CHALMERS

Binary-Level Fault Injection (BLFI) for AUTOSAR-based Systems

Master of Science Thesis
Computer Systems and Networks Programme

NITHILAN MEENAKSHI KARUNAKARAN

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2013

Binary-Level Fault Injection (BLFI) for AUTOSAR-based Systems

NITHILAN MEENAKSHI KARUNAKARAN

Examiner: JOHAN KARLSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 31-772 10 00

Cover:
An illustration of fault injection in AUTOSAR,
3D man with lens © Texelart - Fotolia.com

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

ABSTRACT

Safety is a prime requirement for the automotive industry. Increasing use of complex electrical and electronic systems in vehicles has brought many safety concerns to the industry, in terms of reliability and robustness of these systems. AUTOSAR is an automotive development standard which aims to manage the increasing complexity of E/E systems without affecting their robustness. AUTOSAR facilitates functional safety and promotes a component-based development of automotive software. ISO 26262 is a functional safety standard for road vehicles which provides requirements and processes for developing robust automotive systems. Fault injection and interface testing are robustness assessment methods recommended by ISO 26262. This thesis proposes a binary-level fault injection technique called BLFI, which performs robustness testing on AUTOSAR-based systems. The proposed technique is a wrapping based approach and it can perform black box testing. This technique is evaluated with a proof-of-concept implementation on an AUTOSAR-based LED blinker application.

Keywords: *AUTOSAR, fault injection, binary wrapping, robustness, ISO 26262*

CONTENTS

List of Abbreviations

Abbreviation	Description
ABI	Application Binary Interface
API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
AUTOSAR	AUTomotive Open System Architecture
BFD	Binary File Descriptor
BinUtils	Binary Utilities
BLFI	Binary-Level Fault Injection
BSW	Basic Software
CAN	Control Area Network
COTS	Commercial Off-The-Shelf
DEM	Diagnostic Event Manager
DynInst	Dynamic Instrumentation
E/E	Electrical and Electronic systems
ECU	Electronic Control Unit
ELF	Executable and Linkable Format
FMECA	Failure Mode and Effect and Criticality Analysis
GAS	GNU Assembler
GCC	GNU Compiler Collection
GDB	GNU Debugger
GNU	GNU's Not Unix
GOOFI	Generic Object-Oriented Fault Injection
IDE	Integrated Development Environment
ISO	International Organization for Standardization
LED	Light Emitting Diode
MinGW	Minimalistic GNU for Windows
MSYS	Minimalistic System
OEM	Original Equipment Manufacturer
OS	Operating System
RTE	Run-Time Environment
SW-C	Software Component
SWIFI	Software Implemented Fault Injection
VFB	Virtual Function Bus

Acknowledgements

I would like to express heartfelt gratitude to my examiner and supervisor Prof. Johan Karlsson at Chalmers University of Technology for his tremendous support and expert guidance throughout my thesis work. I am especially grateful to my supervisors Mafijul Islam and Johan Haraldsson at Volvo GTT for their valuable time and technical assistance. I would like to thank Mattias Wallander for making sure I got all the necessary equipment for conducting my study. I would like to offer my special thanks to Fredrik Bernin for clearing all my technical doubts patiently. I wish to acknowledge the help provided by Svante Möller and Mats Olsson. I am thankful to Volvo Group Trucks Technology for making the study possible. Finally, I wish to thank my parents for their continuous support and encouragement.

1. INTRODUCTION

Wheeled vehicles were first seen as early as the 4th millennium BC, from then on vehicles have evolved through ages to automobiles and to present day sophisticated and luxurious cars. With the current trend of integrating more and more computing technology, vehicles are becoming computers on wheels.

One of the reasons for increasing use of E/E (Electrical and Electronic) systems in road vehicles is the performance and capabilities these systems feature. Complex functionalities that were traditionally implemented in a mechanical way are now being implemented using E/E systems. Another reason is the increasing use of autonomous driver assistance systems. One of the main causes for road accidents is human error or negligence. Humans have physical limitations, for example they cannot foresee a vehicle coming at a road crossing, and in many critical situations their reaction to the environment is more emotionally driven and spontaneous, rather than being logical. In this light, computers and E/E systems function efficiently, or at least it is believed so. Active safety systems and automated driving can save lives. Thus, the future of automobiles is moving towards automation and the control is shifting from humans to computers. The underlying assumption that goes unsaid is that these systems are completely reliable and robust.

There is a natural increase in the complexity of these E/E systems with more complicated and specific demands. Safety is a non-negotiable requirement in the automotive industry [1]. Increasing usage of complex E/E systems has increased the probability of failures and poses a threat to the safety requirements. In order to manage this increasing complexity without affecting the quality and reliability factors, major vehicle manufacturers and suppliers came up with a common development standard called AUTOSAR (AUTomotive Open System Architecture) [2]. The main goals of this standard include improving scalability of solutions, increasing the use of Commercial Off The Shelf (COTS) components, and to facilitate functional safety.

Functional safety is defined as “absence of unreasonable risk due to hazards caused by malfunctioning behavior of E/E systems” [3]. Avoidance of faults or else the detection and handling of faults are the fundamentals of functional safety. ISO 26262 is a functional safety standard for road vehicles that addresses possible hazards caused by malfunctioning behavior of E/E systems. ISO 26262 standard recommends using fault injection as a tool for testing and verifying E/E systems in automobiles [3]. Fault injection is defined as a dependability validation technique in which faults are deliberately introduced into the target system, under controlled conditions and the system behavior is studied in the presence of faults [4].

There are many fault injection tools in existence in the research world which perform fault injection on traditional computer systems. Unfortunately, there are not many commercial tools available in the market. The need for more commercial tools has risen with the introduction of AUTOSAR and ISO 26262 standards. Developing a customizable and efficient fault injection mechanism for AUTOSAR based systems is not straight forward, because of the inherent complexity and abstractions in these systems [5]. An attempt to develop a guidance framework for implementing fault injection in AUTOSAR based systems in a customizable and efficient way is made in [5]. Lanigan and Fuhrman [6] discuss a technique that injects fault into an AUTOSAR based system using a CANoe simulation environment. As

AUTOSAR standard is being adopted by many automotive industries, there is a need for adopting existing fault injection techniques to cover AUTOSAR.

In this master's thesis, existing fault injection techniques are studied with intent to adapt them or derive ideas from them, for injecting faults in AUTOSAR. Based on the research a binary level fault injection technique is proposed which introduces faults in AUTOSAR-based systems, using a wrapping approach. The proposed fault injection method is validated by implementing a proof of concept fault injection on an AUTOSAR test application.

The thesis report is organized into nine chapters. Chapter 1 gives an introduction and sets the background for the thesis. Chapter 2 presents the research methodology adopted in this thesis. Chapter 3 gives a broad overview about dependability, AUTOSAR, and other technical concepts used in this thesis. Different fault injection techniques are described in Chapter 4, and Chapter 5 is dedicated to binary level fault injection tools that were explored in detail. Implementation of the proposed technique, its evaluation, and analysis are presented in Chapters 6 and 7. Chapter 8 presents discussion and future work, while final conclusions are made in Chapter 9.

2. RESEARCH METHODOLOGY

This master's thesis is carried out in line with the methodology proposed for research in information systems by Peffers et al [7]. Peffers divides the research process into six activities as shown in Figure 1, they are problem analysis & identification, derivation of objectives of the solution (literature review), design & development, implementation, evaluation, and communication.

Problem analysis and identification is done in collaboration with the stakeholders and supervisors. Objectives of the solution are derived from detailed study of relevant literature, with the goal of identifying techniques and solutions that are applicable to the current context. The literature for study is selected using the snowball technique. From the knowledge gained in previous activity, a prototype tool is designed and implemented which addresses the research problems. Regular feedback from the supervisors and examiner forms the evaluation activity. The thesis report and oral presentation by the author constitutes the communication activity. Development, implementation and evaluation activities are iterated until the research problems are addressed suitably.

This thesis work is undertaken as part of the BeSafe project, at Electrical and Embedded Systems department, Volvo Group Trucks Technology. BeSafe is a research project funded by Vinnova which aims to identify benchmark targets, and develop methodology for performing benchmark testing in the automotive industry. The research partners are, Volvo AB, Volvo Cars Corporation, QRTECH, Scania AB, Chalmers University of Technology, and SP Technical Research Institute of Sweden [8].

