# Software-Only Control-Flow Integrity Against Fault Injection Attacks

François Bonnal, Vincent Dupaquis, Olivier Potin, Jean-Max Dutertre

Authors' version
DSD 2023
DOI: 10.1109/DSD60849.2023.00046

# Software-only Control-Flow Integrity against Fault Injection Attacks

François Bonnal*, Vincent Dupaquis*, Olivier Potin† and Jean-Max Dutertre†
*Trusted-Objects
Aix-en-Provence, France
Email: f.bonnal@trusted-objects.com, v.dupaquis@trusted-objects.com
†Mines Saint-Etienne, CEA, Leti, Centre CMP, F - 13541 Gardanne France
Email: olivier.potin@emse.fr, dutertre@emse.fr

***Abstract*—In this paper, we introduce a new Control-Flow Integrity (CFI) scheme for detecting Fault Injection Attacks (FIA). Our scheme is designed to be as generic as possible and to cover any microcontroller on the market, including non-secure ones. It is a full software approach, designed to detect CFI disruptions caused by FIA. The proposal is portable and designed for a high-level language implementation (C in our case). The main characteristic of our scheme is to link a predictable computed Chain of Trust (CoT) with the assets of a program. This approach classically allows the detection of fault injections leading to an illegitimate path of execution. In addition, this solution is designed to detect when a legitimate execution path is wrongly followed due to FIA. Simulations on several benchmarks finally validate the effectiveness of the method, using a multiple instruction skip faults model.**

***Index Terms*—Control flow integrity, Fault injection attack, Multiple faults**

## I. Introduction

The Internet of Things (IoT) provides an increasing number of devices with connected capabilities, leading to several examples of attacks [12], [24], and thus to new security challenges. Non-secure microcontrollers are widely used for IoT devices due to their low-cost and low-power design, they are vulnerable to FIA which can be done with low expertise and equipment as demonstrated in [7], [9]. To mitigate this threat under the constraint of limited resources, we may consider hardware countermeasures, however, in most cases, these are not suited to deal with the security needs of an already existing microcontroller. To give a generic answer to these security needs, we should not only rely on hardware security features, instead, a software-only approach is cheap to integrate and presents opportunities of evolution. In particular, we choose to insert our countermeasures at the source code level using a high-level language (C99).

In this work, we focus on Control Flow Integrity (CFI) enforcement in order to detect FIA aiming at modifying the execution flow of the target's program. FIA may for instance induce a test inversion allowing a privilege escalation despite entering a wrong password (as shown in section VI). Our solution monitors at runtime the followed execution path which must be consistent with the control flow graph (CFG) and the input data. This software-only CFI solution is based on a Chain of Trust (or CoT) which is a chain of pre-computed values that change according to the program's execution path. Chain values can be monitored at any point of program execution to verify that the execution path is legitimate and correct. Our main design goal is to mesh the program's data with values of the CoT. This CoT is established at compilation time, and then computed and verified during execution.

FIA may result in various faulty behaviors (e.g. data corruption, instruction skip, etc.) [7]. We specifically considered the instruction skip fault model, which consists in preventing an assembler instruction from being executed. The targeted instruction is said to be skipped but is rather corrupted in a way that is similar to erasing it at run-time [8], [25]. This behavior is usually simulated by substituting the original instruction with a no-operation instruction (or `nop`, a legitimate instruction that has no effect on the target computations). Much research papers consider single instruction skip (a unique instruction is skipped), however multiple skips targeting several instructions (consecutive or at different stages of a program) can also be obtained. This latter fault model is difficult to thwart as one fault iteration can be induced in order to deactivate a FIA countermeasure [7]. As described below, the CFI solution we propose is designed to tackle with multiple instruction skips.

The main contributions of this research are:

- Software CFI scheme based on a CoT
- Computation of CoT values from the program data
- Computation of program data from the CoT values
- Detection of simple and double instructions skips

In this paper, we first remind the CFG and CFI notions in section II. Then, section III presents a survey of previous works about CFI and their limits against FIA. In sections IV and V the design and mechanisms of our method are explained. Section VI presents our experiments and results on key examples for CFI enforcement. Lastly, we conclude and present our ideas to further improve our approach.

## II. Background

### A. Control Flow Graph

A Control-Flow Graph (CFG) describes the various paths of execution a program may follow at run-time. The graph's nodes are Basic Blocks (BB), i.e. sequential sections of code

which may start with a jump target and may end with a jump instruction. High-level languages usually define a specific structure to delimit each BB [11]. An example of a CFG representing a `switch case` program over a Key_Size (KS) value is given in Table I and Fig. 1. The KS value, either 128 or 256 leads to two legitimate paths of execution, any other value is caught by the default case as an error.

