would be to systematically validate their fault propagation and discrimination, using fault injection simulation. In this regard, we propose general guidelines to help fault characterization test design, for a wide range of fault models, based on the first comprehensive study of popular fault characterization tests from the literature. Then, with these general guidelines and a simulation-based fault injection tool, `CELTIC`, we find optimal fault characterization tests according to specific fault models, and combine them to generate a minimal test set that covers instruction, register and memory corruption fault models. Accordingly, our contributions are as follows:

- Proposition of metrics to evaluate performance of fault characterization tests, according to fault model assumptions.
- Evaluation of popular fault characterization test designs according to these metrics.
- Proposition of general guidelines to help fault characterization test design according to assumed fault models.
- Proposition of optimized fault characterization tests according to our metrics.

In this article, we will first detail the propagation and discrimination properties and define performance metrics in section II. Then, in section III, we will evaluate the different characterization tests from the literature according to our performance metrics. This first study will give us the main direction to design a fault characterization test efficiently. Next, in section IV, we will generate optimal fault characterization tests w.r.t fault models, using results of section III. In section V, we will present two practical applications of these tests, using voltage fault injection.

II. Properties and Metrics

In this section, we will define two relevant properties of fault characterization tests, *propagation* and *discrimination*. Then, performance metrics will be derived from these properties.

Fault characterization tests are usually divided into three parts:

- Initialization of a set of variables V that will be manipulated during the characterization test execution (general purpose registers, memory locations).
- A sequence of instructions I where the faults are injected. The operands of these instructions are chosen from V. At each instruction execution, the *current program state* is updated (w.r.t the V values). The current program state is not observable until the end of I.
- A subroutine that sends back the final values of V at the end of I, referred to as the *final state*.

The main goal of fault characterization tests is to gather as much information as possible about the microcontroller behavior in response to faults. Therefore, they must *propagate* the fault effects on the microcontroller to the end of I. The fault effect must not be masked during the execution of I, otherwise the results will not be representative of the actual microcontroller behavior.

Moreover, in order to understand more easily the fault effects or to validate fault model assumptions, each final state must be characteristic of a particular fault effect. Accordingly, the fault characterization test must *discriminate* the different effects induced by fault injections.

In the following, we formalize the propagation and discrimination properties of fault characterization tests. Note that the propagation and discrimination properties depend on the assumed fault models. Thus, fault characterization tests can propagate and discriminate some fault effects, but not others.

Definition 1 (Propagation): Let $s_\star$ be the final state of the reference execution trace, that is, an execution trace of I without fault. Let S_m be the set of final states of all the possible execution traces of I, given the fault model m. I ensures a propagation property w.r.t a set M of fault models if:

$$\forall m \in M, s_\star \notin S_m$$

Definition 2 (Discrimination): Let S_m be the set of final states of all the possible execution traces of I, given the fault model m. I ensures a discrimination property w.r.t a set M of fault models if:

$$\forall (m,m') \in M^2, S_m \cap S_{m'} = \emptyset$$

In other words, a characterization program, w.r.t a set M of fault models, must maximize the number of faulty final states (i.e. final states different from the expected final state) and faulty final states must be distinguishable between fault models. We can derive from these properties performance metrics to evaluate a characterization test, namely the propagation, discrimination and coverage rates, which are defined as follows:

Definition 3 (Propagation Rate): N_f denotes the number of faulty final states w.r.t to the set of fault models M. N_e is the number of expected final states. N_c refers to the number of crashes. The propagation rate of the characterization test is defined as:

$$PR = \frac{N_f}{N_f + N_e + N_c}$$

Definition 4 (Discrimination Rate): $N_{\neq}$ denotes the number of distinct final states and $N_{\cap}$ the number of identical final states between the set of fault models M. The discrimination rate of the characterization test is defined as:

$$DR = \frac{N_{\neq}}{N_{\neq} + N_{\cap}}$$

Definition 5 (Coverage Rate): N_c denotes the number of fault models covered and N_t the total number of fault models considered. The coverage rate of the characterization test is defined as:

$$CR = \frac{N_c}{N_t}$$

As we need to maximize N_f, $N_{\neq}$, and N_c, the higher *PR*, *DR*, and *CR* are, the better. We can quickly compare the performance of two tests with the product of *PR*, *DR*, and *CR*. *PR* and *CR* should be prioritized for equipment calibration purposes, whereas *DR* is important to validate fault model assumptions.

We use a simulation-based fault injection tool (`CELTIC`) to evaluate the *PR*, *DR*, and *CR* of a given characterization test, w.r.t the set M. `CELTIC` [9] is a dynamic binary code analyzer which simulates fault injection using fault models, that describe fault effects at the instruction set architecture (ISA) level. The tool itself is built around an efficient emulator for various architectures and designed to meet the needs of the evaluators, such as the ability to easily add new proprietary instruction sets or new fault models. Fault models in CELTIC can be configured and adapted to simulate complex fault effects throughout the instruction cycle (fetch, decode, execute) but also on registers and memory. Furthermore, CELTIC can save traces of faulty execution, useful for analyzing simulation results in an automated way [8], but also evaluate the propagation and discrimination rate.

In the following, we will explain how the different characterization test design choices influence *PR*, *DR*, and *CR*, based on an in-depth study of popular fault characterization tests from the academic literature.

Category	Fault Model Family	Description
Instruction corruption	F1	Single and multiple instruction skips
	F2	Bit-sets on fetched opcode
	F3	Bit-resets on fetched opcode
Register corruption	F4	Bit-sets on source register value
	F5	Bit-resets on source register value
Memory corruption	F6	Bit-sets on memory cell value
	F7	Bit-resets on memory cell value

TABLE II: Fault models used.

III. Design Guidelines for Characterization Tests

In this section, we will evaluate, using fault injection simulation, the fault *propagation* and *discrimination* of different characterization tests with the performance metrics *PR*, *DR*, and *CR*. Then, we will compare the influence of the different design choices on performance and derive general guidelines to design fault characterization tests according to assumed fault models.