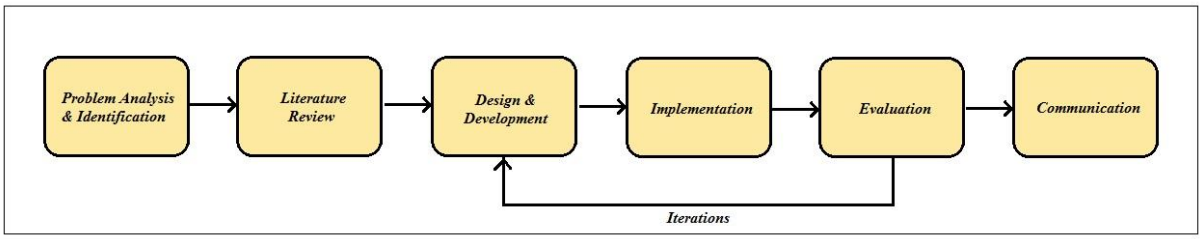


Figure 1. Research Methodology [7]

3. BASIC CONCEPTS

This chapter gives a technical overview of different concepts used in this thesis. Section 3.1 discusses about dependability, its attributes and the pathology of failure. Section 3.2 gives an introduction to AUTOSAR and layered architecture. A broad picture of the ISO 26262 standards for functional safety is given in section 3.3. Section 3.4 gives an introduction to ELF (Executable and Linkable Format) file format, for better understanding of the proposed fault injection method since it is based on extracting information from the ELF file.

3.1 Dependability

Before getting into the details of dependability, we would like to clarify the notion of a 'system'. A system in this thesis can mean different things, from a simple software component to a complex system with several ECUs (Electronic Control Units) in a vehicle.

Avizienis et al. [9] presents the taxonomy of dependability in an organized fashion. Dependability is defined as "the ability of a system to avoid service failures that are more frequent and more severe than is acceptable". The stress is on trust, a system is said to be dependable if the user can trust the services provided by the system. Avizienis presents the following attributes to measure the dependability of a system:

- **Availability**: readiness for correct service.
- **Reliability**: continuity of correct service.
- **Safety**: absence of catastrophic consequences on the user(s) and the environment.
- **Integrity**: absence of improper system alterations.
- **Maintainability**: ability to undergo modifications and repairs.
- **Confidentiality**: absence of unauthorized disclosure of information

Faults, errors, and failures are the threats to dependability. According to Avizienis et al. a fault is: "the cause, either adjudged or hypothesized, behind an error", where error is "the part of total state of a system that may lead to its subsequent failure". A failure or a service failure is "an event that occurs when the service delivered by the system deviates from its correct service". Faults can occur internally or externally in a system, the presence of an internal fault or vulnerability is necessary to enable an external fault to induce an error which may result in subsequent failure(s).

The life-cycle of a system can be divided into two phases, namely the development phase and the use phase. Faults can be introduced into the system in both these phases, due to many reasons like physical environment, users of the system, intruders, and development tools. A fault that can produce an error is called **active fault**, otherwise it is said to be **dormant**. An active fault is either an external fault freshly introduced, or a dormant fault that got activated because of certain conditions. The latter is defined as **fault activation**. An error in one part of a system can lead to errors in other parts, this process is termed as **error propagation**. When the error propagates to the system boundary and becomes visible to the environment then it is becomes a **failure**. The ways in which a system can fail are called the ***failure modes*** of the system. The genesis and manifestation of faults, errors, and failures and the relationship between them is called "pathology of failure" and its causality flow is shown in figure 2.

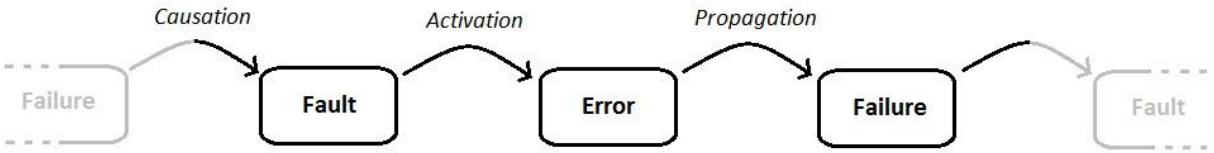


Figure 2. Error propagation

Dependability is achieved in a four-fold way, namely fault prevention, fault tolerance, fault removal, and fault forecasting. These are called "the means of dependability"

- ***Fault prevention*** means to prevent the fault from occurring or getting introduced.
- ***Fault tolerance*** means to avoid service failures in the presence of faults.
- ***Fault removal*** is reducing the severity of faults or their frequency of occurrence.
- ***Fault forecasting*** is the estimation of the number of faults, their future incidence and likely consequences.

3.2 Overview of AUTOSAR

AUTOSAR (AUTomotive Open System Architecture) is an open industrial standard for automotive E/E architectures developed by major automotive OEMs and suppliers [2]. The main incentives for this standardization are to manage the increasing complexity of E/E systems, quality and reliability, to improve flexibility and scalability, and to enable identification of design errors in the early phases of development [2].

The fundamental design concept of AUTOSAR is to separate the application from the software infrastructure [2]. This is achieved by organizing the architecture as layered and modular, where each layer abstracts the underlying layer and provides a standardized set of services (interfaces) to the layer above.

3.2.1 AUTOSAR Layered Architecture

The different layers of the AUTOSAR architecture and how they communicate is described in figure 3. The basic layers of the architecture are Application layer, RTE (Run Time Environment), BSW (Basic Software), and the underlying ECU hardware.

The application layer is the topmost layer and it consists of SW-Cs (Software Component) that encapsulates an application or parts of its functionality. Any communication between SW-Cs is routed through the underlying RTE layer which provides a standardized interface for the SW-Cs [11].

RTE implements the VFB (Virtual Function Bus) functionality on a specific ECU. VFB is a key communication abstraction that hides the underlying layers from the SW-Cs [2]. This makes the SW-Cs independent of the underlying hardware and they can be relocated to other ECUs during system configuration. Since the communication requirements of SW-Cs are dependent on the application, the RTE is generated individually for each ECU satisfying these requirements [11].

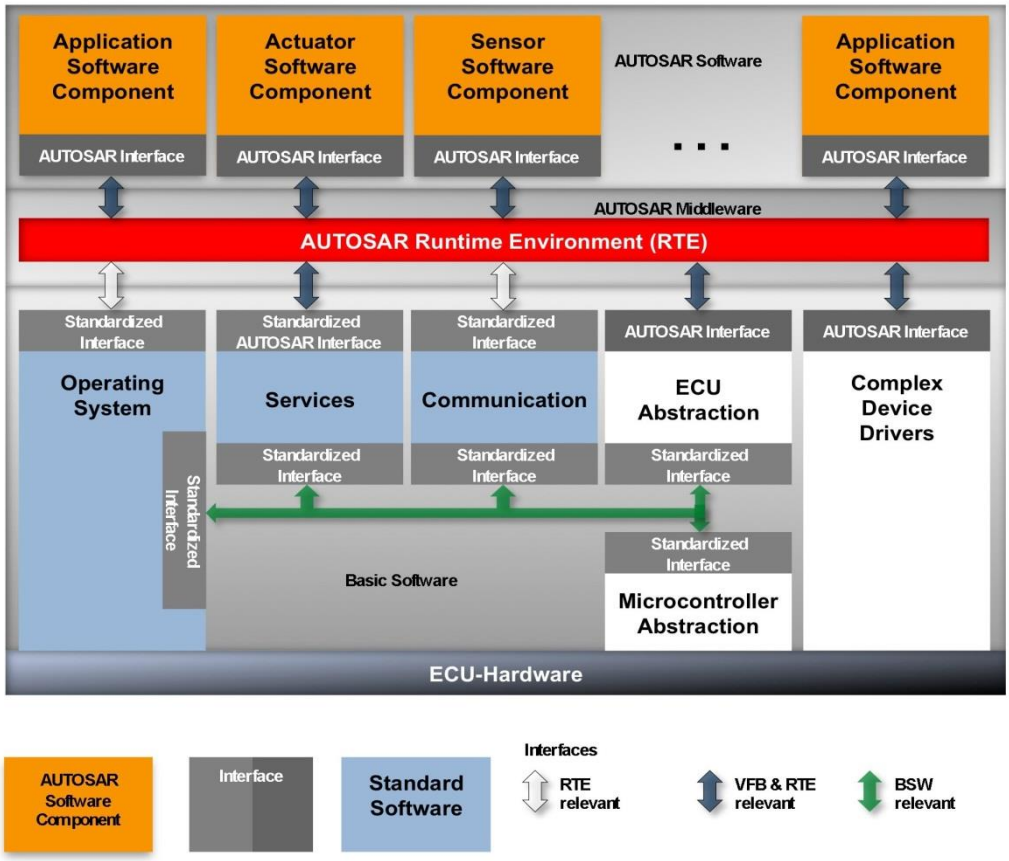


Figure 3. AUTOSAR layered architecture [2]

BSW layer lies below the RTE layer and it provides necessary services to the application running above. It contains both standardized and ECU specific components [2]. The earlier include operating system, I/O and memory management services to the SW-Cs, and microcontroller abstraction that provides access to the underlying hardware. ECU abstraction and Complex drivers come under the latter [11]. ECU abstraction provides software interface to higher-level software layers and complex drivers allow direct access to ECU hardware for resource critical applications [2].

