SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

Faculty of Informatics and Information Technologies

Reg. No.: FIIT-100241-117028

Software-implemented fault tolerance

Bachelor thesis

Study programme: Informatics
Study field: Computer Science
Training workplace: Institute of Informatics, Information Systems and Software Engineering
Thesis supervisor: Ing. Ján Mach

Bratislava 2025 **Filip Ďuriš**

Slovak University of Technology in Bratislava
Institute of Informatics, Information Systems and Software Engineering

Faculty of Informatics and Information Technologies
Academic year: 2024/2025
Reg. No.: FIIT-100241-117028

BACHELOR THESIS TOPIC

Student:	**Filip Ďuriš**
Student's ID:	117028
Study programme:	Informatics
Study field:	Computer Science
Thesis supervisor:	Ing. Ján Mach
Head of department:	doc. Ing. Ján Lang, PhD.

Topic: **Software-implemented fault tolerance**

Language of thesis: English

Specification of Assignment:

Aplikácie kritické z pohľadu bezpečnosti (automobilový priemysel) alebo misie (vesmírny priemysel) potrebujú dosahovať vysokú úroveň spoľahlivosti a bezpečnosti. V poslednej dobe tieto aplikácie potrebujú čím ďalej vyšší výpočtový výkon, načo sú potrebné aj tie najvyspelejšie čipy. Tie čerpajú výhody zo zmenšovania štruktúr tranzistorov, vyššej hustoty integrácie a nárastu frekvencie. Nevýhodou je, že tieto zmeny vedú aj k vyššej náchylnosti na poruchy spôsobené v dôsledku radiácie, čoho prejavom sú náhodné preklopenia hodnôt pamäťových prvkov na čipe. Jedným zo spôsobov zabezpečenia spoľahlivej prevádzky aj pri takýchto podmienkach je použitie softvérovo implementovanej tolerancie hardvérových porúch. Tento spôsob zabezpečenia referuje na sadu techník, ktoré umožňujú softvéru detegovať a v rámci možností aj opraviť poruchy, ktoré ovplyvňujú hardvér, na ktorom sa daný softvér vykonáva. Analyzujte a porovnajte dostupné techniky softvérovo implementovanej tolerancie hardvérových porúch. Navrhnite aplikáciu na báze operačného systém reálneho času FreeRTOS pre vnorený systém, ktorá bude zabezpečená vybranou technikou. Navrhnite testovacie scenáre a vyhodnoťte zabezpečenie.

Length of thesis: 40

Deadline for submission of Bachelor thesis:	12. 05. 2025
Approval of assignment of Bachelor thesis:	15. 04. 2025
Assignment of Bachelor thesis approved by:	doc. Ing. Ján Lang, PhD. – Study programme supervisor

I declare on my honor that I have prepared this work independently, on the basis of consultations and using the listed literature.

In Bratislava 12.5.2025 Filip Ďuriš

Annotation

Slovak University of Technology Bratislava
Faculty of Informatics and Information Technologies
Degree Course: Informatics

Author: Filip Ďuriš
Bachelor's Thesis: Software-implemented fault tolerance
Supervisor: Ing. Ján Mach
Máj 2025

This thesis explores software-implemented fault tolerance for embedded systems, with a particular focus on solutions that do not require specialized or redundant hardware. As embedded systems grow increasingly complex and are deployed in harsher or more unpredictable environments, the risk of system failure rises accordingly. Traditional approaches to fault tolerance often rely on hardware redundancy, such as error-correcting memory or replicated components, which significantly increases development cost, power consumption, and physical space requirements. In contrast, this research investigates software-only techniques that can detect, mitigate, or recover from faults without the need for specialized hardware. The study examines various software fault-tolerance techniques and evaluates their trade-offs in terms of reliability, performance overhead, and implementation complexity. A practical case study is presented using the Rust programming language and the FreeRTOS real-time operating system on a simulated RISC-V platform with fault insertion capability. Through testing of selected techniques under simulated fault conditions, the findings demonstrate that software-based fault tolerance can provide a meaningful level of protection, particularly for non-critical applications.