A. Fault Models

We have considered a set of classical ISA fault models. This set, presented in Table II, covers a wide range of fault effects that can occur during indirect calibration, from instruction corruption (F1-F3), through register corruption (F4, F5) to memory corruption (F6, F7). These fault models have been implemented in `CELTIC` and will be used to compare the *PR*, *DR*, and *CR* of the different fault characterization tests.

B. Fault Characterization Tests Overview

We have selected 8 different and representative fault characterization tests from the literature, presented in Table III. The sequence of instructions I and initial values V, for each test, can be found in Appendix (Table X, Table XI, Table XII, and Table XIII). These tests have been designed to understand various fault effects on different microcontrollers (e.g. [10], [11]) and are commonly used for indirect calibration. Note that we have translated them, if necessary, into the ARMv7-M 16-bit instruction set to compare performance more easily. We have classified these tests according to different criteria in Table III:

- Idempotent versus non-idempotent instructions (I idempotence).
- Number of instructions in I (I size).
- Variety of Instructions in I (I variety).
- Type of Instructions I (I type).
- Initial values of the set of variables V (Initial V values).

In the following, we will highlight how these design choices can influence on the *PR*, *DR*, and *CR* of characterization tests according to fault models considered in Table II, in order to conceive better ones.

Test	I idempotence	I size	I variety	I type	Initial V values	Related Work
T1	No	Large	Medium	Standard data-processing instructions	Distinct	[12]
T2	Yes	Large	Medium	Standard data-processing instructions	Distinct	[11], [13], [14]
T3	No	Small	Medium	Standard data-processing instructions	Distinct	[15]–[17]
T4	No	Large	Medium	Standard data-processing instructions	Identical	-
T5	No	Large	Low	Standard data-processing instructions	Distinct	[12], [18], [19]
T6	No	Small	Medium	Load and Store instructions	Distinct	[10], [18], [19]
T7	Mixed	Large	High	Standard data-processing instructions	Distinct	[10]
T8	Yes	Large	Medium	Load and Store instructions	Distinct	[11]

TABLE III: Characterization tests used.

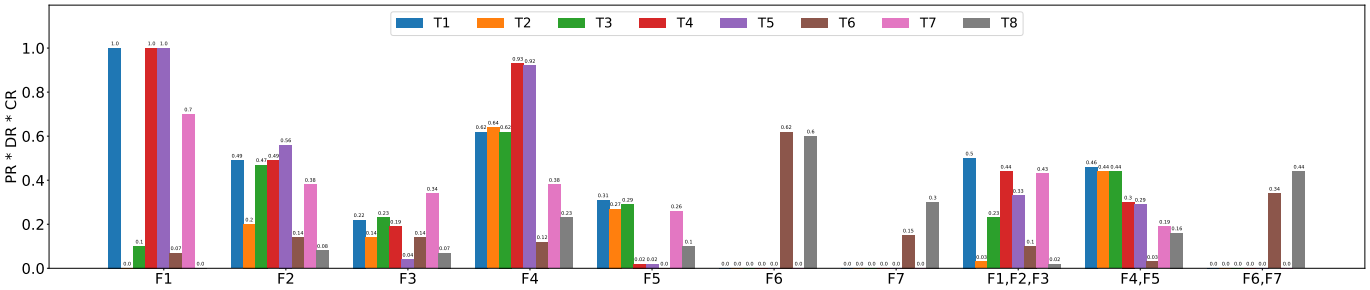

Fig. 1: Comparison of the characterization tests in Table III according to $PR * DR * CR$, w.r.t to fault models in Table II.

C. Performance Comparison

The selected tests have been evaluated using a simulation-based fault injection tool `CELTIC`. The performance metrics *PR*, *DR*, and *CR*, according to fault models and for each characterization test, are detailed in Appendix (Figure 5, Figure 6, Figure 7), and the product $PR * DR * CR$ can be found in Figure 1. We also evaluated the performance between fault models of the same category, and these results will be compared with our tests in section IV. Before going into details, a few general observations can be drawn from the results:

- Instruction skip fault models (F1) have the highest standard deviation, which suggests *PR*, *DR* and *CR* are highly dependent on design choices for these models.
- Bit-set and bit-reset fault models are complementary to one another, thus the sum of *PR* for bit-set and bit-reset fault models of the same category cannot be above 1. For example, test T5 propagates most opcode bit-sets (F2) but misses opcode bit-resets (F3) in Figure 5.
- Except for memory corruption fault models, the T1 test works well on the rest of the benchmark, especially between fault models of the same category.

By cross-checking simulation results (Figure 1, fault models (Table II), and the main characteristics of the tests (Table III), we can identify important design choices, according to assumed fault models, that have an effect on *PR*, *DR* and *CR*.

1) Idempotence: Idempotence is the property of an instruction to be executed several times without changing the result. For example, an instruction *mov* between the same register is idempotent whereas an addition with an immediate value is not. Idempotence strongly affects *PR*, *DR* and *CR* as detailed in Figure 5, Figure 6 and Figure 7. Most characterization tests are non-idempotent except T2 and T8. Both T2 and T8 tests get 0 for *PR*, *DR* and *CR* w.r.t instruction skips (F1). The only situation where an idempotent characterization test can be used is when faults only affect data (F4-F7) and never disrupt the program counter nor the fetch/decode pipeline stage (F1-F3). *Therefore, prefer using non-idempotent instructions rather than idempotent ones for F1, F2 and F3 families.*

2) Number of Instructions: The number of instructions influences *PR* and *CR*, as presented in Figure 5 and Figure 7. The T3 and T6 characterization tests contains 8 and 7 instructions (small) respectively whereas others contain around 80 instructions (large). With regard to the results, using a large characterization test is better if instruction skips are suspected (F1). In addition, a larger sequence I is more convenient to use during practical experiments, as a smaller sequence I tends to be more difficult to target. Hence, there is no particular indication to use a small characterization test. *For F1 family, we strongly recommend to consider larger code size.*