An AUTOSAR-based application consists of interconnected software components which communicate through the standardized interfaces provided by AUTOSAR [5].

3.3 ISO 26262 – Functional Safety

ISO 26262 is a functional safety standard which specifies requirements and processes that need to be adopted to develop safety critical E/E systems in road vehicles. It clearly states that safety requirements are not confined to the end product, but safety features form an integral part of each development phase. Functional safety is influenced by all the development phases like requirements specification, design, implementation, verification and validation. The standard specifies a set of activities that needs to be performed in each sub phase. It also describes how to perform these activities, and how to document the findings [3].

The standard was published in November 2011 and it is intended to be applied for passenger cars. Trucks and buses are expected to be included in the next revision of the standard in 2014. The standard has ten parts, among them parts 4, 5, and 6 specify necessary activities and processes for product development at system level, hardware level, and software level respectively. The standard strongly recommends fault injection and interface testing as methods for assessing the robustness of E/E systems [3].

ISO 26262 provides a risk-based approach for classifying safety functions, these are called risk classes or ASILs (Automotive Safety Integrity Levels). ASIL is a measure of the risk reduction achieved by a safety function in a product. A software component that implements safety functionality is assigned an ASIL. There are four ASIL classes namely, A, B, C, and D, where A stands for lowest amount of risk reduction, and D stands for highest amount of risk reduction. In general a software component with that requires a higher ASIL will have rigid requirements and recommendations at each development stage [3].

3.4 ELF File Format

ELF is a standard file format for storing object files, executable files, shared libraries etc. It was developed by Unix system laboratories as a part of the ABI (Application Binary Interface) [12]. The file format provides two different views of the object file, one used for program linking is called the Linking view, and the other used for program execution is called the Execution view. The organisation of an ELF file and its different views can be seen in figure 4. A general ELF file has a header section, followed by file data. The file data consists of program header table, section header table, sections, and segments [12].

The ELF header provides information about the organisation of the rest of the file (e.g. number of program and section headers), it tells how the file should be interpreted, whether it is an object file, executable or a shared library. Program header table contains information about the segments in the file, it is present in files that are used to create a executable program [13]. Segments are specific to executable view they contain information for runtime execution of the program [12].

Every section in the ELF file has an entry in the section header table, the entry contains details like the section name, size etc. Sections are specific to the linking view, and they contain information like instructions, data, symbol table, and relocation information that are used during linking.

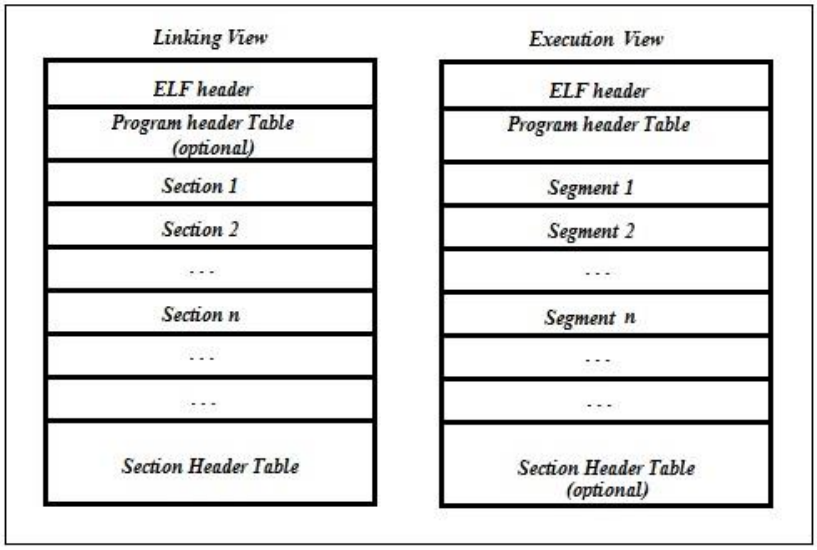


Figure 4. ELF Format [13]

4. OVERVIEW OF FAULT INJECTION

This chapter introduces fault injection and presents various fault injection techniques in existence. We strive to give a picture of the state-of-the-art in fault injection techniques for AUTOSAR-based systems.

4.1 Introduction to Fault Injection

Initial research on fault injection can be traced back to early 1970s [14] and many fault injection techniques have emerged since then. Fault injection is defined as "the dependability validation technique that is based on the realization of the controlled experiments where the observation of the system behavior in presence of faults, is explicitly induced by the deliberate introduction of faults into the system" [4]. Fault injection techniques can be used for fault removal and fault forecasting which play a critical role in improving the robustness of a system as described in Chapter 3 [4].

Depending on the basis of classification, fault injection techniques can be classified in many different ways. On the basis of nature of the target system, fault injection techniques can be classified into software based fault injection and hardware based fault injection [4]. In software fault injection, the software running on the target system is modified in-order to change the system state and then study the effects. In hardware fault injection, the fault is introduced into the target system through specially crafted test hardware (for e.g. flipping a bit in a system register) [4]. There is another class of Hybrid fault injection techniques that combines the versatility of software based methods and the accuracy of the hardware based methods.

From another viewpoint, fault injection techniques can be grouped into execution based and simulation based [4]. The former method involves deploying of the original system and introducing faults in it, these techniques can be useful for evaluating the robustness of a target system which is in final design stage. In the latter, fault injection is performed on a simulation of the actual system, this technique has a natural drawback that it cannot properly capture all the system properties [15].

Based on the intrusiveness, fault injection techniques can be classified into invasive and non-invasive methods [4]. Invasive techniques are those that leave behind a foot print during testing or they take a toll on the performance of the system, whereas non-invasive methods have minimum or no effect on the target system.

4.2 Fault Injection Environment

A general fault injection environment consists of the system under test plus a fault injector, fault library, workload generator, workload library, controller, monitor, data collector, and data analyzer [15]. The different components and how they interact can be seen in figure 5.

Fault injector is the component which injects fault into the target system, fault values that are injected into the system are selected from the **fault library**. **Workload generator** generates work for the target system, the workload is stored in the **workload library**. **Controller** controls the fault injection experiment and **monitor** tracks the execution of the fault injection

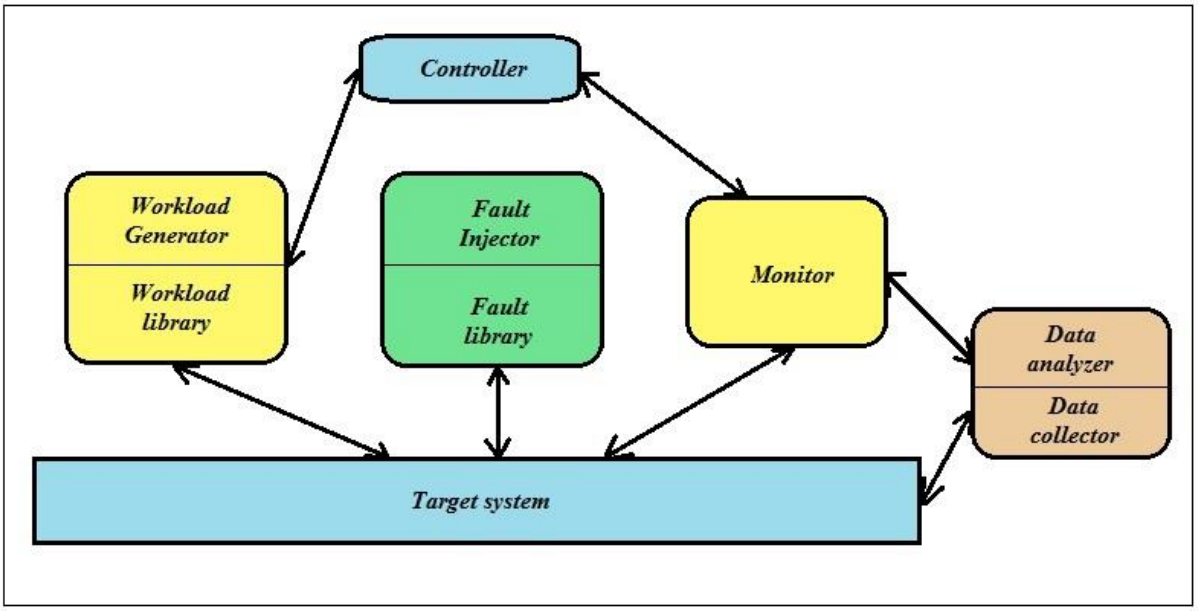


Figure 5. Fault Injection Environment [15]

experiment. **Data collector** is used to collect trace data for analysis during the experiment. **Data analyzer** analyzes the data collected by data collector.

