2022 CPSSI 4th International Symposium on Real-Time and Embedded Systems and Technologies (RTEST), Tehran, Iran

Software-based Control-Flow Error Detection with Hardware Performance Counters in ARM Processors

Hussien Al-haj Ahmad
Dependable Distributed Embedded Systems (DDEmS) Laboratory
Computer Engineering Department
Ferdowsi University of Mashhad
Mashhad, Iran
hussin.alhajahmad@mail.um.ac.ir

Yasser Sedaghat
Dependable Distributed Embedded Systems (DDEmS) Laboratory
Computer Engineering Department
Ferdowsi University of Mashhad
Mashhad, Iran
y_sedaghat@um.ac.ir

Abstract*—The recent trend in processor manufacturing technologies has significantly increased the susceptibility of safety-critical systems against soft errors in harsh environments. Such errors result in control-flow errors (CFEs) that can disturb systems' execution and cause severe financial, human, or environmental disasters. Therefore, there is a severe need for efficient techniques to detect CFEs and keep the systems fault-tolerant. Although numerous control-flow error detection techniques have been proposed, they impose considerable overheads, making them inappropriate for today's safety-critical systems with restricted resources. Several techniques attempt to insert fewer control-flow checking instructions to reduce overheads. However, they limit fault coverage. This paper proposes a software-based technique for ARM processors to detect CFEs. The technique leverages the Hardware Performance Counters (HPCs), which exist in most modern processors, to count micro-architecture events and generate HPC-based signatures. Based on these signatures that capture the correct control flow of the program, the proposed technique can detect CFEs once the correct control flow is violated. We evaluate the detection capability of the proposed technique by performing many fault injection experiments applied on different benchmark programs. Moreover, we compare the proposed technique with common signature-based CFE detection techniques with respect to fault coverage and imposed overheads. The results demonstrate that the proposed technique on average can achieve ~99% fault coverage which is 23.57% higher than that offered by the employed signature-based techniques. Moreover, the memory overhead imposed by the proposed technique is 4.85% lower, and the performance overhead is ~19% lower than that of the studied signature-based techniques.

Keywords—hardware performance counter, control-flow checking, safety-critical systems, fault injection, software-based error detection.

I. INTRODUCTION

Recently, the popularity of embedded systems has increased dramatically due to the rapid improvements in processors technologies. They are incorporated into various applications ranging from information systems and heavy industries to smart cities, distributed systems, and safety-critical embedded systems. However, employing embedded systems in safety-critical systems that operate in harsh environments has raised severe challenges relating to the reliability offered by such embedded systems [1-3]. To be more specific, the technology advancements have negatively affected the ability of processors to tolerate faults and increased their vulnerability against faults. A safety-critical system is a system in which a fault can disturb the system's execution and cause severe financial, human, or environmental disasters [2]. Therefore, these systems should adhere to strict reliability restrictions before they are put into operation [4-6].

It was discovered around 1970 that unprotected processors are susceptible to faults induced by radiations [7]. High-energy particles can cause random, temporary changes in the operating state of the systems due to affecting one or multiple flip-flops in underlying electronic components . Typically, faults that affect the systems manifest in three main classes depending on their duration [8-10]: permanent faults, intermittent faults, and transient faults. Researchers have pointed out that transient hardware faults, i.e., soft errors, are the main challenge to be overcome to build a fault-tolerant system. Typically, soft errors can affect one flip-flop temporarily, causing Single Event Upsets (SEUs), or more than one flip-flop, causing single-event multi-bit upsets (SEMUs) [11]. SEUs in the memory cells or CPU registers can affect the program execution and cause data-flow errors or control-flow errors (CFEs) [1, 2, 12]. A CFE violates the correct program flow and moves it to an incorrect location within the code space or unallocated memory regions. Due to the deviation in the program's control flow, the operation behavior of the program may be incorrectly modified [1, 13-15]. As a result, CFEs can affect the program execution and cause the program to terminate abnormally or produce undesirable outcomes. Thus, it is of paramount importance to address this type of error for building fault-tolerant systems.

Previous work [1] has stated that CFEs detection requires 2.5x more effort than detecting data errors. Considering reduced instruction set computer (RISC) processors, 33% of all soft errors that affect them are reflected in control-flow errors [15]. Therefore, employing fault tolerance techniques to detect soft error effects, primarily CFEs, is highly important to enhance reliability as ISO 26262 and DO178C recommend using control flow checking (CFC) techniques [4].

Many CFC techniques have been proposed to detect CFEs. These techniques can fall into three categories: hardware-based, software-based, and a combination of both software- and hardware-based techniques as hybrid techniques [8, 16]. Most techniques share the same steps to preserve the program flow correctness [17-24]. Initially, a control flow graph (CFG) of the target program is extracted. Then, additional instructions are inserted at predefined locations (typically at compile-time) to validate the correctness of the branch instruction destination. These instructions leverage special variables called *signatures* to perform control flow checking. CFC techniques differ in how signatures are generated, where to insert the signatures in the code, and the number of assigned signatures.

Instrumenting the program (inserting additional instructions) with many checking instructions and signatures could achieve a high fault coverage [1, 13, 14]. However, this will lead to considerable overheads, making the employed techniques are inapplicable for today's safety-critical systems with restricted resources. Therefore, the main challenge facing CFC techniques is how to provide a high fault coverage (CFE detection ratio) while keeping overheads as low as possible. For tuning the trade-off between fault coverage and overheads, authors in [25] and [26], for example, protect only the susceptible basic blocks to CFEs. Other techniques reduce the checking frequency by deferring the check of the branch correctness [1, 14, 27, 28]. While such methods contribute to reducing overheads, the ability of these methods to detect errors may be negatively affected.