Anotácia

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Študijný program: Informatika

Autor: Filip Ďuriš
Bakalárska práca: Softvérovo implementovaná tolerancia porúch
Vedúci bakalárskej práce: Ing. Ján Mach
Máj 2025

Táto práca sa zaoberá softvérovo implementovanou odolnosťou voči poruchám vo vnorených systémoch, so zameraním na riešenia, ktoré nevyžadujú špecializovaný alebo redundantný hardvér. S rastúcou zložitosťou vnorených systémov a ich nasadzovaním v náročnejších a menej predvídateľných prostrediach sa zvyšuje aj riziko zlyhania systému. Tradičné prístupy k odolnosti voči poruchám sa často spoliehajú na hardvérovú redundanciu, ako sú pamäte s korekciou chýb alebo duplikované komponenty, čo výrazne zvyšuje náklady na vývoj, spotrebu energie a nároky na fyzický priestor. Táto práca skúma výlučne softvérové techniky, ktoré dokážu poruchy detegovať, zmierniť ich dopad alebo sa z nich zotaviť bez potreby špecializovaného hardvéru. Štúdia analyzuje rôzne softvérové prístupy k zabezpečeniu odolnosti voči poruchám a hodnotí ich z hľadiska spoľahlivosti, výkonnostnej záťaže a zložitosti implementácie. Praktická demonštrácia využíva programovací jazyk Rust a real-time operačný systém FreeRTOS na simulovanej platforme RISC-V. Pomocou testov vybraných techník v simulovaných poruchových podmienkach práca ukazuje, že softvérovo implementovaná odolnosť voči poruchám môže poskytnúť zmysluplnú úroveň ochrany, najmä pre nekritické aplikácie.

Contents

Acronyms

ALU	arithmetic logic unit
BSS	block starting symbol
CFCSS	Control Flow Checking by Software Signatures
DP	data path
ECC	error correction code
EDDI	Error Detection by Duplicated Instructions
FI	fault insertion
GPR	general purpose register
IR	intermediate representation
MDU	multiply-divide unit
NSCP	N self-checking programming
NVP	N-version programming
RA	return address
RAM	random access memory
RFC	register file control
SEU	single-event upset
SWIFT	Software Implemented Fault Tolerance
TP	trap interrupt register

1 Introduction

With the increasing demand for high-performance embedded software comes the inevitable, difficult task of ensuring error-free functioning of ever-more complex systems. The increased complexity, both in terms of hardware components and software features, increases the number of failure points where faults can manifest an error.

Faults can be caused by both external factors beyond our control such as radiation in space interfering with the hardware, but also as a result of human error while designing the system. Due to the incredibly complex nature of this problem, it is unlikely that we will be designing completely error and fault-free software and hardware in the near future. This makes fault-tolerance an important aspect of software design, especially when designing critical systems whose failure could endanger human life. However, as with most things, fault-tolerance and reliability is a trade-off, which usually comes at the cost of performance and development time.

Historically, specialized hardware was the go-to choice for implementing fault tolerance, specifically by hardening and or duplicating the components. This approach requires designing and producing non-standard computer components, exponentially increasing the development cost. Lately, a more economical solution began gaining traction for non-critical missions. Namely, using off-the-shelf components without specialized hardening, while using software redundancies to implement fault tolerance.

This thesis will aim to analyze various common software-implemented fault-tolerance methods. We will look at the benefits and drawbacks of the utilized methods, as well as construct a working demo based on FreeRTOS running on a simulated RISC-V core, with fault insertion capability, that implements and tests the effectiveness of some selected methods using the Rust programming language.

For the sake of consistency in the terms used in this document, we will refer to the naming conventions and definitions as outlined in [3]. Below are the definitions of the most crucial terms used. Other specific terms will be defined on per-need basis throughout the thesis.

A **service** refers to the intended function delivered to the user by some system. An example could be a remote thermometer delivering correct temperature readings. The **system** encompasses all components necessary to deliver this service, including the processor with its internal elements, I/O peripherals, sensors, and the associated program code. For instance, a microprocessor with temperature sensors and I/O elements. Everything that is considered as a part of this system lies within the **system boundary**. This boundary can be thought as an imaginary partition separating our system from the environment.

Failure is an event that occurs when the service permanently deviates from correct functioning. A service failure is a transition from correct service to incorrect service, i.e., to not implementing the system function [3]. For example, a failure might occur if an I/O component ceases to communicate with the system. Failures are usually caused by an **error**, which is a deviation of the service from its correct state. It is a part of the system's state that may lead to service failure [3]. Errors are the observable result of issues within the software and or hardware, for example, an unexpected change to the output variables. This unexpected change is commonly caused by a **fault**, which is the actual or hypothesized cause of an error [12]. Faults are usually considered dormant until manifested, causing an error [3]. An example being a bit-flip in memory caused by radiation.