The fault injection tool proposed in this thesis consists of a fault injector and fault library. The other components like monitor, data collector, and data analyzer are not implemented yet.

4.3 Fault Model

The most critical step for a fault injection experiment is selection of an appropriate fault model. The fault model is a subset of the real fault space and all the faults that are introduced into the system during fault injection experiments are extracted from the fault model selected. Johansson et al classify fault models into data-type based, random or fuzzing based, and bit-flip fault models [35]. Data-type based fault model is constructed based on the data types used in programming the target system. It provides a list of interesting fault values for each of the data types. On the other hand, fuzzing fault model is based on randomness and it provides random fault values, it doesnot consider the data type. Fault injection can also be brought about by flipping bits in the registers of the target system, the fault model for this kind of fault injection is called bit-flip fault model.

Testing the target system with all the possible fault values from the fault model is expensive and inefficient. Intelligent selection of test values from the fault model is required. Input test values can be categorized into valid inputs and invalid inputs. N. Kropp [34] shows that it is necessary to test both these values during robustness testing.

In this thesis we explore two types of input testing, one based on data types and other using random strings as inputs (fuzzing). Data type based interface testing is used in the Ballista project [17]. Implementation of different data types vary with different programming languages and compilers used. Each data type has valid and invalid range of values that needs to be identified for each system under test. Experience shows that boundary values between valid and invalid ranges constitute interesting test values [34]. For example, for testing Integer

inputs, a good test case would be to use {0, 1, -1, MaxInt, MinInt}. For random testing, random strings are generated and are typecasted to the respective data types to be tested.

4.4 Software Implemented Fault Injection Techniques

In recent years, software faults are the major cause of system outages [4]. A detailed study of the field data to understand the nature of software faults is made in [18]. It is observed that software faults are mostly human born and thus are extremely difficult to emulate, especially with the current trend of COTS (Commercial Off-The-Shelf) development. A natural consequence of this is more interest in developing software-implemented fault injection tools [15].
Based on the type of errors introduced, SWIFI (Software Implemented Fault Injection) techniques attempted till now can be broadly classified into three categories, namely data errors, interface errors and code changes [16].

Data errors: In this approach fault injection is achieved by data corruption, this is an indirect approach since data corruption is an effect of the fault and not the fault itself (e.g. corrupting the contents of a register).

Interface errors: The error is injected at the inter module interfaces in order to test the robustness of the target module (e.g. passing wrong values as input to functions inside a module). This way of injecting faults is called *Interface testing,* and an implementation of this technique is found in [17], called as ballista tool.

Code changes: In this method the code of the target system is changed (e.g., changing the destination address of an assignment operation). But this kind of fault injection is not easy because a clear knowledge of where to introduce the code change is required and the syntactic correctness of the modified code has to be ensured (e.g. having two branches in a SWITCH construct).

Two different ways of performing fault injection are identified in line with the different software access levels [5].They are fault injection at source code level and fault injection at binary level. Instrumentation of source code is possible when the source code of the system is accessible (called as white box access), but providing access to the source code is purely the choice of the supplier and in many cases they don't (black box access). Fault injection at binary level is the way to go in the latter case. In some cases, suppliers provide partial access (e.g. access to the header files and libraries), and fault injection can be attempted at this level. It is also called as grey box access.

There are some well-established and tested SWIFI tools in literature. FIAT is an automated real-time fault injection tool that validates and characterizes the dependability of a system [19]. DOCTOR is a modular software fault injection environment that evaluates the dependability of a system under generated synthetic workloads, and collect performance and dependability data [20]. Xception is a commercial fault injection tool used for testing in space agencies, which is based on target processor's debugging and performance monitoring features [14].

A software fault injection tool G-SWFIT is proposed in [18] which injects fault directly into the binary code by using fault emulation operators. Fault emulation operators are derived

from a library that contains assembly code pattern and corresponding mutations required to emulate the software fault for that code pattern. The tool is applicable to standard architectures like ARM and MIPS [21]. GOOFI is a fault injection tool that performs test port based fault injection by introducing transient bit flips [22].

4.4.1 Binary Level Fault Injection

There are many binary level fault injecting tools in literature. G-SWFIT is a binary level SWIFI technique in which fault injection is brought about by changing the object code. The object code is subjected to pattern analysis to identify code patterns referred as fault operators, where valid faults can be injected. It falls under the category of software implemented fault injection techniques brought about by code changing [18]. This technique is dependent on the hardware platform and compiler of the target system, because programming constructs can be translated differently, depending on the hardware of the system.

Binary wrapping is another technique which performs instrumentation by manipulating the symbol table entries. The routine to be traced or instrumented is renamed by making necessary modifications in the symbol table, a new routine is created which poses as the original routine by assuming its name. This new routine can be used to perform necessary instrumentation and then the original routine is called to do the original operation. Limitation of binary wrapping is that code insertions can be made only at beginning or end of the target routine without affecting the actual implementation of the routine. A possible improvisation would be to hijack unresolved external calls by manipulating the *external reference* section of the symbol table allowing us to possibly change the actual functionality of the routine [23].

Etch is a binary rewriting tool for WIN32/86 binaries used for measurement and optimization purposes [24]. A binary extraction technique based on control graphs is proposed in [25], which involves control flow discovery and function identification. Dyninst [26] is a binary modification tool used in many research environments, which performs runtime code patching.

4.4.2 SWIFI in Modern Automotive Systems

The need to develop robust automotive systems has increased with the introduction of automotive standards like AUTOSAR and ISO 26262. As mentioned in section 3.3, ISO 26262 recommends fault injection for verifying robustness. Performing efficient fault injection in AUTOSAR-based systems is challenging, due to the fact that AUTOSAR systems are built using model-based development and automated code generation, and they can have different software access levels.

Lu et al. propose a fault injection technique for AUTOSAR based systems which performs fault injection by using the software hooks that are provided by the AUTOSAR OS [27]. Software hooks are entry points or empty routines that are provided for debugging purposes, these hooks can be placed intelligently inorder to trace the execution and control flow. The downside of this approach is that it is limited to the OS level and demands white box access for inserting hooks[5]. The same authors also propose another technique which employs the software hooks provided by the RTE of the AUTOSAR-based system to trace the calls between SW-Cs and the RTE at the interface level [28].

A Fault injection approach at the Basic software level is proposed in [6], they use CANoe simulation environment, which is a commercial tool providing simulation and evaluation environment for automotive applications. They also rely on software hooks to achieve fault injection. A guidance framework for developing fault injection tools for different access levels in AUTOSAR is proposed in [5]. Pintard et al [36] tries to integrate fault injection techniques throughout the development process as recommended by ISO 26262. They show the similarity of Failure Mode and Effect and Criticality Analysis (FMECA) and fault injection at model level.

5. METHOD SELECTION

This chapter describes the two methods selected from literature namely, GNU BinUtils and Dyninst in order to implement fault injection in AUTOSAR-based systems. The prototype tool proposed in this thesis is based on GNU BinUtils. We also present why we didn't select Dyninst.

5.1 GNU BinUtils

GNU is a free operating system, which provides a software collection of applications, libraries, and developer tools. It is typically used with Linux kernel [29]. GNU BinUtils is a collection of programming tools that are used for creating and modifying programs at binary level [30]. Most of them depend on BFD (Binary File Descriptor) library and *opcodes* library to perform low-level operations. They are generally used with compilers like GCC and GDB [10].

The most widely used tools are 'ld' which is GNU linker, and 'as' which is GNU assembler also known as GAS [30]. There are other tools like nm, objdump, objcopy, readelf etc. which are predominantly used in binary world. GNU Binutils come with several options which help in tailoring these tools for specific scenarios.

nm lists the symbols from the object file (binary file), for each symbol nm shows the symbol value, symbol type and the symbol name. nm can be used with different options to extract different information from the object file (e.g. *-l* option displays the line numbers of the symbols) [31].

objdump displays information about the object file, the information displayed is controlled by the options (e.g. *-t* displays the symbol table entries of the file) [31]. If supplied with an archive of object files, objdump displays information about each of the object file.

objcopy utility is used to copy one binary file into another. It uses the BFD library to read and write the object files. The exact behavior of objcopy is controlled by the various command-line options. It can copy the contents of an object file and write it in another format different from the source format [31].

readelf displays information about object files in ELF format. It is similar to objdump but it displays more detailed information and it is independent of the BFD library. It also comes with various command-line options (e.g. *-wi* option displays information from the debug section) [31].
Minimalistic GNU for Windows is called MinGW, it is a minimalistic GNU development environment ported to windows environment. It provides GNU Binutils and GCC support on windows for developing native MS-Windows applications. It generally comes with a command line interpreter called MSYS which can be used as an alternative to Microsoft's cmd interpreter [31].