In this paper, we propose an efficient software-based technique to detect CFEs at low overheads without losing significant fault coverage. It leverages the hardware performance counters (HPCs), a common feature in most modern processors, to detect CFEs and keep the system fault-tolerant. In order to detect control-flow errors, we assign, at compile-time, an HPC-based signature to each basic block (a sequence of branch-free instructions) in the control flow graph. The assigned signatures are calculated by counting specific low-level microarchitecture events, e.g., the number of instructions per basic block. During execution, a run-time HPC-signature is calculated and compared with the compile-time one. Any mismatch between signatures is a sign of a CFE.

We evaluate the effectiveness of the proposed technique by conducting many fault injection experiments on diverse benchmark programs that cover different application domains. For comparison purposes, we implement the state-of-the-art random additive signature monitoring (RASM) technique [13] and the Control flow checking by software signature (CFCSS) [22], which is widely referred to in literature because of its high fault coverage. Based on the obtained results, the proposed technique can achieve ~99% fault coverage for CFEs on the selected benchmark and imposes lower overheads.

The remainder of this paper is organized as follows. Section III surveys related work on software-based fault injection techniques. Section IV demonstrates the proposed technique and gives a general background about the hardware performance counter. Section V describes the implementation of the proposed technique. A detailed evaluation of the proposed technique is discussed in Section VI. Finally conclusions are drawn in Section VII.

II. MOTIVATION

Most CFC techniques strive to provide a high fault coverage with acceptable overheads. These techniques operate at different levels of granularity: (1) fine-grain predecessor/successor-assertion, or (2) coarse-grain path-assertion. Despite the high fault coverage provided by the former techniques, they impose significant overheads, as all basic blocks should be instrumented. On the other hand, path-assertion techniques divide a CFG into multiple control-flow paths (a group of basic blocks executed in an uninterrupted sequence) and add check instructions into each path, resulting in fewer inserted check instructions. However, such techniques sacrifice some fault coverage as intra-block CFEs in the same control-flow path cannot be detected.

Another challenge that is quite important and seldom discussed in the literature is how accurate CFG is. The CFG is considered a prerequisite for most CFC techniques. The adopted CFG should be complete and accurate. Lack of precision in the employed CFG may negatively affect the CFC technique, making the results questionable [13]. Therefore, the CFG must be as accurate as possible. In this context, indirect branches represent the major challenge that impedes the construction of a complete and high-precision CFG. Since an indirect jump gives a basic block multiple *undetermined* successors, assigning signatures to validate if the previous (or next) basic block is the correct predecessor (or successor) is not possible.

In this paper, we propose an HPC-based technique to detect CFEs caused by soft errors. Using HPCs allows monitoring the program's execution flow with negligible overheads. The signatures assigned to each basic block in the target program capture the correct control flow of the program. Therefore, based on these HPC-based signatures, the proposed technique can detect CFEs once the correct control flow is violated. Moreover, tuning the overheads can be performed effectively without fault coverage degradation.

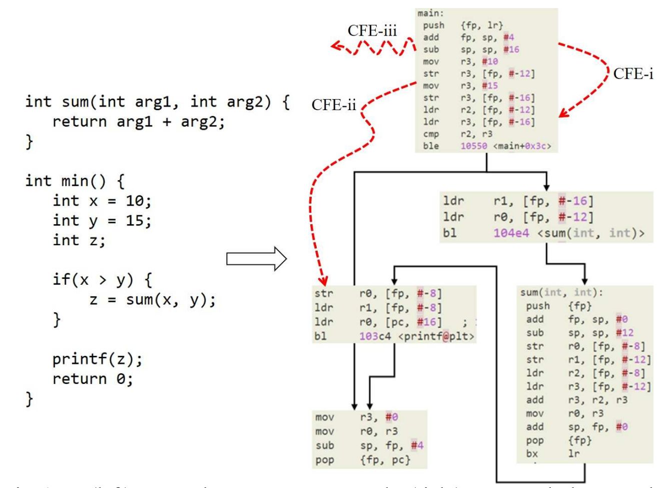
III. RELATED WORK

Before discussing previous CFC techniques, we introduce the following common terminologies widely adopted in most CFC techniques.

Given a program P, it is possible to extract an abstract representation of the execution flow of the program P, i.e., a Control Flow Graph (CFG). Here, we introduce the main terminology.

i. A basic block (BB) is a set of sequential instructions, which are executed one after another from the starting point of the basic block (entry point) to the last one (exit point). The branch instructions are not allowed within the BB body. Such instructions only appear at the end of BB. $V = \{BB_1, BB_2, \ldots BB_n\}$ is a set of all BBs in the source code of P.
ii. Predecessors of BB_i are a set of basic blocks that the program can explore before entering the BB_i.
iii. Successors of BB_i are all possible basic blocks that the program can move to once the BB_i is executed.
iv. A directed edge between BB_i and BB_j represents a legal execution of BB_j after the execution of BB_i in the absence of CFEs. BB_i is a predecessor of BB_j and BB_j is a successor of BB_i. $E = \{e_1, e_2, \ldots, e_n\}$ is a set of directed edges.
v. A Control Flow Graph (CFG) of the program P, CFG_P is an abstract representation of all possible execution paths of the program flow. The control flow graph is usually formed by linking basic blocks using directed edges, i.e., branches. Therefore, $CFG_P = \{V, E\}$ indicates CFG of program P with a set of BBs V and a set of directed edges E.