TABLE I
SWITCH CASE C CODE

| Basic Block | C code |
|---|---|
| $BB_0$ | switch(KS) { |
| $BB_1$ | case 128: |
| | get_key128(); |
| | break; |
| $BB_2$ | case 256: |
| | get_key256(); |
| | break; |
| $BB_3$ | default: |
| | error(); |
| $BB_4$ | } |
| | if (!encryption()) error(); |
| $BB_5$ | if (!encryption_check()) error(); |

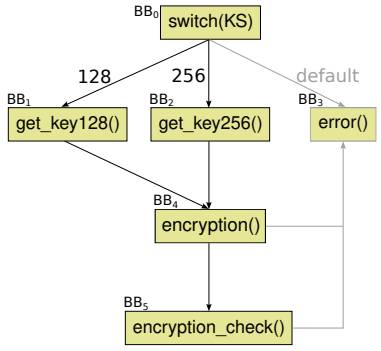


Fig. 1. Control Flow graph

### B. Control Flow Integrity

In the presence of FIA, CFI ensures that a legitimate (present in the CFG) and correct (corresponding to the input data) path of execution is taken. Assuming that KS is equal to 256, the correct path of execution must follow $BB_0 \rightarrow BB_2 \rightarrow BB_4$. As a consequence, $BB_0 \rightarrow BB_1 \rightarrow BB_4$ is a legitimate but incorrect execution path (given the assumption that $KS = 256$) and any other path is illegitimate (e.g. $BB_0 \rightarrow BB_4$).

## III. Previous Works

In previous CFI works, a first category protects against software attacks based on code reuse attacks, such as return-oriented programming [23] and jump-oriented programming [2]. Software based [5] solutions as well as hardware based [26] solutions have been proposed for this kind of threat. However, these CFI schemes are not suited to counter FIA threats which are relatively easy and affordable to mount against low performances devices [9], [17]. FIA's ability to disrupt the execution flow leads to a second category of CFI schemes [4], [11], [14], [16], [19], [21] which ensures that the path of execution and the CFG of the program match. Each BB is given a static signature value that is dynamically recomputed and verified during execution to check that a correct execution path has been followed (see Fig. 2 and Fig. 3). In [11], two variables are used to ensure CFI: the first tracks the current BB's signature. The second variable accumulates errors that come from both signature verification and decision verification. In [14],several counters are used to track the CFG paths, and parallel execution paths are protected with different counters. For both schemes, when a decision is made (a test on a variable leading to several possible BB such as `switch(KS))`, each possible path of execution leads to a different state of the tracking variables set. The detection relies on a re-evaluation of the decision and the verification that it matches the state of the tracking variables set. These mechanisms target at protecting the decisions of the program against single FIA.

Recently, Proy et al proposed a different approach in [20], its aim is to protect the call graph of the program which is a sub-graph of the CFG. This method protects directs function calls in addition to their entry parameters: two tracking variables are respectively updated before and after the function's call, they are eventually compared to ensure that the intended function call has been performed. This protection is a lightweight CFI scheme restricted to function calls and ignoring any other CFG decision making. However, its ability to link the entry parameters to the CFI scheme is an important improvement to the state-of-the-art.

Fig. 2 provides a synthetic view of the main state-of-the-art CFI schemes (such as [11], [14]) applied to CFG of a simple `switch case` program (as illustrated in Table I and Fig. 1). The Tracking Variable (or TV) is a generalization of the approaches in [14] (using a set of counters) and [11] (using a set of BB signatures), it takes $n+1$ values denoted $T_0, ..., T_n$ which are mapped to $n+1$ BB. This TV is used to enforce the CFI of the protected code by updating its value (to keep track of the executed BBs) or by checking that its current value is set according to the code CFG.

In this example, only the protection of the `switch case` decision is considered for our demonstration. For instance, considering an instruction skip fault model, a skip of a branch instruction can result in the execution of the wrong `switch case`, i.e. a corruption of the decision made. The protection relies on the verification of both the validity of the TV and its matching with KS (($T_1$,128) or ($T_2$,256)). This mechanism ensures that a single FIA corrupting the KS's decision will be detected during the check operation before executing the encrypt() function, for instance, if TV is equal to $T_2$, we expect the execution path to go through $BB_2$ and in this case KS should be equal to 256, any discrepancy will be interpreted as a fault being successfully injected.

The main limitation of software state-of-the-art CFI is the lack of dependencies between TV values and program's decisions, these TV values are mapped to the program's CFG,

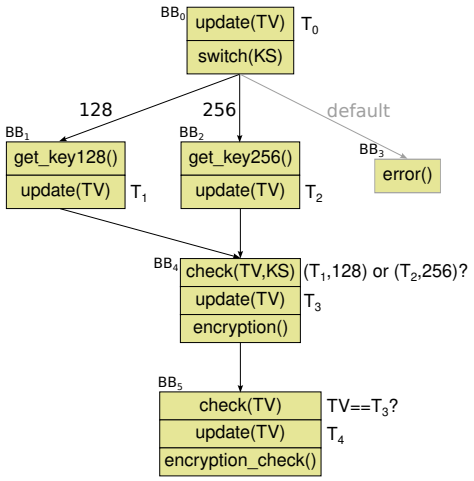


Fig. 2. State of the art CFI

which means that those remain correct in case of a legitimate but incorrect path of execution due to a FIA (e.g. $BB_1$ is executed with KS being 256, a less secure 128-bit key could be used, as exemplified in Fig. 3). In this case of legitimate but incorrect path of execution, the protection entirely relies on a single check test which will detect a single FIA. However, a stronger attacker capable of multiple FIA [8] could both corrupt the decision over KS and skip the check operation in order to entirely defeat the protection: because there is no dependency between the TV and the KS value, skipping the check test following a decision's corruption would let the program correctly update the TV from $T_1$ to $T_3$. The expected detection being avoided by a second fault, any upcoming test would not detect the attack due to the absence of error propagation as depicted in Fig. 3.