3) Variety of Instructions: Although most characterization tests evaluated are based on 7 to 8 different instructions (medium variety), the T5 characterization test only considers 1 instruction (low variety) whereas the T7 used 80 different instructions (high variety). The T7 characterization test outperforms the rest of the benchmark w.r.t *CR* especially for instruction corruption fault models, but it underperforms in terms of fault discrimination (*DR*). We can deduce that increasing variety of instructions improves fault coverage at the expense of fault discrimination. On contrary, reducing variety of instructions increases the performance for specific fault models, as observed with the T5 test for F2 fault models. *In section IV, we will extend this observation to find optimal tests for each fault model.*

4) Instructions type: Only characterization tests using load and store instructions (T6 and T8) can propagate memory corruption fault models (F6, F7). Note that the remaining characterization tests, that use standard data-processing instructions (instructions which manipulate data within registers), have better performance especially for opcode corruption models (F2, F3). *Therefore, except for memory fault models, we recommend standard data-processing instructions.*

5) Initial Values: Depending on the fault model considered, initial values can affect *PR* and *CR*. The T4 characterization test initializes all variables (registers) to zero while others initialize each register with a distinct value. The initial *V* values do not affect *PR* and *CR* for instruction corruption (F1-F3). On contrary, we can observe that the initial values of the variables *V* influences the results, especially for register corruption fault models (F4, F5), as shown in Figure 5 and Figure 7. Although not shown, the initial values loaded and stored in the manipulated array of the characterization test T6 and T8, also influence *PR* and *CR* for memory corruption fault models (F6, F7). *Accordingly, we recommend to choose carefully the initial V values for F4-F7 fault model families.*

D. Design Guidelines

Category	Family	Most Important Design Choices
Instruction corruption	F1	Standard data-processing instructions Non-idempotence Large number of instructions
	F2, F3	Standard data-processing instructions Non-idempotence
Register corruption	F4, F5	Standard data-processing instructions Idempotence Initial *V* values
Memory corruption	F6, F7	Load and Store instructions Idempotence Initial *V* values

TABLE IV: Most important design choices w.r.t fault models in Table II.

To sum up, we have highlighted different design choices that must be studied during the characterization test conception. Table IV summarizes the most important choices that strongly influence *PR*, *DR*, and *CR* according to the simulation results with `CELTIC`. For example, we recommend to use a large number of non-idempotent instructions for instruction skip fault models. In the following, we will use these guidelines to find better fault characterization test for each fault model family, according to our metrics.

IV. Find Better Fault Characterization Tests

In this section, we propose optimal fault characterization tests that maximize *PR*, *DR*, and *CR* for each fault model family. Then, to reduce the time spent on indirect calibration, in order to save time for the actual fault attack, we combine these optimal tests into a minimal set of three tests, while preserving *PR*, *DR*, and *CR* as much as possible.

A. Find Optimal Test for each Fault Model

To maximize *PR*, *DR*, and *CR* we must find the best instruction and the initial values for each fault model family. We found in section III, that the T5 test, based on the same instruction repeated multiple times, can achieve better performance (Figure 1) for opcode corruption (F2) than other tests based on different instructions. We will extend this observation, and find the instruction *i* repeated *n* times and the initial values *V* that maximize *PR*, *DR* and *CR*, for each fault model in Table II.

We propose an approach to solve this problem which is generalizable to different instruction sets easily. For most fault model families, the best instruction *i* is trivial and can be derived directly from guidelines Table IV. For opcode corruption fault models (F2 and F3 families), we will use `CELTIC` to find automatically the optimal instruction *i* which maximizes *PR*, *DR* and *CR* according to opcode corruption fault models.

a) ***Instruction Skips (F1)****:*

Proposed test: an addition with immediate value repeated *n* times, with *n* greater than the largest instruction skip assumed. Initial values *V* can be chosen randomly.

Justification: The instruction *i* must update the current state into a new distinct one, at each execution of *i*. Therefore, if any instruction is skipped, the final state will be different ($PR = 1, CR = 1$), and also, the final state will depend on the number of instructions skipped ($DR = 1$). Note that, the number of distinct states must be greater than the largest instruction skip assumed. For example, an addition or subtraction with immediate value are good candidates for that purpose; at each instruction execution, the destination register will be incremented or decremented (*n* distinct states). On contrary, an *exclusive or* with immediate value is not ideal; at each instruction execution, the destination register will be toggled (only two distinct states).

b) ***Register Corruption (F4, F5)****:*

Proposed test: an instruction *mov* between the same register. The number of instruction *n* does not matter. Initial values *V* must be set to `0x0...0` (`0xf...f`) for bit-sets (bit-resets) fault models.

Justification: The instruction *i* must be idempotent and not update the current state. Therefore, if any corruption of the value of the source or the destination register occurred, then the corrupted value will be propagated ($PR = 1, CR = 1$), and also, the final state will depend on the corrupted bit ($DR = 1$). For example, a *mov* instruction between the same register is a good candidate.

c) ***Memory Corruption (F6, F7)****:*

Proposed test: an instruction *store* followed by an instruction *load* at the same memory address with the same register. The number of instruction *n* does not matter. Initial values *V* must be set to `0x0...0` (`0xf...f`) for bit-sets (bit-resets) fault models.

Justification: The instruction *i* must be idempotent and not update the current state. Therefore, if any corruption of the value of the cell memory occurred, then the corrupted value

Algorithm 1: Find Optimal Instruction For Opcode bit-set (bit-reset) Fault Models (F2, F3)

Input: Instruction set I, Instruction size $size$
Output: The optimal opcode op_{best}.

```
/* Get all encodings w.r.t I sorted by number
   of bits already set (reset) */
E ← GetEncodingsSorted(I);
op_best ← ∅; max ← 0; d ← 0;
/* While max is below the maximum PR at hamming
   distance d */
while max < 1 − d/size do
    j ← 0;
    /* While number of bits already set (reset)
       are less or equal to d */
    while Count(E[j]) <= d do
        /* Get operand bits of encoding E[j] */
        B ← GetOperandBits(E[j]);
        /* Init op_init of encoding E[j] with all
           operand bits reset (set) */
        op_init ← InitAllOperandBits(E[j]);
        /* Generate all bit-sets (bit-resets) on
           B, up to d, minus bits already set
           (reset) */
        C ← Combinations(B, d−Count(E[j]));
        foreach c ∈ C do
            /* Init op_test with operand bits set
               (reset) w.r.t c */
            op_test ← GenOperandBits(op_init, c);
            /* Repeat opcode 2 times, prevent
               idempotent solutions */
            test ← [op_test] ∗ 2;
            stats ← SimulateFI(test, m);
            PR, DR ← GetRates(stats);
            if PR ∗ DR > max then
                max ← PR ∗ DR;
                op_best ← op_test;
        j ← j + 1;
    d ← d + 1;
return op_best;
```