A basic block may have multiple predecessors and multiple successors, except the first BB in the program, which has no predecessors, and the last BB, which has no successors as it terminates the execution. Fig. 1 shows a sample program source code and its corresponding CFG extracted for the ARM assembly code related to the program. The CFE could appear in



```
int sum(int arg1, int arg2) {
   return arg1 + arg2;
}

int min() {
   int x = 10;
   int y = 15;
   int z;

   if(x > y) {
      z = sum(x, y);
   }

   printf(z);
   return 0;
}
```

```
main:
 push   {fp, lr}
 add    fp, sp, #4
 sub    sp, sp, #16
 mov    r3, #10
 str    r3, [fp, #-12]
 mov    r3, #15
 str    r3, [fp, #-16]
 ldr    r2, [fp, #-12]
 ldr    r3, [fp, #-16]
 cmp    r2, r3
 ble    10550 <main+0x3c>
```

```
ldr    r1, [fp, #-16]
ldr    r0, [fp, #-12]
bl     104e4 <sum(int, int)>
```

```
str    r0, [fp, #-8]
ldr    r1, [fp, #-8]
ldr    r0, [pc, #16]    ;
bl     103c4 <printf@plt>
```

```
sum(int, int):
 push   {fp}
 add    fp, sp, #0
 sub    sp, sp, #12
 str    r0, [fp, #-8]
 str    r1, [fp, #-12]
 ldr    r2, [fp, #-8]
 ldr    r3, [fp, #-12]
 add    r3, r2, r3
 mov    r0, r3
 add    sp, fp, #0
 pop    {fp}
 bx     lr
```

```
mov    r3, #0
mov    r0, r3
sub    sp, fp, #4
pop    {fp, pc}
```

Fig. 1. (left) A sample program source code, (right) A Control Flow Graph (the corresponding ARM assembly code)

different scenarios as follows [13]:

i. Intra-BB CFE: an incorrect jump within the current BB.
ii. Inter-BB CFE: an incorrect jump to another BB.
iii. CFE from a given BB to outside the memory space allocated for the program.

Software-based CFC techniques strive to detect such CFE types. Typically, CFE type iii almost leads to exceptions in the target program, e.g., a segmentation fault or time-out, that are more likely to detect by underlying hardware features. For error types i and ii, these CFEs are more critical in the sense we need sophisticated methods to cover them effectively.

Control Flow Checking by Software Signatures (CFCSS) is a signature-based CFE mitigation technique [22]. It relies on assigning two signatures, at compile-time, to each BB. These signatures are used at run-time to check the correctness of the control flow destination. To verify the control flow, CFCSS calculates a run-time signature and verifies its new value, which should hold the same compile-time value in the case of no CFE has occurred. However, CFCSS is not able to detect CFEs if multiple BBs share multiple destination blocks. Therefore, it suffers from an aliasing problem. This problem has been solved by the CEDA technique [28]. CEDA can increase fault coverage up to 98.9%. Yet Another Control-Flow Checking Using Assertions (YACCA) technique assigns multiple signatures to each BB, at compile-time, in order to detect interblock CFEs [21]. At the exit point of each BB, the run-time signature code is updated and verified to detect violations. Enhanced Control-Flow Checking Using Assertions (ECCA) [23] inserts into each BB four instructions and three compile-time signatures that are: BID (the current BB), and two signatures NEXT1, and NEXT2 that represent the next possible successors, as the authors of ECCA have restricted the possible number of successors to two BBs, e.g., loop and if-then-else statements. At run-time, two variables are calculated from the previously defined signatures to detect CFEs. ECCA is able to detect all inter-BB CFEs, but it is neither able to detect intra-BB CFEs, nor errors that cause an incorrect decision in a conditional branch. Other signature-based techniques which have been widely referred to in literature are RSCFC [19], SEDSR [18], SCFC [17], and SIED [20].

The above mentioned techniques rely on inserting a relatively large number of additional instructions in the target program. CFCSS achieves a high error detection rate as it protects every BB with instructions for updating and validating signatures. However, it imposes significant overheads. In order to decrease this huge overhead, one can economize on the number of instructions added to the program. Recent work [27, 29] inserts fewer instructions to decrease the significant overhead. However, this reduction is achieved at the expense of error detection capability. ACS [27] defers checking the control flow until two BBs are executed. However, it suffers from poor fault coverage. Authors in [25] and [26] proposed to protect only the basic blocks that are destinations of the erroneous branches (CFEs). They termed such basic blocks as susceptible blocks and suggested protecting them only for cutting down overheads.

Path Sensitive Signatures (PaSS) [1] technique can detect both inter- and intra-BB CFEs by implementing a lightweight technique based on signatures. By examining the CFCSS, PaSS stated that comparison instructions (*cmp*) are the main source of the overheads incurred by CFCSS. Thus, PaSS relies on a strategy whereby a group of BBs is selected to be protected in a way that does not negatively affect the fault coverage. PaSS was implemented as a compiler extension pass using the LLVM compiler as it provides a high ability to deal with low-level Intermediate Representation (IR) code.

The hardware performance counters (HPCs) feature in tandem with event ticking pins is employed by Enhanced Committed Instructions Counting (ECIC) to provide error protection for single- and multi-threaded programs run on COTS-based real-time embedded systems [30]. ECIC is evaluated by implementing a prototype on a 32-bit Pentium processor and using software-based fault injection to inject 6000 faults into the target system. Like ECIC, in [31] an HPC-based CFC technique is introduced to detect as many CFEs as possible through a purely software-based technique. However, authors of ECIC and [31] techniques have not considered one important aspect of performance counters, namely how accurate they are.

IV. The Proposed Technique

The technique proposed in this work leverages the hardware performance counter (HPCs) to perform control-flow error detection. The signatures to be used for error detection are extracted by monitoring special microarchitecture events using HPCs. The selected events should be accurate and repeatable. The following section sheds light on the HPCs and illustrates how to extract signatures.

A. Hardware Performace Counters

Hardware performance counters are special on-chip registers that facilitate developers to perform run-time profiling through monitoring many microarchitecture events. Regarding ARM processors, a special unit called Performance Monitoring Unit (PMU) is installed to simplify using HPCs [32, 33]. We can monitor and measure a wide range of events. However, the problem lies in measuring these events simultaneously [34]. As the ARM Cortex-A8 has only four counters, we can simultaneously measure only four events.