Being able to detect and tolerate errors originating from faults is the topic of this thesis and is commonly referred to as **fault tolerance**. Fault tolerance refers to the system's ability to continue functioning correctly despite the presence of faults. For example a system delivering service correctly with one of its sensors not responding. Sometimes, the fault might lead to the system not fully meeting its functional requirements, however, it might still be usable to certain extent. This is referred to as **service degradation**, an example of which could be decreasing the frequency of sensor readings, or reporting approximate data.

2 Requirements Specification

The primary objective of this project is to analyze, implement, and evaluate selected software-based fault tolerance techniques. The focus is on embedded systems operating in environments where faults can compromise system behavior. These issues are especially relevant in resource-constrained systems where safety and reliability are paramount, such as the aerospace or the automotive industry. This section outlines the functional and non-functional requirements that the proposed system must fulfill to meet its objectives.

Functional Requirements

- The system shall implement multiple software-based fault tolerance techniques.
- The system shall be capable of detecting and reporting transient faults at runtime.
- The system shall provide a mechanism for recovering from detected faults where applicable.

Non-Functional Requirements

- The system shall introduce minimal performance overhead relative to the unprotected baseline.
- The solution shall be implemented entirely in software using the Rust programming language.
- The system shall be portable and capable of running on a 32-bit RISC-V platform using the FreeRTOS operating system.
- The implementation shall be maintainable and modular, enabling future extension or substitution of fault tolerance mechanisms.

Ultimately, the project aims to deliver a practical and reproducible evaluation framework for software-based fault tolerance, enabling developers to assess the trade-offs between fault tolerance, runtime overhead, and system complexity in embedded contexts.

3 Types of faults

In order to be able to analyze fault-tolerance techniques, we first need to understand the various types of faults that we might encounter. This list is not exhaustive and only covers fault types relevant to this thesis, for more complete list see [3]. It is also important to understand two different classifications are not always mutually exclusive.

Faults can be further split into various categories, the ones mostly relevant to us are **external faults** which originate from outside the system boundaries and propagate into the system. Which also includes **natural faults** that are caused by natural phenomena without human participation. These faults can affect the hardware and the software, which is why we can further classify them as **hardware faults** and **software faults** respectively [3].

3.1 Transient fault

Transient fault is a temporary fault which results in an error, such as a bit-flip in the register file [12]. The most common subtype of transient fault is single-event upset (SEU), when talking about faults, we are referring to SEUs, unless specified otherwise. A key characteristic of transient fault is the possibility to correct it by overwriting the corrupted region. Retrying the same process multiple times is usually enough to deal with a transient fault, since it is extremely unlikely a transient fault will occur repeatedly within the same process at successive time intervals. If a fault persists across multiple attempts, it is possible we are dealing with a permanent fault.

3.2 Permanent fault

Permanent fault is a lasting fault within the system that requires maintenance to remove [12]. This could come in various forms, such as damaged hardware, corrupted memory or missing code. When dealing with a permanent fault, retrying the same process is usually not enough as the same error will keep reoccurring. This approach requires alternatives and fallbacks within the system, which is outlined in more detail in section 4.2.

3.3 Hardware faults

Hardware faults [3] are caused by external factors beyond our control as software developers. They are usually caused by environmental influences, such as cosmic radiation in space, electromagnetic fields, adverse weather conditions or heavy system usage.

One of the most common hardware faults is memory corruption, which can appear in many forms and result from a wide range of causes. For instance, radiation exposure in space can cause SEUs, flipping individual bits in memory and altering data unpredictably. Similarly, physical damage to storage media, such as hard drives or SSDs, might corrupt specific regions of the file system, making certain data inaccessible or incorrect.