These GNU tools can be used to extract and modify ELF files, they can be tailored together intelligently to inject faults in AUTOSAR-based systems. The prototype tool developed in

this thesis relies on GNU BinUtils, the details of the implementation is discussed in next chapter.

5.2 DynInst

Dyninst is a post-compiler program instrumenting API developed by the paradyn project [37]. It supports run-time code patching, it provides a C++ class library that acts as a machine independent interface which can be used to create tools and applications that can perform binary code patching. There are two ways in which binary instrumentation can be realized. One, additional code can be augmented to the existing program to measure performance or to do input testing. Two, the control flow of the program can be changed by mutating the subroutines and function calls [26].

The API is based on two basic abstractions, namely points and snippets. "A point is a location in a program where additional code can be inserted". "A snippet is a representation of a bit of executable code to be inserted into a program at a point," [26]. To illustrate with example, to count the number of times a procedure or function is called, the first instruction in the function body can be used as a point, and a snippet can be used to implement a counter which performs the actual counting. Two additional abstractions, threads and images are included in the API in-order to support multi process instrumenting. "A thread refers to a thread of execution" [26], it can either be a normal process or a lightweight thread. "Image refers to the static representation of a program on disk," [26]. Each thread is associated with exactly one image.

Figure 6 shows how the API is implemented [26]. Mutator is the program which uses the Dyninst interface and abstractions like points and snippets to instrument the target application program (Mutatee). The API provides callback functionality to notify the mutator about interesting events that occur in the application program, for example application process termination.

Dyninst could be used to perform fault injection in an AUTOSAR-based system. An AUTOSAR-based system basically consists of an ECU hardware and a software ELF file which runs on the ECU. The target ELF can be instrumented with snippets and the mutator program can be used to control and monitor the execution of the target system. The main problem that could arise is that the ELF runs on a target embedded platform and the mutator program can either be run from PC or directly on the target platform. Running the mutator on the target platform can add as additional overhead and may result in memory related problems. On the other hand controlling the execution from a PC can add timing overhead. Running the mutator from PC or directly on the target platform is a design related issue.

5.3 Issues with DynInst

Dyninst can perform run-time fault injection in applications. But the tool is not fully mature for Windows environment. When we tried to build the tool on Windows using Eclipse IDE and GCC compiler, we ran into several technical problems. The most prominent problem was "*missing libdwarf.h library"*, we got reply from the Dyninst support team that the possible reasons for the error could be incompatible name mangling between the test program and Dyninst libraries, linker misinterpreting how to handle an exports lib/code in DLL situation, or incompatible ABIs between GCC C++11x and MSVC C++11x.

We successfully built Dyninst using Microsoft Visual Studio 2010, and we were able to extract function prototype information from .exe and .dll files compiled with the Visual C++ compiler. But the tool doesn't recognize .lib and .elf formats, and files compiled using other compilers like GCC. Another drawback of Dyninst is that it cannot instrument binary files on Windows environment, it only can attach to a running process or create a process and attach to it. So, the executable needs to be up and running for the tool to perform fault injection. In our case, we needed a tool which can implement fault injection on ELF files running on target hardware. Getting Dyninst to do this will require building the tool for the target architecture (PowerPC in our case) on a Windows environment. This was considered out of the scope for this thesis and we decided to proceed with GNU BinUtils. Dyninst can be a very attractive area of future work for fault injection in AUTOSAR. There are other APIs like SymtabAPI, ParseAPI developed by the paradyn project [37] which can also be explored for possibilities of using them for fault injection.

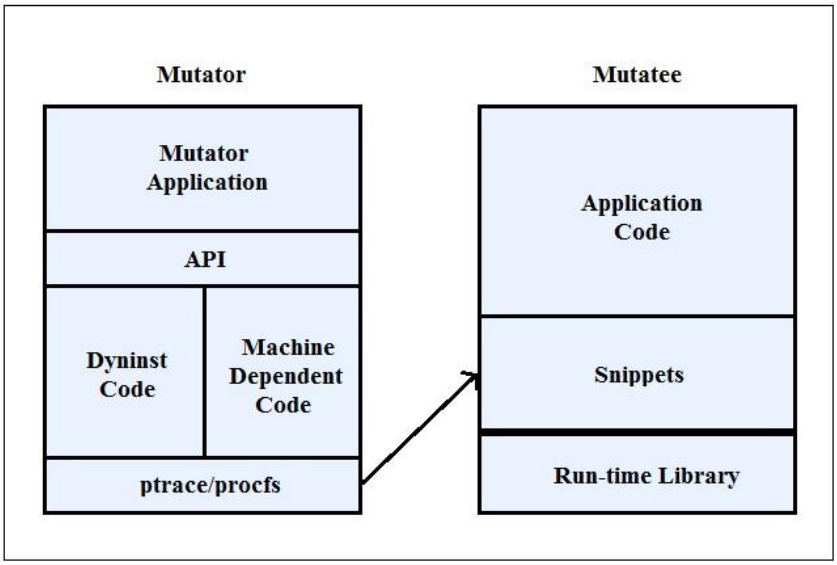


Figure 6. Dyninst API [26]

6. PROTOTYPE IMPLEMENTATION

This chapter describes the development environment, and the steps involved in implementing the Binary-Level Fault Injection (BLFI) tool, henceforth we would refer to the tool as BLFI. The actual fault injection is brought about by wrapping a function call and passing wrong values as input to the original function (this is called as interface testing). BLFI relies on the "*wrap*" feature provided by the GNU linker to perform the actual wrapping.

6.1 GNU *wrap* Option

GNU linker *ld* provides a --*wrap* option for redirecting function calls [33]. We will understand the usage of wrap with a simple hello world example. Let us consider a helloworld.C program as shown in figure 7.

For instance, the *"print_number"* function can be wrapped using --*wrap* option by following steps. We will write a wrapper module called hellowrap.C which contains the wrapper function named *"__wrap_print_number"*. The naming convention used for wrapping is specified by the GNU linker and it has to be followed carefully [33]. Figure 8 shows a simple code snippet illustrating the implementation.

Now the wrapping can be achieved by compiling the wrapper module (hellowrap.C) together with the target program (helloworld.C) and supplying --*wrap=print_number* to the GCC compiler. When the compiled program is executed any call to *"print_number"* function will be redirected to *"wrap_print_number"* function. And in order to call the original *"print_number"* function *"real_print_number"*should be used. In this way we can modify the value of parameters passed to *"print_number"* function and thus perform interface testing.

```
#include "stdio.h"
void print_number(int);
void main()
{
      print_number(5);
}

void print_number(int i)
{
      printf ("the entered number is %d", i);
}
```

Figure 7. Code for helloworld.C

```
__wrap_print_number (int a)
{
    printf ("wrapped successfully,
passing argument %d", a);
    printf ("Calling original
function");
    __real_print_number (a);
}
```

Figure 8. Wrap Module

This --wrap feature of GCC compiler is the core of the proposed BLFI. The prototype tool is implemented in three major steps namely, function prototype extraction, wrapper generation, and wrapping.

6.2 Function Prototype Extraction

The AUTOSAR test system is generated from a library and linker script using the Arctic Studio IDE. The library contains object files for different modules of the AUTOSAR test system, the linker script contains information about how to map the object files into the executable. The generated AUTOSAR test system is an ELF file. As discussed in Chapter 3 the resulting ELF file contains an ELF header followed by sections of file data. All the above steps can be skipped if we directly test on a target ELF file.

Inside the ELF there is a section named *.debug* which contains information about symbols used in the ELF file. These symbols can be function names, function parameters, and variables. The debug section is in DWARF debugging format which is a file format used by compilers to support source level debugging [32], and it has a nested tree structure. It contains information about function names, function return type, parameter names, and their types. GNU BinUtils provides a tool called *readelf* to dump the contents of the debug section [31].