It is essential to consider one important aspect of HPCs, namely how accurate they are. Given that modern ARM processors use features such as out-of-order execution and branch prediction, it is expected that such features may negatively affect the accuracy of HPC measurements [33-35]. In other words, not all events measured by HPCs are suitable for CFC because of perturbation in counting events. HPCs run-to-run variation may negatively affect the correctness of the measurements, making it difficult to distinguish between correct behavior and incorrect one caused by CFEs.

However, CFE-induced effects on the program's behavior can be differentiated from the natural perturbations of some HPCs [34]. An illegal jump from BB_i to BB_j, i.e., inter-BB CFE, can result in a distinct difference in the monitored event. Thus, by measuring some precise events, the correct behavior can be differentiated from the wrong, abnormal behavior of the target program. The next section discusses the HPC selection scheme to generate the intended signatures.

B. Accurate HPC-based signatures

The accuracy of HPCs measurements has been experimentally discussed for different architectures [34-38]. Generally, the internal interactions between different operating system modules, hardware interrupts, and the behavior of applications can result in measurement variations. For CFEs detection, the main challenge lies in determining the most accurate microarchitecture events to be monitored. The employed signatures to detect CFEs should be extracted from such events that must be deterministic, high accurate, and do not show run-to-run variation as much as possible. In other words, the more accurate the events, the more confident the results we obtain.

Accordingly, we need to identify the most accurate microarchitecture events in order to generate accurate HPC-based signatures. To this end, we repeatedly profile several programs on ARM architecture simulated using the cycle-accurate Gem5 simulator towards computing signatures. Among a large number of available events, we identify those events that show *insignificant* run-to-run variations in measurements. Based on the profiling information, the most suitable events to be monitored in order to compute signatures are:

i. Instruction executed
ii. Branch instruction executed
iii. Load/Store instructions executed

As the employed signatures are computed from hardware events, the fewer the selected events, the less the overheads required for CFE detection. Therefore, based on our profiling information and previous studies [34, 37], the *instruction executed* event (instruction architecturally executed) is accurate enough to be counted in order to compute accurate signatures. Any deviation from the correct signature assigned to each BB is a sign of CFE. Because the program often contains different BBs with respect to the type and the number of instructions, each BB will be assigned a unique HPC-based compile-time signature $CTsig_i$. The $CTsig_i$ signature is an integer value calculated at compile-time for each $BB \in V$ as follow:

$$CTsig_i = \sum_{k=1}^{n} Insn_k \qquad (1)$$

Therefore, $CTsig_i$ denotes the count of *static* instructions ($Insn_k$) of *ith* basic block that contain n instructions. At the execution, a run-time signature is calculated as follows:

$$RTsig_i = C_i\,(E) \qquad (2)$$

Where C_i denotes the count of the selected event E (*instruction executed*) from the execution of *ith* basic block.

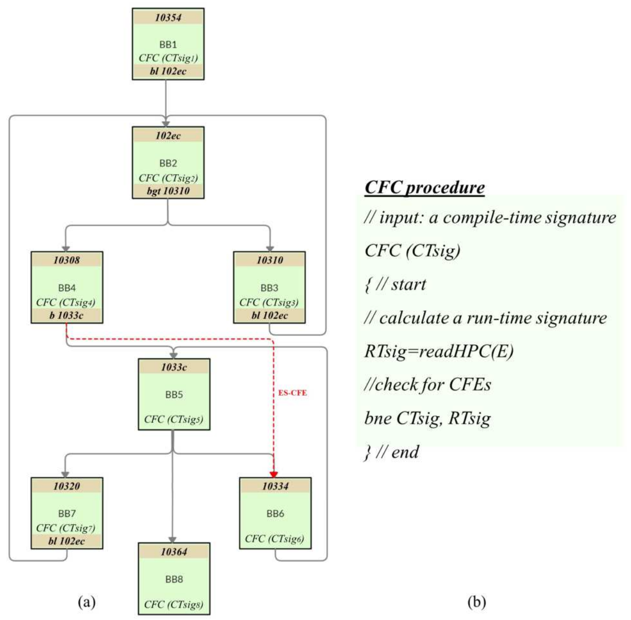
C. Method for control flow checking

Each basic block in the CFG comprises several instructions whose execution triggers a specific number of events. Then, each $BB_i \in V$ is assigned, at compile-time, a signature $CTsig_i$ referring to the correct number of instructions in BB_i. Once a control flow error occurs at run-time, the program is more likely to produce events that their measurements (count) differ from the correct expected count. To be more specific, a perturbation caused by CFEs will affect the count of executed instructions of the program, making the measurements of the corresponding events differ from the compile-time signature. Therefore, a CFE is said to occur if the following condition is met:

$$CTsig_i \neq RTsig_i \quad \forall\ i \in V \qquad (3)$$

This is can be described as follows. Given a BB_i with a known instructions count, a bit-flip error may affect the program's control flow and cause a CFE. As a result, the count of monitored event, i.e., instructions executed, will deviate from the expected one assigned at compile-time, i.e., $CTsig_i$.

Fig. 2.a shows the CFG of *Fibonacci sequence* algorithm with eight basic blocks. Before the execution transfers from BB_1 to its successor BB_2, we should ensure that the instructions of BB_1 have been executed in the correct sequence. Therefore, before executing the BB_2, we check the correctness of the directed edge between BB_1 and BB_2 by performing the following (check instructions):



```
CFC procedure
// input: a compile-time signature
CFC (CTsig)
{ // start
// calculate a run-time signature
RTsig=readHPC(E)
//check for CFEs
bne CTsig, RTsig
} // end
```

Fig. 2 (a) CFG of Fibonacci sequence algorithm, (b) pseudo code of the CFC procedure (CFEs check instructions)

i. Retrieve the measurement of HPC, i.e., $RTsig_l$, that is set to count the number of executed instructions,
ii. Compare the $RTsig_l$ with the reference signature, i.e., $CTsig_l$, assigned at compile-time.