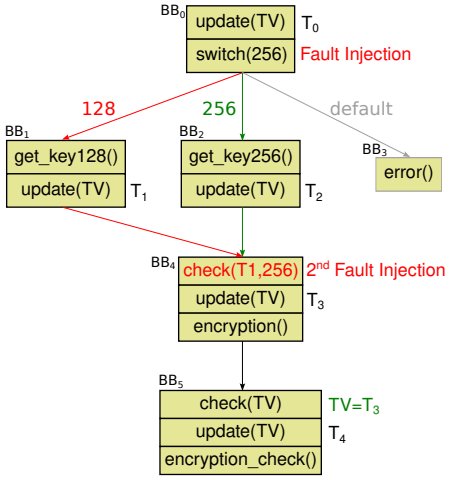


Fig. 3. State-of-the-art CFI and vulnerability to multiple FIAS

## IV. Design of the protection

The objective of our proposal is to deepen as much as possible the dependencies between the TV and the decisions of the program. Subsection IV-A presents the core mechanism used to assign BB's signatures. Then, subsection IV-B presents our first improvement mechanism to state-of-the-art CFI.

### A. Chain of Trust

The general idea of the protection is to implement a CoT: a sequence of values which current state is stored into a variable, and map these values to the program's CFG. Each step value of the CoT is recursively computed with a transition function, i.e. each step value is chained with the previous one through the transition function. The transition function and CoT expected properties are:

- The transition function must provide a long enough series of distinct values to avoid collisions.
- Each function or relevant part of the software should have its own CoT. For this purpose, each CoT will start with a distinct and arbitrary seed.
- Except when seeding the CoT, the current state must never be set to a value, it must be computed from the previous one.

For a given function composed of n+2 BB, its CoT is a set of n+2 values $C_0, ..., C_n, C_f$, the seed $C_0$, being an arbitrary value unique to this function, and $C_f$ the final value, which is expected to be reached in case of correct execution. Any update of the CoT value is always made through a call of the transition function `TFunc` with a transition value Tvalue as follows:

$$CoT \leftarrow TFunc(CoT, Tvalue) \tag{1}$$

These transition values are pre-computed during a pre-processing step (see section VI-A), in particular, the transition step values TOSTEP(i) are arbitrary and used for the `next_step` operation [3] which updates the CoT from $C_{i-1}$ to $C_i$. The complete definition of a n-length CoT is:

$$\begin{cases} C_0 = SEED \\ \forall i \in [\![1, n]\!], C_i = TFunc(C_{i-1}, TOSTEP(i)) \\ \forall i \in [\![0, n]\!], C_f = TFunc(C_i, TOFINAL(i)) \end{cases} \tag{2}$$

The final value $C_f$ can be computed from any CoT step, including from $C_0$, this is particularly interesting when a caller function has to check the resulting CoT value of a called function (see section VI-A). For this purpose, $C_f$ is defined directly from $C_0$ and an arbitrary value named KEY_CFI, it is computed at execution time from $C_i$ using a constrained transition value TOFINAL(i) (in particular, TOFINAL(0) is equal to KEY_CFI):

$$\begin{cases} C_f = TFunc(C_0, KEY\_CFI) \\ \forall i \in [\![0, n]\!], C_f = TFunc(C_i, TOFINAL(i)) \end{cases} \tag{3}$$

During the pre-processing phase, for all i in the set of defined exit points of the CoT, TOFINAL(i) will be computed such that it allows the transition from $C_i$ to $C_f$. This mechanism defines the starting and ending values of the CoT without

knowledge of its internal complexity, while during execution, every CoT value is computed from the previous one through the transition function `TFunc`.

Several implementations of the transition function may be considered, we only focus here on a simple and yet efficient XOR operation $\oplus$:

$$TFunc(CoT, Tvalue) = CoT \oplus Tvalue \quad (4)$$

### B. Decision's protection

The above-described mechanism allows protecting a sequential section of code. In order to enforce the decision-making we need an additional functionality that relies on the CoT. Protecting decisions is done with a `feed/compensate` (or CPS) mechanism, which involves inserting the decision value into the CoT computation. This allows, for instance, to split and merge back a CoT to protect parallel paths of executions as depicted with our `switch case` example in Fig. 4. The computation details of each operation present in Fig. 4 are given in Table II, in particular:

$$OutputValue = InputValue \oplus TransitionValue \quad (5)$$

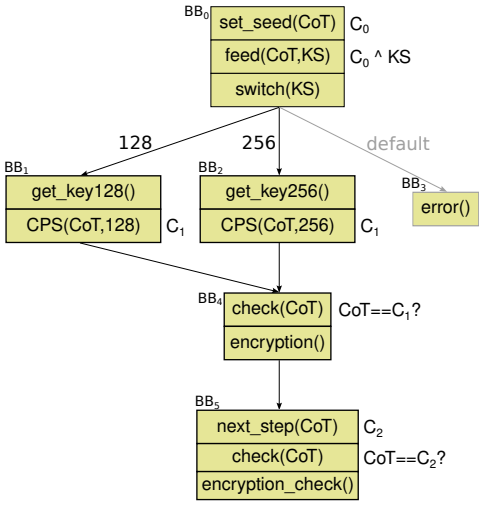


Fig. 4. CoT CFI

TABLE II
DETAILED STEPS OF OPERATION USING THE XOR TRANSITION FUNCTION.