Figure 9 shows the snapshot of a portion from the debug section. The attribute name *Dw_Tag_subprogram* stands for the function name (in this case Rte_DigitalOutput_Set), the following lines give other details about the function, like code address ranges, whether the function is external or not, the function's return type, the file containing the source code etc. [32]. *Dw_Tag_formal_parametemer* contains details about the function parameters, parameter type, location etc. [32].

```
<1><b44>: Abbrev Number: 16 (DW_TAG_subprogram)
   <b45>   DW_AT_sibling     : <0xba8>
   <b49>   DW_AT_external    : 1
   <b4a>   DW_AT_name        : Rte_DigitalOutput_Set
   <b60>   DW_AT_decl_file   : 1
   <b61>   DW_AT_decl_line   : 61
   <b62>   DW_AT_prototyped  : 1
   <b63>   DW_AT_type        : <0x301>
   <b67>   DW_AT_low_pc      : 0x1dc
   <b6b>   DW_AT_high_pc     : 0x248
   <b6f>   DW_AT_frame_base  : 0x81    (location list)
<2><b73>: Abbrev Number: 17 (DW_TAG_formal_parameter)
   <b74>   DW_AT_name        : SignalId
   <b7d>   DW_AT_decl_file   : 1
   <b7e>   DW_AT_decl_line   : 60
   <b7f>   DW_AT_type        : <0x8f8>
   <b83>   DW_AT_location    : 2 byte block: 91 70
<2><b86>: Abbrev Number: 17 (DW_TAG_formal_parameter)
   <b87>   DW_AT_name        : value
   <b8d>   DW_AT_decl_file   : 1
   <b8e>   DW_AT_decl_line   : 61
   <b8f>   DW_AT_type        : <0xba8>
   <b93>   DW_AT_location    : 2 byte block: 91 72
```

Figure 9. Snapshot of debug section

A parser is used to extract this information from the debug section, the output is a text file containing the prototype of all the functions, in a human readable format.

6.3 Wrapper Generation & Wrapping

The function to be wrapped is selected from the function prototype file (generated in previous step), the test values to be fed to the function are generated by the fault library, which can be configured to be either data-type based or fuzzing based. A generator takes the selected function name and the test values as input and generates the actual wrapper (a C source file). Then the powerpc-eabi compiler (which is a cross compiler for PowerPC architecture) is used to link this wrapper with the test AUTOSAR system and produce the wrapped ELF file. The steps involved can be seen in Figure 10, where the fault injection module performs function prototype extraction and wrapper generation, and ELF generator module compiles the wrapper together with the library to generate the wrapped ELF. Table 1 summarizes the whole tool chain.

6.4 Development Environment

The BLFI is developed on a PC running Microsoft Windows 7 operating system. The function prototype extractor and wrapper generator are implemented in C# using Microsoft Visual Studio 2010 development environment. The actual wrapper is generated in C language, and the wrapping is implemented for a PowerPC target, using powerpc-eabi development tools, which contains the GCC, GDB, and GNU binary utilities which include the GNU assembler and linker. The AUTOSAR test system is built using Arctic Studio, which is an Eclipse based IDE provided by Arccore.

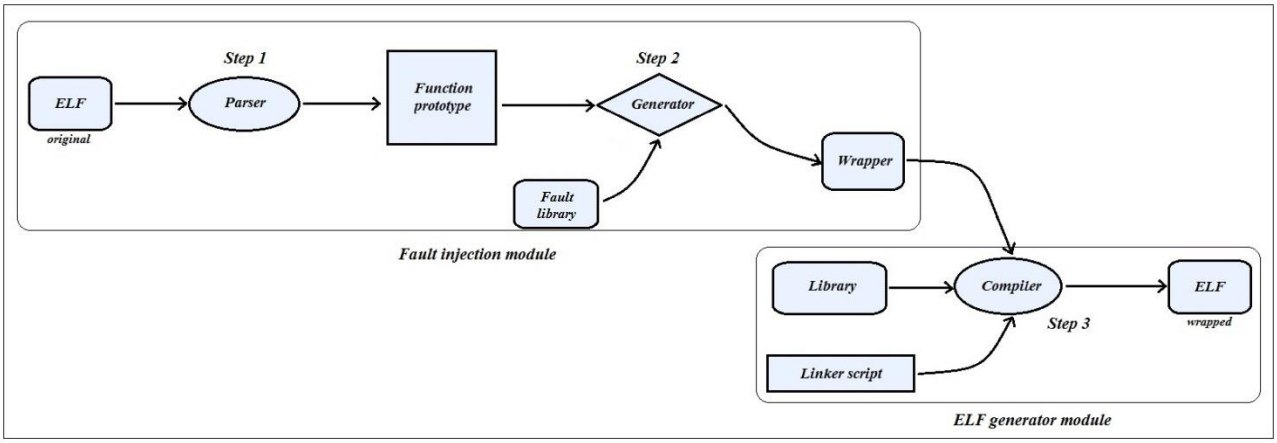


Figure 10. BLFI Tool Chain

Steps	Description
Step 1	Extracting function prototype from target ELF file. If target ELF is not available it is generated from target library using linker script as shown in the ELF generator module.
Step 2	Generating wrapper using the input from step 1 and test values from the fault library.
Step 3	Compiling the wrapper module together with the target library to produce the wrapped ELF.

Table 1. Steps Involved

7. EVALUATION

A proof-of-concept implementation of the BLFI is performed on an AUTOSAR-based test application. In this chapter we describe the details of the experiment and also analyze the efficiency of the tool.

7.1 Experimental Setup

The proposed method was tested on a Led blinking application. The Led blinking application is a full-fledged AUTOSAR system built using Arctic Studio IDE supplied by Arccore. It consists of two software components, namely LedActuatorSWC and SignalMirrorSWC. The LED actuator sends signals periodically to the LED on the ECU board, it keeps the LED blinking continuously. The SignalMirror component reads messages from the CAN bus and send the message back to the CAN bus as a new message with a new message ID. The system is compiled using powerpc-eabi cross GCC compiler for PowerPC target architecture, the end result is an ELF file which can be flashed onto an ECU board. The system layout can be seen in figure 11. The test system is run on an ECU board with MPC5567 series freescale microprocessor. P&E micro debugger is used to control the application, like setting breakpoints in the program or resetting the ECU board.

Different possible failure modes for this system are, LED blinking without any effect (error is masked), LED continuous On, LED continuous Off, LED blinks with different pattern.

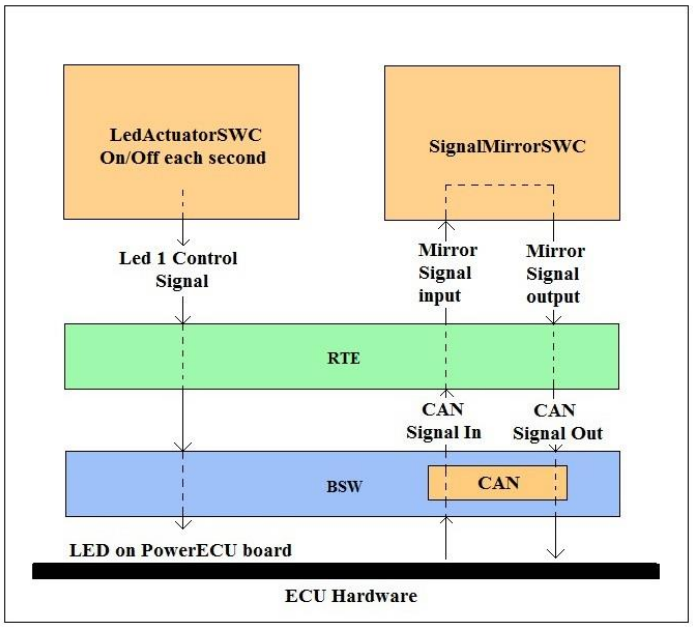


Figure 11. Target System

7.2 Fault Injection

To show the efficacy of the prototype tool, a series of fault injection experiments were performed on the blinker module (LedActuatorSWC) of the test application. The executable (ELF) of the original test system is generated using Arctic studio, the parser extracts from the ELF, function prototypes of all the functions in the test system. Then the function to be wrapped is selected by the user, in this case the *blinker* function is selected for wrapping. Then the wrapper generator generates the *wrap_blinker.C* module. In this experiment the wrapper module is tailored intelligently in order to produce a different blink pattern on wrapping, which is a proof for successful wrapping and fault injection.

Figure 12(a) shows the call pattern before wrapping, figure 12(b) shows after wrapping. Figure 12(c) shows how the blinking pattern changes because of wrapping. The high in the square wave signifies LED On and low signifies LED Off. Before wrapping, whenever the *"blinker"* function is called the original blinker function gets executed. Whereas after wrapping, any call to *"blinker"* is redirected to *"wrap_blinker"* function, which performs neccesary changes to the input values and calls the original function inside its implementation as *"real_blinker"*. In this way fault injection is achieved.