It is evident that any control flow error that causes skipping some instructions can be detected. Applying CFC techniques can increase the size of the original code, which in turn might impair the program's resilience against soft errors. Therefore, instead of adding the check instructions at the end of each BB, we instrument these BBs with *a function call instruction* to the CFC procedure (Fig. 2.b) that contains check instructions, reducing performance and memory overheads.

D. CFE detection capability

A single bit-flip in the address operand of a branch instruction at the exit point of a basic block can lead to the following scenarios: (i) a legal but wrong flow of control and (ii) an illegal flow of control. Regarding the latter, an illegal control flow can be intra-BB CFE, inter-BB CFE, or an illegal branch outside the program's address space. Several techniques have not considered intra-BB CFEs [14, 22, 25-28]. Our proposed technique can detect both intra-BB and inter-BB CFEs, except for one inter-BB CFE, which is a CFE from the end point of BB_i to the start point of a non-successor BB_j (we term this type of control-flow error as End-to-Start CFE or ES-CFE). At the BB_i, we do not check the source of the incoming branch, however, we check the number of executed instructions once the execution reaches the end point of BB_j. Since the instructions of BB_j will be executed correctly from the start point of BB_j till the last one, the ES-CFE (from BB_i to the non-successor BB_j) cannot be detected as the $RTsig_j$ will be equal to $CTsig_j$. Here it is important to note that the employed HPC-based signatures are assigned per each basic block without considering the next or previous basic block. In other words, the proposed technique is not a predecessor/successor CFC technique. Therefore, unlike most of CFE error detection techniques, such as [13, 21-23, 26-28], our proposed technique does not suffer from the problem of indirect branches discussed in Section II. Moreover, to reduce overheads, we can perform control-flow checking at control-path granularity without losing significant fault coverage.

The probability of ES-CFE that occurs due to a single bit-flip in the address operand of branch instructions at the end point of BBs is extremely low. As a result, protecting non-susceptible basic blocks against ES-CFEs can increase the overheads without improving fault tolerance. Therefore, it is important to identify the susceptible basic blocks against ES-CFEs to avoid unnecessary protection.

Given a BB_i that ends with a branch instruction, and a $BB_j \notin$ succ (BB_i), the BB_j is an ES-CFE susceptible basic block if a single bit-flip in the address operand of the branch instruction at the end point of BB_i can move the control flow "*illegally*" to exactly the first instruction (the start point) in the BB_j. The total number of possible ES-CFEs can be calculated as follows:

$$ES\text{-}CFE = \sum_{i=1}^{m} n - BBi_{fan\_out} \qquad (4)$$

Where n denotes the total number of basic blocks, m denotes the total number of basic blocks that end with a branch instructions, and BBi_{fan_out} is the number of successors of BB_i. Considering Fig. 2.a, m is equal to 5, where n is equal to 8. Therefore, there are a total of 34 ES-CFEs. However, not all of these ES-CFEs are a result of a single bit-flip. Since the adopted fault model in this paper is a single bit-flip model, we should discard ES-CFEs that require more than single bit-flip error. To this end, in order to identify only the ES-CFEs caused by a single bit-flip error, we employ an $m \times n$ matrix where their elements are the hamming distances between the address operand of the branch instructions at the end point of the source basic blocks and the address of the first instruction in the non-successor basic blocks (we use the ARM version of *objdump* Linux utility to disassemble the executable file and extract instructions' addresses). We term this matrix as HM. Therefore, if HM (BB_i, BB_j) is equal to 1, this indicates that an ES-CFE from BB_i to BB_j can be occurred due to a single bit-flip, and hence, this ES-CFE should be considered. Table I show HM extracted for the CFG presented in Fig. 2.a.

The susceptible basic blocks against ES-CFE are those having Hamming distance 1. Therefore, among 34 ES-CFEs (Table I), only one ES-CFE can occur, an ES-CFE from BB4 to BB6 as a single bit-flip in the address operand of a branch instruction at the endpoint of BB4 (1033c) can modify this address to (10334), which is the address of the first instruction in BB6. As a result, we can significantly reduce overheads without affecting fault coverage by avoiding protecting basic blocks with HM (BBi, BBj) $\neq$ 1.

Regarding Fig. 2.a, one possible solution to detect ES-CFEs is to treat BB4 and BB5 as one basic block. To this end, the check instructions are inserted only into BB5. Considering the ES-CFE from BB4 to BB6, when the program execution reaches the check instructions at the endpoint of BB6, the number of executed instructions will be different from the $CTsig_6$, as the $RTsig_6$ will be equal to $RTsig_4 + RTsig_6$. Therefore, the CFE can be detected.

V. Implementation

Manual implementation of error detection techniques is impractical because it can be very time-consuming, and hence, it cannot be used to study larger workloads. Therefore, the automated implementation of error detection techniques is essential, as it saves the trouble of implementation.

Most techniques use the CFG to detect control-flow errors. Accordingly, we can implement our proposed technique where it is possible to extract an *accurate* CFG, i.e., a CFG that accurately includes the program instructions to be executed. The availability of source code facilitates constructing the CFG of the corresponding program. Since the code is written in a high-level language, the whole program behavior and logic are evident.

TABLE I. The matrix of hamming distance (Each item indicates the total number of bits that need to be flipped for a ES-CFE to occur between relevant basic blocks)

	BB1	BB2	BB3	BB4	BB5	BB6	BB7	BB8
BB1	5	*	7	5	5	5	4	3
BB2	2	7	*	*	2	2	3	4
BB3	5	*	7	5	5	5	4	3
BB4	3	4	3	3	3	**1**	*	3
BB5	5	*	7	5	5	5	4	3

Asterisks (*) denote legal control flow between the relevant BBs

However, automated implementation at source code cannot guarantee accuracy. The compiler may change the program because of optimizations, e.g., instructions reordering and loop unrolling. Thus CFC technique applied to the source code can be affected during compilation.