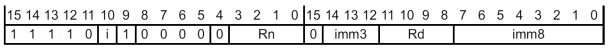

Fig. 2: ARMv7-M Instruction encoding of `ADDW<c> <Rd>, <Rn>, #<imm12>`.

will be propagated ($PR = 1, CR = 1$), and also, the final state will depend on the corrupted bit ($DR = 1$). Unfortunately, an instruction that atomically performs a *load* and a *store* operation does not exist. Instead, two instructions are required, a *store* then a *load* at the same memory address.

*d) **Opcode Corruption (F2, F3)**:*

Overview: with CELTIC and algorithm 1, we can find the optimal instruction i that depends on the instruction set considered. The number of instruction n does not matter and initial values V can be chosen randomly.

Model	Optimal 16-bit Instruction ARMv7-M			
	Opcode	Mnemonic	PR	DR
F2	`3201`	`adds r2, #1`	0.75	1.00
F3	`3fff`	`subs r7, #ff`	0.875	1.00

TABLE V: Optimal 16-bit Instruction from the ARMv7-M Instruction Set.

Algorithm: find optimal instruction i for opcode corruption fault models is more difficult than other families, for several reasons:

- The optimal instruction depends on the instruction set.
- The opcode must be valid according to the instruction set.
- We have to bruteforce the possible combinations.

Accordingly, we propose algorithm 1 to find the optimal instruction for opcode bit-set (bit-reset) fault models. The main idea is that the maximum PR possible for an opcode at hamming distance of d from the initial opcode `0x0...0` (`0xf...f`), w.r.t the instruction $size$, is $1 - d/size$. In addition, the maximum DR is 1 as DR does not depend on d, and $CR = PR$ as the same instruction is repeated multiple times. Furthermore, if we gradually increase the hamming distance d from the initial opcode, and as an opcode at d can theoretically gets a better PR than an opcode at $d+1$, the best opcode is the first one whose $PR * DR$ is greater or equal to $1 - d/size$.

Moreover, the algorithm aims to reduce combinatorial explosion, especially for larger instruction size (32-bit or more). For example, without optimization, the optimal ARMv7-M 32-bit instruction w.r.t bit-set fault models is at a hamming distance of 7 from `0x0...0` (Table VI). This can quickly take times to test every combinations, as $\sum_{k=0}^{7} \binom{32}{k} = 4514873$.

Rather than testing every opcode up to a hamming distance of d, only operand bits B of valid encodings are used. For example, only operand bits `i`, `Rn`, `imm3`, `Rd` and `imm8` of the encoding presented in Figure 2 are considered to generate all the possible bit-sets (bit-resets) up to a hamming distance d from the initial opcode, minus the number of bits already set (reset) of that encoding.

Example: We used an initial instruction subset I from the ARMv7-M instruction set, detailed in Appendix Table XIV. From that initial instruction subset, we can derive *54* 16-bit and *222* 32-bit encodings. The optimal 16-bit and 32-bit instructions from the ARMv7-M instruction set, according F2 and F3 fault models families, found with our algorithm 1, are presented in Table V and Table VI. For reference, in section III, the best evaluated tests have $PR * DR * CR = 0.56$ for F2 (T5 test) and 0.34 for F3 (T1 test). In comparison, our fault characterization tests based on optimal 16-bit instructions (Table V) reach 0.56 and 0.77 respectively.

B. Minimal Set of Fault Characterization Tests

As indirect calibration is a preliminary step, and not an end in itself (in comparison with direct calibration, Table I),

Model	Optimal 32-bit Instruction ARMv7-M			
	Opcode	Mnemonic	PR	DR
F2	`f1400001`	`adc r0, r0, #1`	0.67	0.90
F3	`f57777ff`	`sbcs r7, r7, #0x1fe`	0.70	0.94

TABLE VI: Optimal 32-bit Instruction from the ARMv7-M Instruction Set.

Version	Memory Corruption (MC) Test			

I	`ldr r0, =#0x20002000 ; [r0] = 0x00000000`
	`ldr r2, =#0x20002004 ; [r2] = 0xffffffff`
	`str r1, [r0]`
	`ldr r1, [r0]`
	`str r3, [r2]`
	`ldr r3, [r2]`
	} Repeat *n* times			

V	R0	0x20002000	R1	0x00000000
	R2	0x20002004	R3	0xffffffff
	[R0]	0x00000000	[R2]	0xffffffff

TABLE VII: Memory Corruption (MC) Test for the ARMv7-M instruction set.

the number of test should remain limited to save time for the actual fault attack.

Accordingly, we propose a minimal set of fault characterization tests. The optimal tests found in subsection IV-A can be grouped into three categories, namely instruction corruption (F1-F3), operand corruption (F4, F5) and memory corruption (F6, F7) (Table IV). With only three characterization tests, we cover all the fault model families presented in Table II. The three characterization tests are as follows:

- Instruction Corruption (IC) Test (Table IX) contains an alternation of the optimal instructions generated with `CELTIC` for F2 and F3 fault models families (e.g. Table V), in order to propagate instructions skips and opcode corruption.
- Register Corruption (RC) Test (Table IX) is based on idempotent *mov* instructions which propagate efficiently register value corruption models (F4 and F5 fault models families).
- Memory Corruption (MC) Test (Table VII) can help the propagation of memory corruption (F6 and F7 fault models families) with *load* and *store* instructions.

The code size *n*, for each test, is arbitrarily large as it is easier to inject faults in practice with larger characterization tests. The performance of each test are summarized in Table VIII.