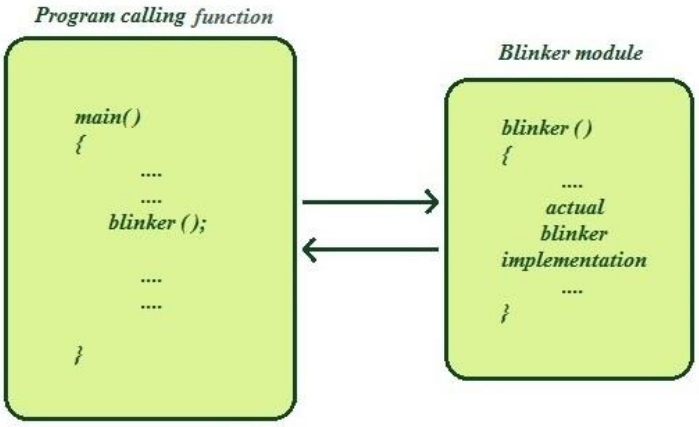
7.3 Analysis

A series of test runs were performed on the LED actuator module of the target system, passing different values as input. The values were selected based on the data type of the input (Unsigned Integer in this case) as discussed in section 4.3. The module was also tested with random inputs (fuzzing). The results from the tests are presented in Table 2. As we can observe, in more than 50% of the cases the LED is just lit without blinking. Even though this is a simple analysis, it successfully demonstrates the functionality of the tool.

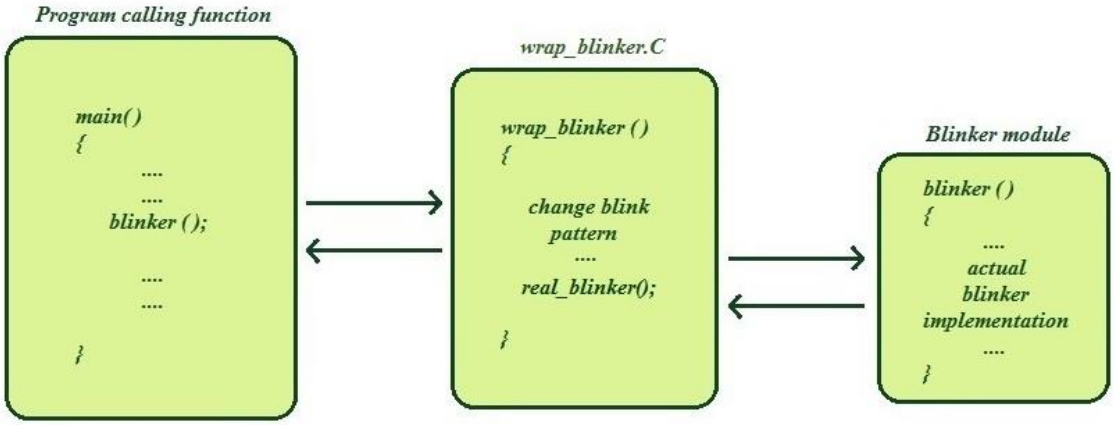
BLFI does not modify the source code of the target application, it is compiled together as an extra module while compiling the application. This method is intrusive in the sense that it changes the size of the resulting ELF file, but in terms of overhead it doesn't affect the performance of the system significantly. It also supports simultaneous wrapping of multiple functions, so it is possible to see the effect of the fault introduced immediately when the function returns, or we can study how the fault propagates and transforms as the program executes, by wrapping another module at different layer of the application.

Value	**Failure mode**
1	Led Off
0	Led On (no blinking)
-1	Led Off until -5, but then Led On for all negative values below -5
UINT_MAX	Led Off
UINT_MAX+1	Led On (no blinking)
UINT_MAX-1	Led On (no blinking)
Random Input	Led On (no blinking)

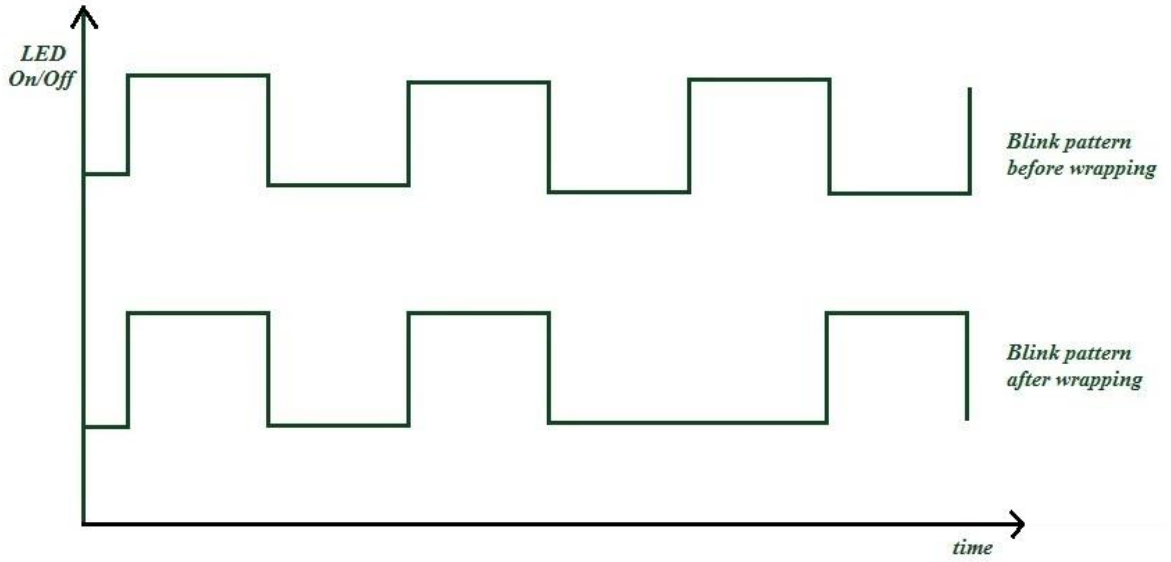
Table 2. Test Results



(a) Original call Pattern



(b) Call pattern After Wrapping



(c) Blink Pattern

Figure 12. Wrapping Call Pattern & Effect

8. DISCUSSION AND FUTURE WORK

In this chapter we discuss the current capabilities and limitations of BLFI, and also suggest some methods for improving and expanding the tool.

BLFI is dependent on the compiler for wrapping (GCC *wrap* option). At present the tool cannot be used on AUTOSAR systems compiled using other compilers. For example, the Wind River compiler which is used at Volvo GTT does not provide a wrap feature. This restricts the tool to be applicable only on AUTOSAR systems compiled using GCC compiler. A positive remark at this point is that Wind River suppliers are taking steps to incorporate the wrap feature in their future compiler releases. A possible solution for overcoming the compiler dependency is to replace the target function name entry in the symbol table of the ELF file with the function name of the wrapper. In this way any call to the target function can be redirected to the wrapper module. This technique is discussed in [23].

The parser that is implemented as a part this tool extracts function prototype from the debug section of the target ELF. This extraction is based on pattern matching and regular expressions provided by the C# language, but it was observed that the format of the debug section varies a little based on the compiler used for building the AUTOSAR system. So in order to use the tool with other compilers, the parser needs to be updated depending on the format of the debug section for that specific compiler. Since the tool is built using GNU Binary utilities there are some low level dependencies, like libraries that need to be installed before we can use the tool.

A fault injection tool is complete when it has all the basic components as described in section 4.2. In that sense, we have implemented only the first two components, namely the fault injector and the fault library, rest of the components are yet to be implemented. A possible approach to implement the data collector component can be to wrap the functions that are output interfaces of the target system, and write to a CAN bus, the data that gets passed to the function. In figure 13 we can see how we can track the values that get passed as input to the original *"print_number"* function and also since the original function returns to the wrapper, it is also possible to capture the return values of the original function. Then the fault injection effect can be studied by analyzing the collected data.

```
__wrap_print_number (int a)
{
    //input parameters of the function
    // can be written to a CAN bus
    __real_print_number (a);
    //if the original function returns any value,
    //that can be captured here
}
```

Figure 13. Capturing Trace Data Inside Wrapper

AUTOSAR architecture provides facility for storing state information in the form of freeze frames, through the Diagnostic Event Manager (DEM) [38]. DEM stores events detected by software components, the diagnostic events are stored in a dedicated memory location called event memory. So another solution for implementing data collector could be to configure the DEM in the AUTOSAR test system.