A. Automated CFE detection using Compiler extension

The compiler can insert additional instructions during the compilation of the program. The employed compiler in this paper is the GNU Compiler Collection (GCC) [39], a cross-architecture compiler widely used for different programming languages and operating systems. As Fig. 3 shows, the compilation pipeline of GCC comprises three steps, namely front-end, middle-end, and back-end. We are interested in the back-end step where the code is lowered and optimized to emit the assembly code.

Since GCC version 4.5 [39], a compiler extension can be developed as a *plugin* to extend the compiler's functionalities. Thus, we implement the proposed technique at compile-time by developing a GCC-plugin compiler extension that operates at the GCC compiler's Register Transfer Language (RTL) representation code. The RTL representation can be seen as a generic assembly code that can be moved across different architectures. The GCC-plugin appends an RTL-pass and makes it a successor RTL-pass to the *pass-free-cfg*, an RTL-pass executed once before emitting the assembly code. To be more specific, we add the RTL-pass right before emitting assembly code and after ensuring the completion of all optimization passes (Fig. 3). Consequently, none of the check instructions added by the GCC-plugin will be affected or modified, ensuring realistic and accurate implementation.

B. Access hardware performance counters

Hardware performance counters are special-purpose registers built into modern processors. Using these counters for monitoring the selected event to detect CFEs should be performed accurately. Typically, HPCs can be accessed at kernel level or user-level using different interfaces proposed in literature. The *perf_event* interface is a part of the Linux kernel and enables users to access HPCs at the kernel level [35, 36]. Performance Application Programming Interface (PAPI) provides a platform, operating system, and machine, independent access to the hardware performance counters a user-level [40]. PAPI encapsulates the *perf_event* interface's functions and provides a high-level library for flexible access HPCs. However, PAPI's flexibility comes at the price of accuracy and large overheads.

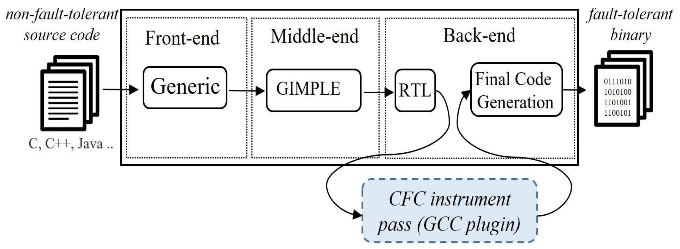


Fig. 3 Compilation pipeline of GCC compiler and the proposed plugin (CFC instrument pass)

To this end, we decide to access HPCs at the kernel-level using the low-level *perf-event* interface. Therefore, we eliminate possible perturbations added by high-level tools. Since the cycle-accurate Gem5 simulator is used to evaluate the proposed technique, it is important to check how Gem5 offers access to HPCs. Gem5 provides a Python script configuration file, namely "ArmPMU.py", which defines raw code for events that gem5 provides for the ARM architecture. Each event has a unique code. These codes differ across the architectures (architecture-dependent) [32].

C. The granularity of checking

The additional instructions inserted to check the program's control flow are considered the main source of overhead. It is essential to consider how frequently these instructions are executed to tune the imposed overheads. Protecting each basic block will reduce the Error Detection Latency (EDL) value, the interval between activating the fault and detecting it, and achieve a high error detection rate, but can impose higher overheads. On the other hand, treating a set of basic blocks as one block (like path-assertion technique) can reduce overheads at the price of the EDL that will be increased, as a CFE cannot be detected until the program reaches the checking instructions. However, a CFG contains several basic blocks with one or few instructions. Therefore, it is possible to treat a set of basic blocks just like a single one.

Authors in [41] proposed the "node expansion" concept to reduce overheads so that only a particular set of basic blocks are protected. The technique proposed in [14] has leveraged the "node expansion" concept to protect only super-node (a single-entry single-exit code region) instead of each BB. However, while such techniques make it possible to reduce overheads, they fail to detect CFEs within each super-node. Therefore, the granularity of node expansion is restricted as its benefits come with a low error detection capability. However, our proposed technique does not suffer from such restrictions, as it can detect any intra-BB CFEs.

VI. EVALUATION AND RESULTS

This section describes the evaluation of the proposed technique. After describing the experimental setup and the selected benchmark programs, we present the Gem5-based fault injection tool that was developed to evaluate the proposed CFC techniques.

A. Target system and benchmark

We evaluate the proposed technique experimentally on ARM architecture using the Gem5 simulator [42]. The Gem5 simulator provides full-system and cycle-accurate (fine-granularity per cycle) simulations. It is publicly available and fully maintained by developers. Moreover, Arm architecture is best supported by Gem5 [42]. We choose the ARM cortex A-53 as it is available in a microarchitecture-level model in Gem5. Moreover, diverse programs were selected to evaluate the error detection capability of the proposed technique. The selected programs are as follows: bubble sort (BS), quick sort (QS), matrix multiplication (MM), and Fibonacci sequence (FS). These programs use our implementation. Two additional programs were selected from MiBench version 1.0 [43], which are bit count (BC) and cyclic redundancy check (CRC).

To implement the proposed technique for the employed programs, we use our compiler extension, which operates at the GCC back-end, and instruments the program at compile-time before emitting the assembly code. We use the GCC cross-compiler with our developed plugin (compiler extension) to compile and implement the proposed technique. The instrumented executable files are then executed on the Gem5 simulator. We implement the state-of-the-art random additive signature monitoring (RASM) [13] and CFCSS [22] techniques to assess the error detection capability of our technique. Several control-flow error detection techniques [1, 13, 15, 28] have selected CFCSS as a baseline to compare and evaluate their techniques. CFCSS and RASM provide high fault coverage as they perform conservative control flow checking and instrument each basic block with multiple instructions.