Test	Fault Model Family	*PR*	*DR*	*CR*	$PR*DR*CR$
Instruction corruption	F1,F2,F3	0.66	0.93	0.88	0.54
Register corruption	F4,F5	0.50	1.00	1.00	0.50
Memory corruption	F6,F7	0.50	1.00	1.00	0.50

TABLE VIII: *PR*, *DR*, and *CR* of the IC, RC and MC test.

According to our metrics *PR*, *DR*, and *CR*, the tests we propose get the best performance in each category, instruction, register and memory corruption.

Note that we tried to combine all the optimal instructions found in subsection IV-A into one characterization test, but this drastically reduces *PR*, *DR*, and *CR*. This is mainly due to the fact that the considered instructions are highly specialized for one fault model and not for others. For example, if we combine non-idempotent and idempotent instructions in the same characterization test, it is more difficult to observe distinct instruction skips (F1 fault model family). To put theory into practice, we will conduct indirect calibration with voltage fault injection using this set of tests to validate fault models assumptions and force calibration toward specific fault injection settings.

V. Indirect Calibration with our Minimal Test Set

Our performance metrics are crucial to understanding the fault effects that are propagated and to what extent, for a given fault characterization test, before conducting indirect calibration. Accordingly, we have identified the best minimal test set according to *PR*, *DR*, and *CR*, that covers instruction, register and memory corruption.

In this section, this test set is used to calibrate our voltage fault injection setup for a Cortex-M4 32-bits microcontroller. Furthermore, we detail two possible applications with the IC, RC, and MC fault characterization tests to extend the possible uses of indirect calibration: 1) fault model validation and 2) fault injection settings specialization.

A. Voltage Fault Injection Setup

Voltage fault injection (VFI) is one of possible techniques to inject fault. The main advantages of VFI are that equipment is cheap and almost no target preparation is required. Although for most VFI equipment, only a few parameters need to be set (glitch amplitude and length), the VFI technique proposed by [1] significantly increases the number of parameters considered, but gives more control over the voltage levels and the glitch waveform.

Our VFI test bench is similar to the setup described by Bozzato et al. [1]. We use a custom 30 Msps Digital-to-Analog Converter (DAC) to generate arbitrary glitch waveforms. The waveform of the glitch, sent to the DAC, is generated with a function that takes a set of 8 instantaneous voltage levels, that are then interpolated with cubic interpolation on a grid, up to 2048-by-256, that depends on the waveform size requested. This setup is cheap ($\approx$ 100$) and yet offers great versatility to adapt to different targets with the ability to generate a large spectrum of glitch waveforms, as shown in Figure 3.

B. Target of Evaluation

We have evaluated the IC, RC and MC characterization tests under same conditions on the same target device. We choose an ultra-low-power ARM Cortex-M4 32-bit microcontroller running at 48 Mhz, which implements the architecture ARMv7-M which includes a three-stage pipeline, cache mechanisms and Thumb-2 instruction set compatibility.

Version	Instruction Corruption (IC) Test				Register Corruption (RC) Test			

I	`adds r2, #1`
	`subs r7, #ff`
	`adds r2, #1`
	`subs r7, #ff` } Repeat *n* times
					`mov r0, r0`
					`mov r1, r1`
					`mov r0, r0`
					`mov r1, r1` } Repeat *n* times			

V	R0	0x00000000	R1	0x11111111	R0	0x00000000	R1	0xffffffff
	R2	0x22222222	R3	0x33333333	R2	0x22222222	R3	0x33333333
	R4	0x44444444	R5	0x55555555	R4	0x44444444	R5	0x55555555
	R6	0x66666666	R7	0x77777777	R6	0x66666666	R7	0x77777777

TABLE IX: Instruction Corruption (IC) Test and Register Corruption (RC) Test for the ARMv7-M instruction set.

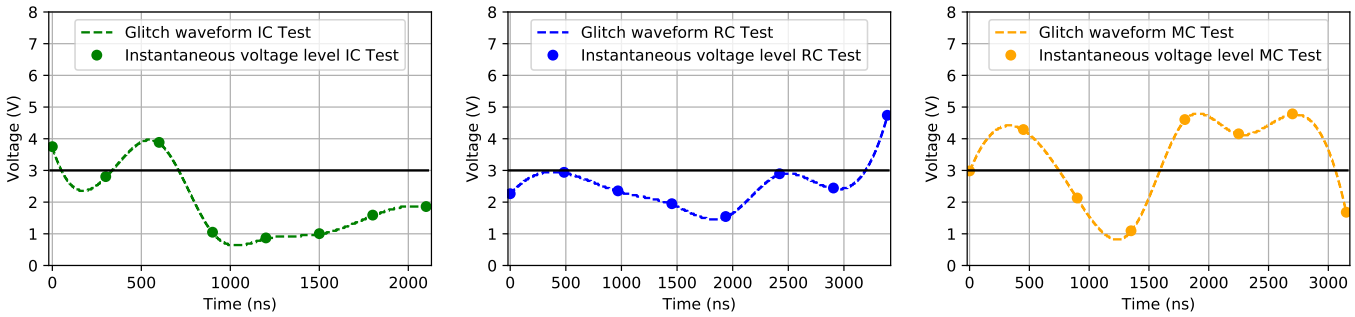

Fig. 3: Glitch waveform with the highest fitness w.r.t IC, RC and MC fault characterization tests, after 600 generations.

C. Experimental Protocol

We conduct an indirect calibration with our minimal test set, using an evolution based optimization method, due to the complexity of the parameter space. The evolutionary algorithm optimizes fault injection parameters, each genome represents a possible set of 8 instantaneous voltage levels, along with the size of the glitch. Bozzato et al. [1] show that they can explore the parameter space and converge to the most interesting settings faster with an evolutionary-based approach than classical grid search. However, the overall performance of fault characterization tests, in terms of *PR*, *DR*, and *CR*, does not depend on the method. We perform 750 generations with a population of 50 individuals. Nearly 200,000 faults have been injected during 12 hours for the IC, RC and MC tests.