In the implemented tool, we perform two types of fault injection, one based on the data-type of the function parameters and other by passing random values to the functions (fuzzing). In data-type based testing the scope was limited only to wrapping functions that have parameters of basic types (e.g. integer, char). More data types may be introduced in the AUTOSAR standard in the future. Pointers and strings are interesting candidates for data-type based testing, for example knowing how a null pointer is handled by the system would be very useful. BLFI needs to be extended to handle more complex data types like enumerations and structures. We have implemented the two main components of the fault injection environment as discussed in section 4.2, namely the fault injector and the fault library. Other components like monitor, controller, data collector, and data analyzer are left for future work.

BLFI extracts necessary information from the ELF of the AUTOSAR test system, it does not require any source code access, so it can be used for black box testing of components supplied by different vendors. Another important feature of BLFI is that it can wrap functions from any layer of the AUTOSAR layered architecture. It can inject faults by wrapping functions on the application software layer and simultaneously track the impact by wrapping functions in the basic software layer or the RTE. BLFI can also be used to inject timing errors in the target system. The flow of the program can be blocked briefly inside the wrap function, before or after the wrapped function is called.

9. CONCLUSION

Increasing usage of E/E systems in vehicles has increased the complexity but reduced the reliability of safety critical automotive systems. Automotive standards like AUTOSAR and ISO 26262 specify requirements and methods to build robust systems. Automotive industries are adapting to these standards and they need a means to verify the robustness of E/E systems. Fault injection can be a viable technique for assessing the robustness of automotive systems. We presented in this thesis a binary-level fault injection tool (BLFI) that can perform robustness testing on AUTOSAR-based systems. BLFI is a wrapper based approach which relies on the *wrap* option provided by the GCC compiler. As it requires no source code access it can be used to perform robustness testing in black box software systems. This can assist automotive industries to compare and select software supplied by third party vendors. An important feature of the tool is that it can inject faults into any layer of the AUTOSAR layered architecture and it can be extended to perform fault tracing in different layers. BLFI was evaluated on a LED blinker application which is an AUTOSAR-based test system. Faults were injected into the application layer and basic software layer of the test system, and a preliminary study on the results was performed. We use data-type based and fuzzing based fault injection in this tool. These fault injection models are used in other researches and are found to be good in exposing faults. The wrapping based BLFI can be a promising tool for verifying robustness in AUTOSAR environment.

REFERENCES

[1] AUTOSAR, "AUTOSAR Basics," 2013. [Online]. Available: http://autosar.org/index.php [Accessed 26 April 2013].

[2] AUTOSAR, "AUTOSAR Technical Overview v2.2.2," AUTOSAR, Munich, 2011a.

[3] ISO, "International standard ISO 26262 – Road vehicles – Functional safety," ISO, Geneva, 2011.

[4] Y. Yangyang and B. W. Johnson, "Fault Injection Techniques – A Perspective on the State of Research," in *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, A. Benso, P. Prinetto (Eds.), Frontiers in Electronic Testing, Vol. 23, pp. 7-39, 2004.

[5] T. Piper, S. Winter et al, "Instrumenting AUTOSAR for dependability assessment: A guidance framework," *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2012.

[6] P. E. Lanigan and T. E. Fuhrman, "Experiences with a CANoe-based Fault Injection Framework for AUTOSAR," *In Proceedings, IEEE/IFIP International Conference on Dependable Systems and Networks*, vol. IEEE Computer Society, p. 569—574, 2010.

[7] K. Peffers, T. Tuunanen, M. A. Rothenberger and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45-78, 2007.

[8] C. T. University, "BeSafe - benchmarking of functional safety,"2011. [Online]. Available: http://www.chalmers.se/safer/EN/projects/pre-crash-safety/associated-projects/besafe-benchmarking. [Accessed 13 May 2013].

[9] A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11-33, 2004.

[10] GNU Binutils, Wikipedia, The Free Encyclopedia, [Online]. Available: http://en.wikipedia.org/wiki/GNU_Binutils. [Accessed 15 April 2013].

[11] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, K. Nishikawa, J.-L. Mate, and T. Scharnhorst, "AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E Architectures," *Convergence International Congress & Exposition On Transportation Electronics*, 2004, pp. 325–332.

[12] Executable and Linkable Format, Wikipedia, The Free Encyclopedia, [Online]. Available: http://en.wikipedia.org/wiki/ Executable_and_Linkable_Format. [Accessed 07 May 2013].

[13] ELF, "Executable and Linkable Format," Tool Interface Standards (TIS), Portable Formats Specification, Version 1.1, Chapter 1, 2.

[14] J.M. Voas and G. McGraw, "Software Fault Injection – Inoculating Programs Against Errors", New York: John Wiley & Sons, Inc, pp. 5-6, 1998.

[15] M.-C. Hsueh, T. K. Tsai, R. K. Iyer, "Fault Injection Techniques and Tools," *IEEE Computer,* pp. 75-82, April 1997.

[16] D. Cotroneo, R. Barbosa et al, "Experimental Analysis of Binary-Level Software Fault Injection in Complex Software," *Dependable Computing Conference (EDCC),* 9th European, pp. 162 -172, 2012.

[17] P. Koopman, K. DeVale, and J. DeVale, "Interface robustness testing: Experiences and lessons learned from the ballista project," *Dependability Benchmarking for Computer Systems,* p. 201, 2008.

[18] J. Durães and H. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. on Software Engineering,* vol. 32, no. 11, pp. 849-867, 2006.

[19] J. H. Barton, E. W. Czeck, Z. Z. Segall et al, "Fault injection experiments using FIAT," *IEEE Transactions on Computers,* vol. 39, no. 4, pp. 575-582, 1990.

[20] S. Han, H. Rosenberg, and K. Shin, "DOCTOR: An integrated software fault injection environment," Technical Report, University of Michigan, 1993.

[21] A. Jin, J.-h. Jiang, "Fault Injection Scheme for Embedded Systems at Machine Code Level and Verification," *15th IEEE Pacific Rim International Symposium on Dependable Computing,* 2009.

[22] J. Aidemark, J. Vinter, P. Folkesson, J. Karlsson, "GOOFI: Generic Object-Oriented Fault Injection Tool," *International Conference on Dependable Systems and Networks,* 2001.

[23] J. Cargille, B. P. Miller, "Binary wrapping: A Technique for Instrumenting Object Code," *ACM SIGPLAN Notices,* 27(6):17 18, June 1992.

[24] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen., "Instrumentation and optimization of win32/Intel Executables using Etch," *Proceedings of the USENIX Windows NT Workshop,* pages 1-7, August 1997.

[25] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code*," SIGARCH Comput. Archit. News* 33(5), 2005.

[26] B. R. Buck and J. Hollingsworth, "An API for Runtime Code Patching," *Journal of High Performance Computing Applications,* 14(4), pp.317-329, 2000.

[27] C. Lu, J.-C. Fabre, and M.-O. Killijian, "An approach for improving Fault-Tolerance in Automotive Modular Embedded Software," *Proc. of the 17th International Conference on Real-Time and Network Systems (RTNS),* 2009.

[28] C. Lu, J.-C. Fabre, and M.-O. Killijian, "Robustness of modular multilayered software in the automotive domain: a wrapping-based approach," *Proc. of the 14th IEEE International Conference on Emerging Technologies & Factory Automation*, pp. 1102–1109, 2009.

[29] GNU Operating System, [Online]. Available: http://www.gnu.org/home.html. [Accessed 15 April 2013].

[30] GNU Binutils, [Online]. Available: http://www.gnu.org/software/binutils. [Accessed 15 April 2013].

[31] MinGW, [Online].Available: http://www.mingw.org/ [Accessed 15 April 2013].

[32] DWARF, "Dwarf Debugging Information Format," UNIX International, Programming Languages SIG, Revision: 2.0.0, 1993.

[33] Linux man pages, [Online]. Available: http://linux.die.net/man/. [Accessed 09 May 2013].

[34] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *Fault Tolerant Computing. Digest of Papers. Twenty-Eighth Annual International Symposium*, pp. 230-239, IEEE, 1998.

[35] A. Johansson, N. Suri, and B. Murphy, "On the selection of error model(s) for OS robustness evaluation," in *Dependable Systems and Networks. DSN'07. 37th Annual IEEE/IFIP International Conference*, pp. 502–511, IEEE, 2007.

[36] L. Pintard, J-C Fabre, K. Kanoun, M. Leeman, and M. Roy, "Fault Injection in the Automotive Standard ISO 26262: An Initial Approach," in *14th European Workshop on Dependable Computing,* LNCS 7869, pp. 126-133, 2013.

[37] Dyninst, [online]. Available: http://www.dyninst.org/. [Accessed 02 June 2013].

[38] AUTOSAR DEM, "Specification of Diagnostic Event Manager v4.2.0," AUTOSAR, R4.0.