B. Gem5-based Fault injection

We perform many fault injections on the selected benchmark programs executed on simulated ARM architecture. As the evaluation process is performed on the Gem5, we need to perform fault injection on the Gem5 simulator where the target benchmark programs execute. We take advantage of the built-in GDB remote debugger interface provided by Gem5 to perform remote debugging on the target program executed on Gem5. The adopted debugging strategy is summarized as follows. We run the simulated ARM architecture on Gem5 and execute the target program in debug mode. The program will stop and wait for a connection request from a remote debugger (the host debugger). Usually, the connection is established via TCP/IP protocol. Then, we launch and connect the host debugger with the Gem5's remote interface. At this point, we can control the target program running on Gem5 and perform fault injection.

The fault model we assume is a single bit-flip, as it has been widely adopted in many previous studies [5, 13, 18, 25, 26]. To conduct an effective evaluation, we should inject faults that are more likely to manifest as control-flow errors rather than other error types. A bit-flip error in control registers and operand of branch instructions is more likely to cause a control-flow error. Accordingly, we inject faults in control registers, such as link register (LR), Program counter register (PC), and in the operand of branch instructions to emulate CFEs. We employ the Statistical Fault Injection (SFI) [44] to reduce the number of fault injection experiments while ensuring the results accuracy. Each fault injection campaign involves 1000 fault injection experiments for each benchmark program. Overall, we inject 18000 faults (6 benchmarks × 1000 injections × 3 techniques = 18000 injections). Therefore, we ensure 99% as a confidence level with 1% as the error margin [44].

C. Experimental results

Table II presents the evaluation results of the proposed techniques (Our Tech.) and the employed conservative signature-based CFC techniques. Like previous studies, we define fault coverage as the ratio of detectable errors to total injected faults. We exclude the faults that lead to correct results as they have no effects. Typically, the fault coverage is not suitable to conduct a fair comparison between techniques because it does not consider overheads. Therefore, the overheads in terms of performance overhead (execution time) and memory overhead (code size) should be calculated.

TABLE II. EVALUATION RESULTS

(a) Fault Converge ratio

	BS	QS	MM	FS	BC	CRC	Avg.
Our Tech.[a]	98.1	97.4	99.6	98.2	98.4	100	98.93
CFCSS	82.6	96.8	88.3	82.4	66.2	35.9	75.36
RASM	95.3	96.7	99.2	92.6	95.6	98.3	96.28

(b) Execution time overhead

	BS	QS	MM	FS	BC	CRC	Avg.
Our Tech.	88.2	90.6	75.1	95.2	84.2	24.6	76.31
CFCSS	70.6	145.3	77.3	82.4	132.2	64.2	95.3
RASM	65.4	125.8	64.1	78.7	143.2	36.5	85.61

(c) Memory overhead

	BS	QS	MM	FS	BC	CRC	Avg.
Our Tech.	6.2	21.5	16.1	16.6	9.5	7.2	12.85
CFCSS	8.3	31.6	28.4	22.3	11.0	4.6	17.7
RASM	7.6	19.4	14.6	18.7	12.6	6.3	13.2

a: our proposed technique

Based on the results presented in Table II, on average, the results demonstrate that the proposed technique can achieve ~99% fault coverage which is 23.57% higher than that offered by CFCSS technique (Table II a). Moreover, the memory overhead (Table II c) imposed by the proposed technique is 4.85% lower, and the performance overhead (Table II b) is ~19% lower than that of the CFCSS technique. The overhead, in terms of code size, of the proposed technique is roughly the same as that of RAMS. However, our technique shows less overhead in terms of execution time, as RAMS, like CFCSS, instruments each basic block with multiple instructions. As a result, the proposed technique can be used in the context of embedded systems with restricted resources to detect CFEs, both inter-block, and intra-block errors, at a low level of overheads without losing significant fault coverage.

VII. CONCLUSION

This paper presents a software-based control-flow error detection technique for ARM-based embedded systems. It takes advantage of the hardware performance counter common feature in modern processors to perform control flow error detection and keep systems fault-tolerant. The proposed technique can detect both inter-BB and intra-BB CFEs effectively. We evaluated the effectiveness of the proposed technique on ARM architecture by performing many fault injection experiments on different benchmark programs that cover different application domains. Experimental results showed that the proposed technique achieves a high fault coverage at lower performance and memory overheads. Moreover, the proposed technique is portable across different architectures, as most modern architectures are equipped with HPCs.

References

[1] Z. Zhang, S. Park, and S. Mahlke, "Path Sensitive Signatures for Control Flow Error Detection," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2020, pp. 62-73.

[2] J. Vankeirsbilck, "Advancing Control Flow Error Detection Techniques for Embedded Software using Automated Implementation and Fault Injection," 2020.

[3] A. Tajary, H. R. Zarandi, and N. Bagherzadeh, "IRHT: An SDC detection and recovery architecture based on value locality of instruction binary codes," Microprocessors and Microsystems, vol. 77, p. 103159, 2020.

[4] M. Peña-Fernández, A. Serrano-Cases, A. Lindoso, M. García-Valderas, L. Entrena, A. Martínez-Álvarez, et al., "Dual-Core Lockstep enhanced with

redundant multithread support and control-flow error detection," Microelectronics Reliability, p. 113447, 2019.

[5] Y. Nezzari and C. Bridges, "ACEDR: Automatic Compiler Error Detection and Recovery for COTS CPU and Caches," IEEE Transactions on Reliability, vol. 68, pp. 859-871, 2019.

[6] P. A. Laplante and J. F. DeFranco, "Software engineering of safety-critical systems: Themes from practitioners," IEEE Transactions on Reliability, vol. 66, pp. 825-836, 2017.

[7] M. Hoffmann, F. Schellenberg, and C. Paar, "ARMORY: Fully Automated and Exhaustive Fault Simulation on ARM-M Binaries," IEEE Transactions on Information Forensics and Security, vol. 16, pp. 1058-1073, 2020.

[8] I. Oz and S. Arslan, "A survey on multithreading alternatives for soft error fault tolerance," ACM Computing Surveys (CSUR), vol. 52, pp. 1-38, 2019.