| Operation | Input value | Transition value | Output value |
|---|---|---|---|
| set_seed(CoT) | Null | $C_0$ | $C_0$ |
| feed(KS) | $C_0$ | KS | $C_0 \oplus$ KS |
| CPS(128) | $C_0 \oplus 128$ | $C_0 \oplus 128 \oplus C_1$ | $C_1$ |
| CPS(256) | $C_0 \oplus 256$ | $C_0 \oplus 256 \oplus C_1$ | $C_1$ |
| next_step(CoT) | $C_1$ | TOSTEP(2) | $C_2$ |

KS is used both for the CoT computation and the condition's evaluation: instead of mapping $BB_1$ and $BB_2$ to static values $T_1$ and $T_2$, these BB are mapped to values derived from $C_0$ and the decision value KS. This is done through the `feed` operation which allows diverging from the nominal CoT by merging the decision value (KS) into it. The `compensate` operation allows converging back into the expected CoT only if the right decision about KS has been made and executed. More precisely, the transition value involved in the `compensate` operation is computed to go from $C_0 \oplus$ KS to $C_1$ for each expected value of KS. This `feed/compensate` mechanism comes as a replacement for a `next_step` operation to transition from $C_0$ to $C_1$, which means that skipping these operations would result in an invalid CoT (missing an update). A mismatch between the fed value and its compensation counterpart would result in a corrupted CoT and therefore be detected in all subsequent CoT tests. With this mechanism, faulting the decision and then the check operation would not defeat the protection because the CoT value would remain incorrect and transmit this error to the next verification as depicted in Fig. 5. This principle is applicable to any data that is known at compilation time. Using this protection enforces the execution and decision of even complex parts of code by establishing links between program's important data (assets) and the sequence's correctness.

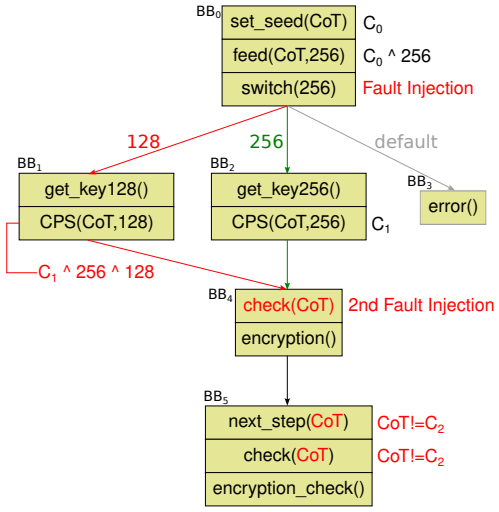


Fig. 5. CoT CFI and FIA

## V. COT DERIVATION

In previous sections, we have shown the error propagation properties of the CoT, allowing to capture a fault even long after it occurred, and a mechanism to compute the CoT from program's data. To further increase the link between the CoT and the program execution, we describe a new mechanism allowing to compute a data from the CoT. Fig. 6 shows an application of this mechanism to the protection of a function call.

In Fig. 6, the `feed/compensate` mechanism allows the detection of an unauthorized or corrupted execution of function G. Both functions F and G have their own CoT, CoT_G is fed into CoT_F, then CoT_G's expected value $CG_{final}$

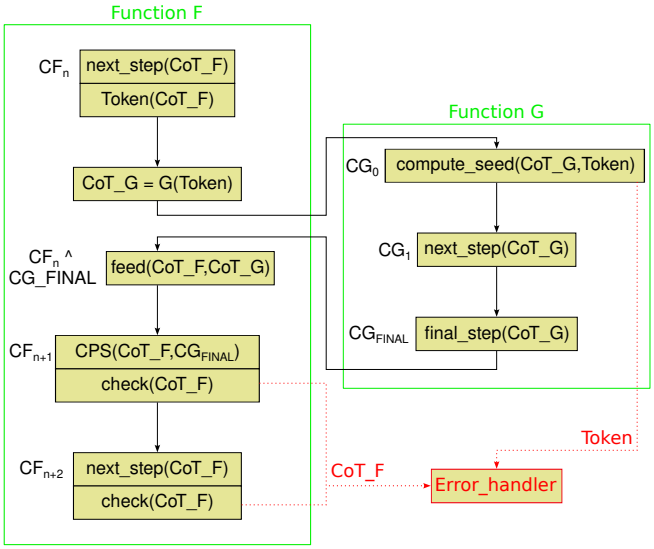


Fig. 6. FCall benchmark

is compensated, allowing F to integrate in its own CoT all potential misbehavior reported by CoT_G during its execution. In particular, this mechanism ensures that a missing execution or the execution of another function is detected (forward and backward edge protection). Considering the case G is a critical function, it is indeed mandatory to perform this detection *a priori*, as an *a posteriori* detection is potentially happening too late.

To perform an *a priori* detection, we propose providing an execution token to function G, computed from CoT_F. Through the `compute_seed()` (see Fig. 6) function, function G checks the execution token and uses it to initialize its CoT or redirect the program's control flow towards an error handler (in the case where G was not intended to be called or function F has been disrupted by a fault).

$$\text{TOKEN} = \text{CG}_0 \oplus \text{CoT\_F} \oplus \text{CF}_n \tag{6}$$

This execution token is a computation of G's seed ($\text{CG}_0$) under the condition that CoT_F takes its expected value $\text{CF}_n$. Not only an indirect verification of the caller's CoT is made before starting G's execution, but in case an attacker skips this verification, CoT_G would carry the potential error due to its assignation to token. In Fig. 6, we represent the conditional access to function G from the $n^{th}$ step of function F, this mechanism prevents the execution of function G outside a set of chosen contexts.