D. Results

Experiment results have been summarized in Figure 4, which shows the evolution of fault probability over first 600 generations, for each test; and Figure 3 which details the glitch waveform with the highest fitness (the highest fault probability) for each test. The main takeaways are discussed in the following:

1) Fault Model Validation: Experiment results (Figure 4) are used to validate (or refute) fault model assumptions. Given the properties of propagation and discrimination according to assumed fault models (subsection IV-B) of the IC, RC and MC fault characterization tests, we can deduced that it is unlikely that injected faults corrupt register values, as the fault probability of the RC test is very low (0.15 on average). On contrary, the high probability of fault with the IC test, tells us that it is very likely that injected faults induce instruction skips and/or opcode corruption. As IC test has a $DR = 0.93$, we can distinguish easily one fault model from others. We have investigated and we have found that 90% of the faulty final states have been caused by instruction skips (F1 family) during the indirect calibration with the IC test.

2) Fault Injection Settings Specialization: We use the properties of our fault characterization tests to force indirect calibration toward different settings. Figure 3 presents three different glitch waveforms. These glitches must induce different fault effects, because each test only propagates specific fault models. That means it is possible to act on the indirect calibration results with the fault characterization test design. For example, we get completely different fault effects with the best glitch waveforms found during indirect calibration with the *IC* and *RC* tests, presented in the Figure 3. A set of different fault injection settings (e.g. glitch waveforms) mapped to different fault effects is useful to bypass complex security mechanisms [8].

3) Convergence: For each fault characterization test, within only 100 generations (25,000 fault injections), we identify relevant fault injection settings. For example, in Figure 4, the 66^{th} generation of the IC test is the first generation with average fault probability above 0.6. With a reasonable number of fault injections, the best parameters according to each test are found, reducing the time spent on each test, and therefore on indirect calibration. Accordingly, indirect calibration can be conducted with three different characterization tests even for time-constrained security evaluations. In addition, results can be transferred between microcontrollers sharing same reference, thus the initial time investment spent on indirect calibration quickly pays off. It is possible to reuse the results

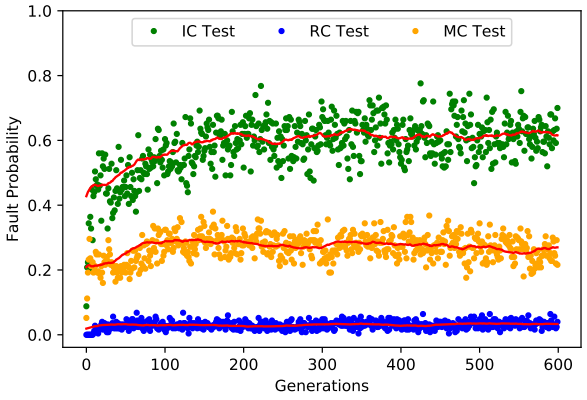

Fig. 4: Evolution of fault probability over 600 generations w.r.t IC, RC, MC fault characterization tests, using voltage fault injection. Red lines are moving averages with 50 generations window size.

to evaluate various sections of the target application, or to assess a completely new application, saving several days of testing.

VI. Conclusion

Fault injection requires preliminary equipment calibration. In this article, we have proposed optimal fault characterization tests in order to fine-tune a given fault injection equipment and to better understand fault effects on the target microcontroller. More precisely, we have evaluated the design influence of fault characterization tests on fault propagation and fault discrimination, which are properties we have defined. From these properties, we have derived performance metrics *PR*, *DR*, and *CR* to compare fault characterization tests from the literature using fault injection simulation, in order to highlight the most important design choices according to assumed fault models. Based on these observations and a simulation-based fault injection tool `CELTIC`, we have found optimal fault characterization tests that maximize *PR*, *DR*, and *CR* for specific fault models. Then, we have combined these optimal tests into a minimal set of three tests which covers a wide range of fault effects that can occur during practical experiments. Our approach to design fault characterization tests is generalizable to any instruction sets, although not shown in this study. Furthermore, we have presented two possible applications of these tests, on a Cortex-M4 32-bit microcontroller, using a state-of-the-art voltage fault injection setup. In less than 20,000 fault injections, we can identify glitch waveforms with high fault probability and validate fault model assumptions. The possibility to converge quickly toward relevant settings can significantly help in carrying out complex multi-fault attacks, which depends a lot on equipment calibration. As future work, it will be interesting to apply our tests on microcontrollers with different instruction sets, and using other fault injection techniques, to investigate on the fault propagation and discrimination with more unusual fault effects. For example, laser fault injection allows the perturbation of specific parts of the microcontroller, in order to induce a wide range of fault effects. Finally, our ongoing research is focused on direct applications of our minimal test set, such as fast equipment calibration using novel optimization approaches.

References

[1] C. Bozzato, R. Focardi, and F. Palmarini, "Shaping the glitch: optimizing voltage fault injection attacks," *IACR CHES*, pp. 199–224, 2019.

[2] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault attacks on rsa with crt: Concrete results and practical countermeasures," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2002, pp. 260–275.

[3] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *International workshop on cryptographic hardware and embedded systems*. Springer, 2002, pp. 2–12.

[4] A. Dehbaoui, J.-M. Dutertre, B. Robisson, and A. Tria, "Electromagnetic transient faults injection on a hardware and a software implementations of aes," in *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2012, pp. 7–15.

[5] S. Anceau, P. Bleuet, J. Clédière, L. Maingault, J.-l. Rainard, and R. Tucoulou, "Nanofocused x-ray beam to reprogram secure circuits," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 175–188.

[6] V. Werner, L. Maingault, and M.-L. Potet, "Fast calibration of fault injection equipment with hyperparameter optimization techniques," in *Smart Card Research and Advanced Applications*, 2022, pp. 121–138.

[7] J. Van den Herrewegen, D. Oswald, F. D. Garcia, and Q. Temeiza, "Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis," *IACR CHES*, pp. 56–81, 2021.

[8] V. Werner, L. Maingault, and M.-L. Potet, "An end-to-end approach for multi-fault attack vulnerability assessment," in *2020 FDTC*. IEEE, 2020, pp. 10–17.