[9] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," ACM Computing Surveys (CSUR), vol. 48, p. 44, 2016.

[10] N. Laranjeiro, J. Agnelo, and J. Bernardino, "A Systematic Review on Software Robustness Assessment," ACM Computing Surveys (CSUR), vol. 54, pp. 1-65, 2021.

[11] H. Cho and K.-W. Kwon, "Modeling Application-Level Soft Error Effects for Single-Event Multi-Bit Upsets," IEEE Access, vol. 7, pp. 133485-133495, 2019.

[12] S. Schuster, P. Ulbrich, I. Stilkerich, C. Dietrich, and W. Schroder-Preikschat, "Demystifying Soft-Error Mitigation by Control-Flow Checking - A New Perspective on its Effectiveness," Acm Transactions on Embedded Computing Systems, vol. 16, p. 19, Oct 2017.

[13] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random Additive Signature Monitoring for Control Flow Error Detection," IEEE Transactions on Reliability, vol. 66, pp. 1178-1192, Dec 2017.

[14] M. Zhang, Z. Gu, H. Li, and N. Zheng, "WCET-Aware Control Flow Checking With Super-Nodes for Resource-Constrained Embedded Systems," IEEE Access, vol. 6, pp. 42394-42406, 2018.

[15] A. Rhisheekesan, R. Jeyapaul, and A. Shrivastava, "Control flow checking or not? (for Soft Errors)," ACM Transactions on Embedded Computing Systems 18, 1, 2019.

[16] A. Kritikakou, R. Psiakis, F. Catthoor, and O. Sentieys, "Binary Tree Classification of Rigid Error Detection and Correction Techniques," ACM Computing Surveys (CSUR), vol. 53, pp. 1-38, 2020.

[17] S. A. Asghari, H. Taheri, H. Pedram, and O. Kaynak, "Software-based control flow checking against transient faults in industrial environments," IEEE Transactions on Industrial Informatics, vol. 10, pp. 481-490, 2013.

[18] S. A. Asghari, A. Abdi, H. Taheri, H. Pedram, and S. Pourmozaffari, "SEDSR: Soft error detection using software redundancy," Journal of Software Engineering and Applications, vol. 5, p. 664, 2012.

[19] A. Li and B. Hong, "Software implemented transient fault detection in space computer," Aerospace science and technology, vol. 11, pp. 245-252, 2007.

[20] 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, 2003, pp. 589-596.

[21] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems, 2003, pp. 581-588.

[22] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," IEEE transactions on Reliability, vol. 51, pp. 111-122, 2002.

[23] Z. Alkhalifa, V. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," IEEE Transactions on Parallel and Distributed Systems, vol. 10, pp. 627-641, 1999.

[24] Y. Sedaghat, S. G. Miremadi, and M. Fazeli, "A software-based error detection technique using encoded signatures," in 21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2006, pp. 389-400.

[25] D. Rodrigues, G. Nazarian, M. Á, L. Carro, and G. Gaydadjiev, "A non-conservative software-based approach for detecting illegal CFEs caused by transient faults," in IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), 2015, pp. 221-226.

[26] G. Nazarian, D. G. Rodrigues, A. Moreira, L. Carro, and G. N. Gaydadjiev, "Bit-Flip Aware Control-Flow Error Detection," in 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2015, pp. 215-221.

[27] D. S. Khudia and S. Mahlke, "Low cost control flow protection using abstract control signatures," in Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems, 2013, pp. 3-12.

[28] R. Vemu and J. A. Abraham, "CEDA: Control-Flow Error Detection Using Assertions," IEEE Transactions on Computers, vol. 60, pp. 1233-1245, 2011.

[29] Z. Zhu, J. Callenes-Sloan, and B. C. Schafer, "Control Flow Checking Optimization Based on Regular Patterns Analysis," in IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC), 2018, pp. 203-212.

[30] A. Rajabzadeh and S. G. Miremadi, "Transient detection in COTS processors using software approach," Microelectronics Reliability, vol. 46, pp. 124-133, 2006.

[31] H. A.-h. Ahmad, Y. Sedaghat, and M. Rezaei, "A performance counter-based control flow checking technique for multi-core processors," in 7th International Conference on Computer and Knowledge Engineering (ICCKE), 2017, pp. 461-465.

[32] A. Holdings, "ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile," ed, 2019.

[33] M. Spisak, "Hardware-Assisted Rootkits: Abusing Performance Counters on the {ARM} and x86 Architectures," in 10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16), 2016.

[34] X. Wang, C. Konstantinou, M. Maniatakos, R. Karri, S. Lee, P. Robison, et al., "Malicious firmware detection with hardware performance counters," IEEE Transactions on Multi-Scale Computing Systems, vol. 2, pp. 160-173, 2016.

[35] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security," in SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security, 2019, p. 0.

[36] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2013, pp. 215-224.

[37] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," in Proceedings of the sixth ACM workshop on Scalable trusted computing, 2011, pp. 71-76.

[38] V. Weaver and J. Dongarra, "Can hardware performance counters produce expected, deterministic results," in Proc. of the 3rd Workshop on Functionality of Hardware Performance Monitoring, 2010.

[39] G. Team, "GCC internal manual for gcc 10.0.0."

[40] J. J. Dongarra, K. S. London, S. V. Moore, P. Mucci, and D. Terpstra, "Using PAPI for Hardware Performance Monitoring on Linux Systems," 2001.

[41] R. Vemu and J. A. Abraham, "Budget-dependent control-flow error detection," in 14th IEEE International On-Line Testing Symposium, 2008, pp. 73-78.

[42] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, et al., "The gem5 simulator: Version 20.0+," arXiv preprint arXiv:2007.03152, 2020.

[43] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538), 2001, pp. 3-14.

[44] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in Design, Automation & Test in Europe Conference & Exhibition, 2009, pp. 502-506.