Computing a program data from the CoT opens several other interesting applications, for instance, it is possible to protect data through a contextual masking, which will only reveal the correct unmasked value if the CoT is correct. Extending this idea to contextual use of critical data, it enables limiting their use to very specific pieces of code, any out-of-context access to these masked data resulting in getting back a corrupted (unusable) version of them. This mechanism further increases the dependency between the CoT computation and the program's execution, not only a corrupted CoT due to FIA would lead to a detection, but in case of a skipped detection, the program itself would not proceed correctly in its execution due to corrupted data and denied function's execution.

## VI. Experiments and Results

### A. Insertion of the protection

The protection insertion is made at the source code level, through a set of manual and automated steps. A preliminary vulnerability analysis is performed in order to determine the assets (e.g. KS in Fig. 1) to protect, then, the CoT is manually inserted and linked to these assets through a set of macros. Involving a security expert for identifying the places and assets to protect is mandatory to ensure that primary assets are protected. Insertion process is available at [3], it concerns three steps:

1) The program is pre-compiled in order to expand the macros, which defines every CoT value that need pre-computation (mainly transition values).
2) A python script computes the CoT values.
3) Finally, the compilation process restarts, using the newly computed values.

It is possible to implement our protection without the preprocessing step; however, it was convenient to do so in order to switch easily between transition functions with diverse mathematical properties.

As previously mentioned the final value $\text{C}_f$ is defined from the seed $\text{C}_0$ and dynamically computed from a step $\text{C}_i$ during execution: the consequence is that a CoT complexity has no impact on its final value. Encapsulating a CoT into another one (to handle for instance a function call) is trivial, one must simply `feed` the computed CoT value and `compensate` the expected $\text{C}_f$ value. This eases the process of inserting countermeasures into a given program, as well as the maintenance of the code because any modification on a given CoT does not have any impact on the encapsulated or the encapsulating CoT. For instance, in Fig. 6, adding an arbitrary number of steps to function G does not change the expected final value $\text{CG}_{final}$ and as a consequence does not have any impact on the computation of CoT_F.

### B. Tested code and attack scenarios

For the purpose of testing our protection method, we used an Arm® Cortex®-M3 fault injection simulator based on unicorn [15]. This simulator allows observing the effects of several instruction skips on the execution of a program. Three different faults models are implemented:

- Single instruction skip: the simulator lists every instruction present in a reference execution of the program, then every possible substitution of an instruction with a `nop` is tested. For every substitution scenario, if an asset is altered without detection, the scenario is deemed an "attack success".
- Consecutive instruction skips [10] are similarly handled than single instruction skip, once the address of the

first instruction to be skipped is determined, multiple consecutive instructions are replaced with a "nop".
- Double instruction skips are based on single instruction skip scenario, for each scenario resulting in a fault detected, every possible instruction skip of a second instruction are tested in an attempt to bypass the detection.

The benchmarked functions are the following:

- `switch()`: as presented in sections II, III and IV.
- `FCall()`: as presented in section V.
- `min_array()`: Presented in [11] to illustrate the CFI protection named YACCA. It compares term-by-term two arrays and stores the smallest element into a third array.
- Memory access functions, i.e. functions used to manipulate stored data in memory: `memcmp()`, `memcpy()` and `memset()`. These functions are involved in critical operations such as erasing critical data or comparing two buffers to make a decision over the comparison result (e.g. a signature verification).
- `VerifyPin()`: it consists in testing a four-digit user code and comparing it with a reference four-digit PIN code (as described in [6]), a counter tracks the number of attempts to avoid brute force attacks.
- `ECDSA_verify()` is an open source implementation [22] of `ECDSA` 's signature verification function (Elliptic Curve Digital Signature Algorithm).
- `ECDSA_sign()` is an open source implementation [22] of `ECDSA`'s signature function.

For these functions we considered the followings attack scenarios:

- `switch()`: The attack is successful if the loaded key is incorrect without detection.
- `FCall()`: The aim of the attack is to obtain a correct return code for the function F with an incorrect return code for the function G.
- `min_array()`, `memset()`, `memcpy()`: We compare the faulted execution's resulting array with a reference one, if there is a difference and no detection, the attack is pass.
- `memcmp()`: two arrays with one different term are compared, if the comparison succeeds the attack is considered a success.
- `VerifyPin()`: We tested two different scenarios with this example code and several versions with different countermeasures as proposed in [6] (results are shown for `VerifyPin7`, but we tested versions zero and one as well):
  - The first scenario of attack consists into making an incorrect pin code being accepted by the verification process.
  - The second one consists into bypassing the maximum number of attempts resulting in a fast brute force attack. We consider that the attacker enters the pin code and that he has zero attempts left (if the maximum number of attempts can be bypassed a brute force attack is then trivial).
- `ECDSA_verify()`: The aim of the attack is to make an incorrect signature accepted.
- `ECDSA_sign()`: based on the attack described in [1], this attack has a simple countermeasure that consists in verifying that the point P is on the elliptic curve [18]. In this scenario, we assume that the elliptic curve parameters have been compromised by the attacker which aims to bypass the countermeasure with fault injection. If a fault injection allows getting the corrupted signature in output, then according to [1] we assume that a further cryptanalysis allows recovering information about the secret key and the attack is pass.

