Mohammadreza Amel Solouki

Simulation Techniques For Rapid Software Development and Validation

Department of Control and Computer Engineering
Politecnico di Torino,Turin, Italy

Abstract

Random hardware failures (RHFs) pose a significant risk, potentially leading to data corruption and Control Flow Errors (CFEs) within embedded systems. To counteract these vulnerabilities, hardening strategies are employed, leveraging either specialized hardware or Software-Implemented Hardware Fault Tolerance (SIHFT) methods. This thesis introduces a novel approach, focusing on the C-level implementation of SIHFT techniques to detect CFEs, alongside the development of simulation techniques for rapid software development and validation. Our proposed approach centers on applying SIHFT methods to detect CFEs within C language-based application code preceding compilation.

However, evaluating these methods presents challenges, notably in terms of the introduced overhead to code size and, critically, real-time application execution. The majority of these methods in the literature are implemented using low-level languages like Assembly. Unfortunately, the development flow for embedded systems applications prefers high-level programming languages like C, aligning with functional safety standards.

Nevertheless, a portion of code persists in Assembly language, where the compiler can automatically insert SIHFT methods, albeit typically limited to highly optimized routines or device drivers. An alternative approach, compiling the application code and then hardening the resultant assembly code, introduces more substantial overhead compared to protecting individual statements in a high-level programming language before compilation.

Therefore, our proposed approach in this thesis centers on applying SIHFT methods to detect CFEs, also recognized as Control Flow Checking (CFC), within C language-based application code preceding compilation. To illustrate this approach, we conducted a comparative analysis of two established software-based control flow error detection methods—Yet Another Control-Flow Checking using Assertions (YACCA) and Random Additive Control Flow Error Detection (RACFED)—implemented in the C programming language. We also assessed the impacts of compiler optimizations.

In the contemporary automotive industry, there is a prevailing trend toward adopting the model-based software design approach. This involves automatically translating executable algorithm models into C or C++ source code. In this context, CFC methods have been integrated into the application behavioral model, and off-the-shelf code generators seamlessly produce the fortified source code for the application. It is worth noting that the majority of SIHFT methods primarily target soft errors, such as single-event upsets typically manifesting as bit flips.

Consequently, the diagnostic metrics commonly provided in the literature,such as error detection latency, fault coverage, and mean time to failure (MTTF), fall short of effectively characterizing these methods when considering the broader spectrum of faults, especially permanent random hardware faults like stuck-at faults. To bridge this gap, our thesis addresses a scenario pertinent to the automotive industry, where the primary concern revolves around permanent random hardware faults, particularly stuck-at faults. Furthermore, we propose a classification scheme aligned with ISO26262 compliance. This classification aims to benefit developers within the automotive sector, where cost and safety considerations often drive the adoption of software-only strategies.

Our results indicated that the diagnostic coverage (DC) for the YACCA method was highest with the `O1` optimization level, showing a marked improvement over the unoptimized version (`O0`). For RACFED, a similar trend was observed, with the detection rate increasing significantly from `O0` to `O1`. However, at higher optimization levels (`O2` and `O3`), while the code size was reduced, some intra-block detection capabilities were lost.

Additionally, our experiments quantified the overheads introduced by these methods. For the TS benchmark, YACCA imposed a text segment size (TSS) overhead ranging from 43.8% at `O0` to -28.8% at `O3`, while RACFED's TSS overhead ranged from 261.23% at `O0` to 100.69% at `O3`. Execution time overheads, measured as the increase in the number of executed instruc-

tions, showed that YACCA imposed a 318.17% overhead at `O0`, decreasing to 90.55% at `O3`. RACFED exhibited a 44.58% overhead at `O0`, which turned into a slight reduction of 5.06% at `O3`. These results underscore the trade-offs between different levels of compiler optimizations and the effectiveness of CFC methods, providing crucial insights for developers optimizing embedded systems for reliability and performance.