[9] L. Dureuil, M.-L. Potet, P. de Choudens, C. Dumas, and J. Clédière, "From code review to fault injection attacks: Filling the gap using fault model inference," in *International conference on smart card research and advanced applications*. Springer, 2015, pp. 107–124.

[10] J. Balasch, B. Gierlichs, and I. Verbauwhede, "An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus," in *2011 FDTC*. IEEE, 2011, pp. 105–114.

[11] T. Trouchkine, G. Bouffard, and J. Clédière, "Fault injection characterization on modern cpus," in *IFIP International Conference on Information Security Theory and Practice*. Springer, 2019, pp. 123–138.

[12] J. Proy, K. Heydemann, F. Majéric, A. Cohen, and A. Berzati, "Studying em pulse effects on superscalar microarchitectures at isa level," *arXiv preprint arXiv:1903.02623*, 2019.

[13] B. Colombier, A. Menu, J.-M. Dutertre, P.-A. Moëllic, J.-B. Rigaud, and J.-L. Danger, "Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller." *IACR Cryptology ePrint Archive*, vol. 2018, p. 1042, 2018.

[14] I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle, and P. Maistri, "Microarchitecture-aware fault models: Experimental evidence and cross-layer inference methodology," in *16th DTIS*, 2021, pp. 1–6.

[15] L. Riviere, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer, and L. Sauvage, "High precision fault injections on the instruction cache of armv7-m architectures," in *2015 HOST*. IEEE, 2015, pp. 62–67.

[16] T. Korak and M. Hoefler, "On the effects of clock and power supply tampering on two microcontroller platforms," in *2014 FDTC*, 2014, pp. 8–17.

[17] O. Trabelsi, L. Sauvage, and J.-L. Danger, "Characterization of electromagnetic fault injection on a 32-bit microcontroller instruction buffer," in *2020 AsianHOST*, 2020, pp. 1–6.

[18] N. Timmers, A. Spruyt, and M. Witteman, "Controlling pc on arm using fault injection," in *2016 FDTC*. IEEE, 2016, pp. 25–35.

[19] J.-M. Dutertre, T. Riom, O. Potin, and J.-B. Rigaud, "Experimental analysis of the laser-induced instruction skip fault model," in *Nordic Conference on Secure IT Systems*. Springer, 2019, pp. 221–237.

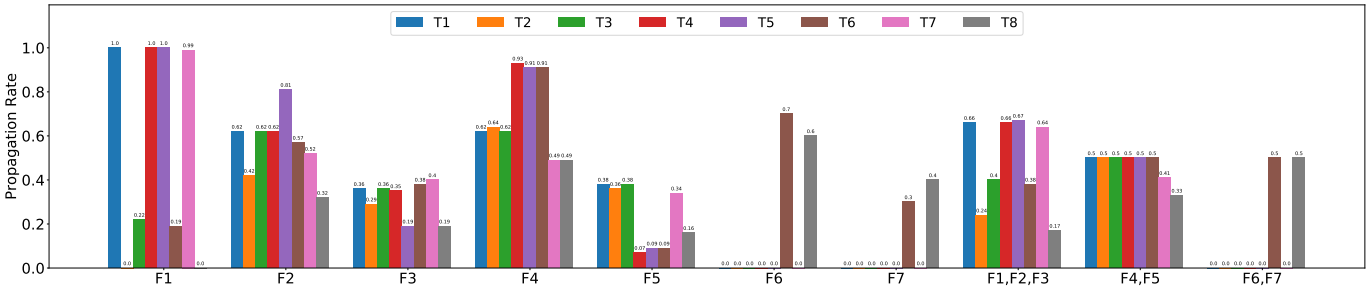

Fig. 5: Propagation Rate of characterization tests in Table III, w.r.t to fault models in Table II.

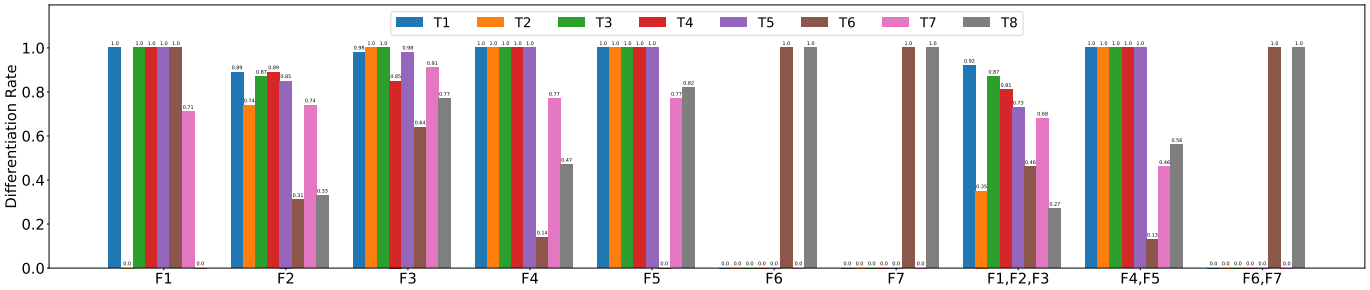

Fig. 6: Discrimination Rate of characterization tests in Table III, w.r.t to fault models in Table II.

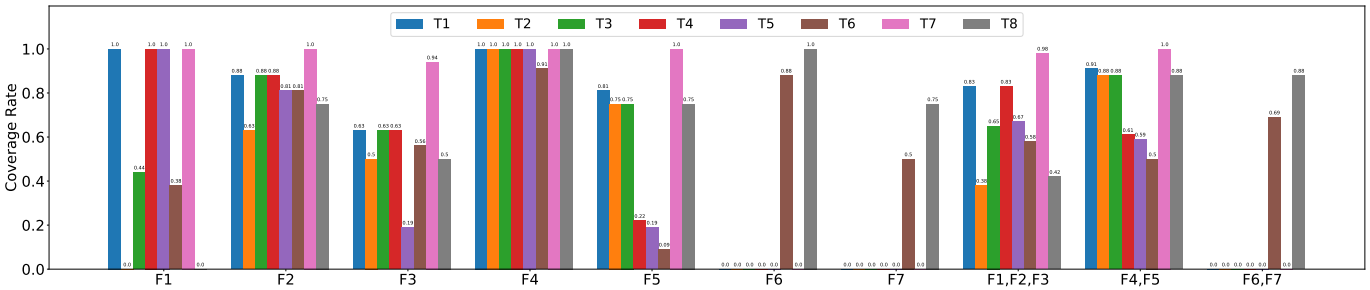

Fig. 7: Coverage Rate of characterization tests in Table III, w.r.t to fault models in Table II.