### C. Results

In Table III, we show the number of attack successes for each attack scenario previously introduced with and without our CFI scheme in the presence of a single instruction skip. Every optimization option present with the GCC compiler were tested but only the -O3 optimization results are presented. The -O3 option is one of the most aggressive levels of optimization (with -Os) and it is the most unfavorable option for our countermeasures, the results with others levels of optimization would not add any additional "attack success" scenario.

TABLE III
SIMPLE INSTRUCTION SKIP ATTACK

| Scenario | attack success | |
|---|---|---|
| | no CFI | CFI |
| `switch()` | 276 | 0 |
| `FCall()` | 3 | 0 |
| `memset()` | 12 | 0 |
| `memcpy()` | 16 | 0 |
| `memcmp()` | 3 | 0 |
| `VerifyPin()` Password | 2 | 0 |
| `VerifyPin()` Counter | 3 | 0 |
| `ECDSA_verify()` | 4 | 0 |
| `ECDSA_sign()` | 3 | 0 |

Table IV presents the overhead of our protection for the presented scenarios, this overhead is highly variable, from negligible (`ECDSA` scenario) to a thirty-five time increase (`FCall()` scenario). This high variability comes from the fact that the overhead fully depends on the protected assets manipulations. For instance, `memcmp()` has a high overhead, as almost every line of code needs to be protected. However, in `ECDSA` examples, only a small portion of the code is involved in the attack and the overhead is marginal. In the `switch()` example we consider only one asset to protect which is the key size, its protection comes with a low overhead but as it is a simplified example, pushing the details even further would certainly add other assets to protect and an higher overhead.

TABLE IV
OVERHEAD: CODE SIZE AND NUMBER OF CLOCK CYCLES

| Scenario | code size | | clock cycles | |
|---|---|---|---|---|
| | no CFI | CFI | no CFI | CFI |
| `switch()` | 200 | 384 | 528 | 580 |
| `FCall()` | 32 | 500 | 4 | 142 |
| `memset()` | 14 | 448 | 55 | 538 |
| `memcpy()` | 98 | 476 | 40 | 620 |
| `memcmp()` | 66 | 552 | 18 | 630 |
| `VerifyPin()` Password | 36 | 108 | 20 | 145 |
| `VerifyPin()` Counter | 36 | 108 | 11 | 67 |
| `ECDSA_verify()` | 27808 | 28472 | 9.8M | 9.8M |
| `ECDSA_sign()` | 27704 | 28344 | 8.5M | 8.5M |

The scenario `min_array()` comes from [11] as an illustration of the YACCA CFI method. We tested this protection (enriched version from [11]) with our simulator as described in Table V. We declared the variables manipulated by YACCA as `volatile` [13] otherwise the execution time would be significantly lower (550 vs 1116 cycles) but the number of attack success would be significantly higher (62 vs 25). Under the same test conditions we can see that with our method we were able to cover every attack success scenario with a much lower overhead than YACCA (507% vs 1298%). In the given example, the minimum term by term of arrays a and b is stored into a third array x. YACCA ensures that the path of execution is legitimate, and the decisions are correctly made, which means that the correct value between a[i] and b[i] is chosen and the corresponding BB is executed. But an asset is not protected: the resulting array x, the assignation of x[i] to a[i] or b[i] is easily corrupted with our fault model.

TABLE V
ENRICHED YACCA VS COT

| Scenario | attack success | | | clock cycles | | |
|---|---|---|---|---|---|---|
| | no CFI | CFI | YACCA | no CFI | CFI | YACCA |
| `min_array()` | 56 | 0 | 25 | 86 | 436 | 1116 |

Memory functions manipulate critical assets in more complex programs, they represent an interesting example as protecting them at all cost can drastically increase the security of a program without too much global overhead despite an important local one. These functions are missing a context of execution as for instance, `memcpy()` is able to write arbitrary data to nearly any address, this is an opportunity for an attacker to extract, insert or erase critical data. This design issue led us to give a special attention to these functions, in every scenario we attacked the function's call and its parameters along with the internal part of the function. We redefined the memory functions to take a C99 structure data type as parameter (see [3]), this structure contains the previous parameters as well as an integrity data which allowed us to counter every attack scenario even in the most unfavorable case where faults are injected while building the parameter's structure.

In order to highlight the conditional execution of a function described in section V, we tested a variation of the second attack scenario for `VerifyPin()`. We still considered an attacker in possession of the pin code with zero attempts left and trying to bypass this protection with a FIA. But, in this case we did not want the program to execute the comparison function at all. Previously, the attack detection by the CFI protection would be sufficient to consider the scenario as "fault detected", we consider here as an "attack success" any case where the comparison function is executed, even with the CoT detecting an attack later on. This leads to two new attacks scenarios uncovered with the `feed/compensate` mechanism only. The implementation of the mechanism described in section.IV-B allows covering the two mentioned "attack success" scenarios.

TABLE VI presents our multiple fault results following the fault models described in Section VI-B. Consecutive instruction skips have been studied in previous works [10] and the results of our protection against such threat is presented in the column "1 double skip". The other category "2 single skips" is an attempt to bypass the detection based on single instruction skip scenario, this implies a high simulation time: for instance, the `switch()` scenario requires 580 simulations to exhaustively test single instruction skips, but 36 745 simulations to test double instructions skips.

TABLE VI
FIA: DOUBLE INSTRUCTIONS SKIP ATTACK

| Scenario | 1 double skip | 2 single skips |
|---|---|---|
| | Attack success (%) | Attack success (%) |
| `switch()` | 0 | 0 |
| `FCall()` | 0 | 0 |
| `memset()` | 1.86 | 0.04 |
| `memcpy()` | 0 | 0.09 |
| `memcmp()` | 0 | 0 |
| `VerifyPin()` Password | 0 | 3 |
| `VerifyPin()` Counter | 0 | 0 |
| `ECDSA_verify()` | 0 | 0 |
| `ECDSA_sign()` | 0 | 0 |

## VII. DISCUSSION

### A. Transition function

We implemented three different transition functions: XOR, affine modular and Cyclic Redundancy Check (CRC). To keep the explanations of our method as simple as possible, we focused on practical examples with the XOR transition function. One must keep in mind that the trivial mathematical properties brought by the XOR are not present with more complicated transition's functions such as CRC. The main downside of the XOR transition function is the presence of a neutral element 0, this is an issue because applying 0 to the CoT leaves it unchanged. To avoid this, instead of applying a value expected to be 0 to the CoT, we apply this value plus an arbitrary constant. The XOR transition function is similar to the affine modular transition function in terms of performance and security, while the CRC transition function is significantly slower but has the advantage of not having neutral element in top of other advantages beyond the scope of this paper. The

CRC transition function (or any other complex function) should be chosen over the XOR in case of a hardware functionality justifying this choice.

### B. Fault model

Our protection proposal is designed to detect transient instruction corruptions, depending on the assumptions made on the attacker, it addresses single or multiple faults.

In the case of a single fault injection, we tested exhaustively every single instruction skip (a subcategory of instruction corruptions [25]) with our simulator, testing exhaustively instruction corruption would take too much computation time even for small examples. However, we state that our proposal covers this fault model because the detection relies on loading a protected asset from memory twice, any corrupted or misused loaded value would result in a mismatch between the computed CoT and the execution path.

Considering multiple faults injection, we state that our protection offers high coverage against multiple instruction skips. Due to the CoT being computed from the decision values, a fault corrupting a decision value results in a corrupted CoT value. Then the error propagation properties of the CoT implies that the attack described in section III which corrupts an asset and skips the corresponding verification cannot defeat the protection.

However, a powerful enough attacker can defeat the implemented protection with double fault injection, considering that he is able of corrupting instructions and has enough control over the corruption. For instance in Fig. 5, the attacker needs to corrupt the load instruction over the key size and change the value loaded from 256 to 128 twice. The first time, as the `feed` operation is made and the second when the switch decision is taken. Doing this will produce the exact behavior of both the program and the protection as if 128 was stored in memory. Such a powerful attacker defeats the implementation of our protection because it is based on a double load of the KS value from memory, however, it is possible to implement a stronger version of our protection based on triple (or more) loading of assets from memory, increasing both the security level and the overhead of the protection.

### C. Compiler optimizations

We choose to position our protection on top of the compiling process, which means that compiler's optimizations must be dealt with in order to prevent them from optimizing out our countermeasures.

The first idea that naturally comes is to declare the CoT value as `volatile` [13]. This is necessary in order to avoid the CoT to be optimized out and reassigned later, its value must exist at all time. This is not sufficient because while linking an asset to the CoT, the protection against simple FIA relies on the redundancy of the access to the asset in memory. As an example if we take a decision over an asset A present in memory, we must load its value into a register R and the decision will be made over the value contained in R. If the access is corrupted and the value A' is loaded instead of A, we can argue that both the decision and the `feed/compensate` will be made over the corrupted value and the attack undetected. However, if A is `volatile`, its value will be read both before the decision and before its integration to the CoT, leading to a detection if both readings are not consistent. This means that a solution is to have both the CoT value and the fed values defined as `volatile`. Defining the assets as `volatile` isn't convenient, this is why we used a macro to make `volatile` access to any variable in order to force the compiler to read an asset when we access it. Then, we were able to obtain the same fault coverage with every optimization options present with GCC against FIA.

### D. Method Comparison

TABLE VII
CFI PROTECTIONS

| Method | fault model | double faults coverage | simple fault coverage | Overhead | Insertion |
|---|---|---|---|---|---|
| Proposal [3] | instruction corruption | high | high | low | selective |
| Lalande et al [14] | jump attack | no | high | high | systematic |
| Goloubeva et al (YACCA) [11] | bit-flips | no | high | high | systematic |
| Proy et al [20] | control flow corruption | no | low | low | automated |

TABLE VII presents the overview of the main CFI protections discussed in this paper. Our proposal offers a different approach regarding countermeasures insertion compared to state-of-the-art protections. The automation of countermeasures insertion is viewed as a requirement, however we choose a different approach for two reasons. The first reason is the balance between overhead, fault coverage and insertion, given a high fault coverage, automating the countermeasures insertion means that they are systematically applied to the totality of the code and this results in additional overhead. A selective manual insertion allows us to carefully pick the important parts of the code to protect and to avoid as much as possible unnecessary countermeasures and overhead. The second reason is the fault coverage, defining critical data (assets) and parts of the code is necessary in order to ensure that they are correctly secured, for instance our method allows different protection mechanism for securing a function call depending on its criticality (see section V).

Regarding fault coverage, our protection provide a slightly better single fault coverage, however the main appeal is the double fault coverage. State of the art CFI are not designed to detect double fault attacks, verification are placed into the code which allows to detect single fault attacks. However, there are no mechanisms to protect these verification which allows double faults to successfully bypass these schemes, our proposal fix this issue by meshing program's values with signatures values and ensuring error propagation through the signature value CoT. In top of the proposed CFI detection mechanism based on a CoT, we propose others applications

focused on contextual execution such as conditional execution, data access or contextual masking (see section V).