Version	Characterization Test T1				Characterization Test T2			

I	`adds r0, #2`
	`adds r1, #3`
	`adds r2, #5`
	`adds r3, #7`
	`adds r4, #11`
	`adds r5, #13`
	`adds r6, #17`
	`adds r7, #19`
	} Repeat *n* times
					`mov r0, r0`
					`mov r1, r1`
					`mov r2, r2`
					`mov r3, r3`
					`mov r4, r4`
					`mov r5, r5`
					`mov r6, r6`
					`mov r7, r7`
					} Repeat *n* times			

V	R0	0x00000000	R1	0x11111111	R0	0x00000000	R1	0x11111111
	R2	0x22222222	R3	0x33333333	R2	0x22222222	R3	0x33333333
	R4	0x44444444	R5	0x55555555	R4	0x44444444	R5	0x55555555
	R6	0x66666666	R7	0x77777777	R6	0x66666666	R7	0x77777777

TABLE X: The characterization tests T1 and T2

Version	Characterization Test T3				Characterization Test T4			

I	`adds r0, #2`
	`adds r1, #3`
	`adds r2, #5`
	`adds r3, #7`
	`adds r4, #11`
	`adds r5, #13`
	`adds r6, #17`
	`adds r7, #19`
					`adds r0, #2`
					`adds r1, #3`
					`adds r2, #5`
					`adds r3, #7`
					`adds r4, #11`
					`adds r5, #13`
					`adds r6, #17`
					`adds r7, #19` } Repeat *n* times			

V	R0	`0x00000000`	R1	`0x11111111`	R0	`0x00000000`	R1	`0x00000000`
	R2	`0x22222222`	R3	`0x33333333`	R2	`0x00000000`	R3	`0x00000000`
	R4	`0x44444444`	R5	`0x55555555`	R4	`0x00000000`	R5	`0x00000000`
	R6	`0x66666666`	R7	`0x77777777`	R6	`0x00000000`	R7	`0x00000000`

TABLE XI: The characterization tests T3 and T4

Version	Characterization Test T5				Characterization Test T6			

I	`adds r0, #2`
	`adds r0, #2`
	`adds r0, #2`
	`adds r0, #2`
	`adds r0, #2`
	`adds r0, #2`
	`adds r0, #2`
	`adds r0, #2` } Repeat *n* times
					`ldr r0, =array ; [0,1,2,3,...]`
					`ldr r1, [r0]`
					`ldr r2, [r0, #4]`
					`ldr r3, [r0, #8]`
					`ldr r4, [r0, #12]`
					`ldr r5, [r0, #16]`
					`ldr r6, [r0, #20]`
					`ldr r7, [r0, #24]`			

V	R0	`0x00000000`	R1	`0x11111111`	R0	`0x20005000`	R1	`0x11111111`
	R2	`0x22222222`	R3	`0x33333333`	R2	`0x22222222`	R3	`0x33333333`
	R4	`0x44444444`	R5	`0x55555555`	R4	`0x44444444`	R5	`0x55555555`
	R6	`0x66666666`	R7	`0x77777777`	R6	`0x66666666`	R7	`0x77777777`

TABLE XII: The characterization tests T5 and T6

Version	Characterization Test T7				Characterization Test T8			

I	`sub r7,r5`
	`add r8,r4`
	`subs r0, #0xff`
	`...`
	`lsl r1, r5, #8`
	`mov r2,r6` } *n* standard data-processing instructions
					`ldr r0, =array ; [r1,r2,...]`
					`str r1, [r0]`
					`ldr r1, [r0]`
					`...`
					`str r7, [r0, #24]`
					`ldr r7, [r0, #24]` } Repeat *n* times			

V	R0	`0x00000000`	R1	`0x11111111`	R0	`0x20002000`	R1	`0x11111111`
	R2	`0x22222222`	R3	`0x33333333`	R2	`0x22222222`	R3	`0x33333333`
	R4	`0x44444444`	R5	`0x55555555`	R4	`0x44444444`	R5	`0x55555555`
	R6	`0x66666666`	R7	`0x77777777`	R6	`0x66666666`	R7	`0x77777777`

TABLE XIII: The characterization tests T7 and T8

Instruction	Description	ARMv7-M Opcode Example	
		Opcode	Mnemonic
ADC	Add with carry	0xEB410503	adc.w r5, r1, r3
ADD	Add	0xEB010203	add.w r2, r1, r3
ADR	Form PC-relative address	0xA301	adr r3, #4
AND	Bitwise AND	0xF402497F	and sb, r2, #0xff00
BIC	Bitwise bit clear	0xF02100AB	bic r0, r1, #0xab
EOR	Bitwise exclusive OR	0xF09B3718	eors r7, fp, #0x18181818
MOV	Copies operand to destination	0x0008	movs r0, r1
MVN	Bitwise NOT	0xEA6F0807	mvn.w r8, r7
ORN	Bitwise OR NOT	0xEA6B173E	orn r7, fp, lr, ror #4
ORR	Bitwise OR	0xEA400205	orr.w r2, r0, r5
RSB	Reverse subtract	0xF5C464A0	rsb.w r4, r4, #0x500
SBC	Subtract with carry	0x418B	sbcs r3, r1
SUB	Subtract	0xF1B608F0	subs.w r8, r6, #0xf0
ASR	Arithmetic shift right	0xFA48F709	asr.w r7, r8, sb
LSL	Logical shift left	0x4091	lsls r1, r2
LSR	Logical shift right	0xFA25F406	lsr.w r4, r5, r6
ROR	Rotate right	0xFA65F406	ror.w r4, r5, r6

TABLE XIV: The 17 Instructions considered.