## VIII. Conclusion

We designed an original software-only CFI protection method applied at the source code level, which allows protecting the assets of a program against instruction(s) skip(s) attacks without binding ourselves to a given compiler or CPU core. By establishing a strong interdependence between the assets of a program and a CoT, we can detect a corrupted decision-making, propagate this information to ensure detection later on or protect the access to an asset or function. The protection not only allows the detection of illegitimate and incorrect execution paths, but also prevents the correct execution of the program in case of a corrupted CoT. The resulting overhead of our protection is highly variable. By carefully choosing the assets of our program to protect we have shown through examples that it can be limited to under one % for a complete example and up to thirty-five time increase for limited but critical sections of code.

For future improvements, we are looking into using our mechanism for different purposes such as anti-tampering and reverse engineering protection. The idea to rely on a CoT as a basis to implement protection functionalities is not fully explored with a CFI approach, and we want to dive deeper into this software protection.

## References


[1] Alessandro Barenghi, Guido Bertoni, Andrea Palomba, and Ruggero Susella. A novel fault attack against ecdsa. In 2011 IEEE International Symposium on Hardware-Oriented Security and Trust, pages 161–166, 2011.

[2] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In ASIACCS '11, 2011.

[3] François Bonnal. Source code of the protection. Available at url=https://github.com/CFICOT/CFICOT.

[4] Thomas Chamelot, Damien Couroussé, and Karine Heydemann. Sci-fi: Control signal, code, and control flow integrity against fault injection attacks. 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 556–559, 2022.

[5] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: a detection tool to defend against return-oriented programming attacks. In ASIACCS '11, 2011.

[6] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh Ha Le, Aude Crohen, and Philippe Choudens. Fissc: A fault injection and simulation secure collection. pages 3–11, 09 2016.

[7] Jean-Max Dutertre, Alexandre Menu, Olivier Potin, Jean-Baptiste Rigaud, and Jean-Luc Danger. Experimental analysis of the electromagnetic instruction skip fault model and consequences for software countermeasures. Microelectronics Reliability, 121, 2021.

[8] Jean-Max Dutertre, Timothé Riom, Olivier Potin, and Jean-Baptiste Rigaud. Experimental analysis of the laser-induced instruction skip fault model. In The 24th Nordic Conference on Secure IT Systems, Nordsec 2019, pages 221–237, Cham, 2019. Springer International Publishing.

[9] Chris Gerlinsky. Breaking code read protection on the nxp lpc-family microcontrollers. RECON, Brussels, Belgium, 2017.

[10] Antoine Gicquel, Damien Hardy, Karine Heydemann, and Erven Rohou. Samva: Static analysis for multi-fault attack paths determination, 03 2023.

[11] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. Improved software-based processor control-flow errors detection technique. Annual Reliability and Maintainability Symposium, 2005. Proceedings., pages 583–589, 2005.

[12] D. Goodin. Record-breaking ddos reportedly delivered by 145k hacked cameras. Ars Technica, volume 28, 2016.

[13] Programming languages, their environments and system software interfaces. Standard, International Organization for Standardization, 1999.

[14] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. Software countermeasures for control flow integrity of smart card c codes. In Computer Security - ESORICS 2014, page 200–218, Berlin, Heidelberg, 2014. Springer-Verlag.

[15] Anh Quynh NGUYEN and Hoang Vu DANG. https://www.unicorn-engine.org/bhusa2015-unicorn.pdf. 2015.

[16] B. Nicolescu, Y. Savaria, and R. Velazco. Sied: software implemented error detection. In Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems, pages 589–596, 2003.

[17] Johannes Obermaier and Stefan Tatschner. Shedding too much light on a microcontroller's firmware protection. In 11th USENIX Workshop on Offensive Technologies (WOOT 17), 2017.

[18] National Institute of Standards and Technology. Digital signature standard (dss). Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 186-4, U.S. Department of Commerce, Washington, D.C., 2013.

[19] N. Oh, P.P. Shirvani, and E.J. McCluskey. Control-flow checking by software signatures. IEEE Transactions on Reliability, 51(1):111–122, 2002.

[20] Julien Proy. Automatic hardening of embedded applications against physical attacks. PhD thesis, 06 2019.

[21] Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. Compiler-assisted loop hardening against fault attacks. ACM Transactions on Architecture and Code Optimization, 14:1–25, 12 2017.

[22] Misoczki Rafael, Heath Constanza, Santes Flavio, Sakkinen Jarkko, Morrison Chris, Bolivar Marti, and Ian King Colin. https://github.com/intel/tinycrypt.

[23] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur., 15:2:1–2:34, 2012.

[24] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. Iot goes nuclear: Creating a zigbee chain reaction. In 2017 IEEE Symposium on Security and Privacy (SP), pages 195–212. IEEE, 2017.

[25] Junichi SAKAMOTO, Daisuke FUJIMOTO, and Tsutomu MATSUMOTO. Laser-induced controllable instruction replacement fault attack. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, E103.A(1):11–20, 2020.

[26] Christoph Spang, Yannick Lavan, Marco Hartmann, Florian Meisel, and Andreas Koch. Dexie - an iot-class hardware monitor for real-time fine-grained control-flow integrity. Journal of Signal Processing Systems, 94:1–14, 07 2022.