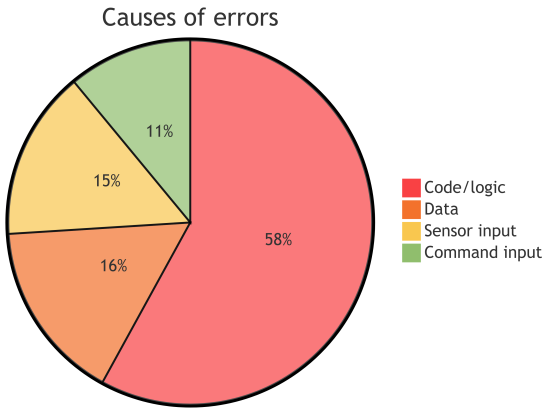

Figure 1: Causes of errors [15]

3.4 Development faults

Development faults is a category which includes all the various faults whose causes are introduced during software development stage [3]. According to research conducted by NASA, majority of errors stem from faults within the code and logic of the afflicted software, followed closely by faults in data [15] (see Figure 1). We are employing the use of a language with built-in static code analysis - Rust, to minimize the possibility of development faults, allowing us to focus on the main problem.

4 Fault-tolerance techniques

This section focuses on the analysis and comparison of theoretical aspects of fault tolerance and detection, notes certain methods that were already implemented and tested in practice and outlines methods chosen for our own implementation.

4.1 Single-version

Single-version is a category of techniques which focus on creating a singular, robust implementation of software by integrating safety checks and redundancies directly into the software design process [19].

This technique emphasizes the detection of errors within the software and the ability to recover from them. Error detection typically involves monitoring the system for unexpected behaviors or inconsistencies, which could signal the presence of a fault [12]. Recovery mechanisms then act to mitigate the effects of these faults.

Handling an error can be done in various ways. Most common would be to backtrack to a saved checkpoint and retry the part of the application where an error was discovered in the hopes of getting the correct result on the subsequent retries. This usually works well with transient faults, but it is likely to fail in the presence of a permanent fault.

Drawbacks of single version techniques are primarily the lack of alternatives and fallbacks, should the program fail. Single-version techniques heavily rely on error detection and recovery, which might not always work in practice. The reason single-version techniques are viable is an observation that transient faults are way more common than permanent faults [18], meaning that single-version is usually enough for noncritical parts of the system.

4.1.1 Modularity

Perhaps the simplest way we can create a more resilient software is to structure it into independent modules. Each module should handle one task and, when possible, not directly rely upon any other modules for its functionality.

A technique commonly utilized to achieve modularity is partitioning, which can be divided into horizontal and vertical partitioning. Horizontal partitioning aims to split the software into independent structural branches communicating through interfaces. Vertical partitioning splits the software

in a top-down fashion, where higher level modules are tasked with control logic, while lower level modules do most of the processing [19].

Benefit of partitioning is the ability of software to isolate errors. Provided the software is correctly structured, an error occurring in a single module should not propagate to other modules. Meaning we can use modularity as a way to pinpoint the erroneous parts of software and attempt recovery. If recovery is not possible, the software should still be able to partially function, given that other parts of the software are not influenced by the fault. In most situations, partial functioning of a software is preferable to a complete shutdown.

4.1.2 Error detection

Fault-tolerant single-version application should meet two main criteria: self-protection and self-checking. Self-protection means that the application should be able to protect itself from external corruption by detecting errors in its input and output data as well as errors in its control flow. Self-checking means that the application component must be able to detect errors within itself and prevent propagation of these errors into other components [19]. These two traits combined can be together considered as the ability of "error detection".

Error detection covers a wide range of techniques used to locate errors and mitigate them. Some common approaches include:

a) checksums and error correction codes (ECCs), which embed additional metadata with the actual data in order to verify integrity and attempt to correct corrupted data. This approach allows for some degree of memory corruption mitigation but comes at the cost of memory overhead and additional processing per data-chunk which uses checksum or ECCs.

b) assertion and runtime checks, which perform independent checks on the data during execution which ensures the data matches the expected outcomes at certain checkpoints.

c) watchdog timers, whose main purpose is to catch deadlock states by giving a task a certain amount of time to execute before aborting it.

Error detection is a crucial aspect of single-version application, since there is no alternate version to fall back upon.

4.1.3 Fault recovery

Fault recovery is a process of restoring the program to a functional state after an error has been detected. Fault recovery in single-version techniques mostly consists of a program rollback, also known as *backward recovery* [12]. This can be either a full restart, effectively rolling back the program to its initial state, or a more advanced technique such as checkpoint and restart[19].

Checkpoint and restart could be considered the basis of some of the more advanced fault tolerance techniques. Before the execution of a critical process a checkpoint is created, which captures some or all of the program state. At the end of the execution, an acceptance check is performed on the process output, if an error is detected rollback is initiated and the process is restarted.

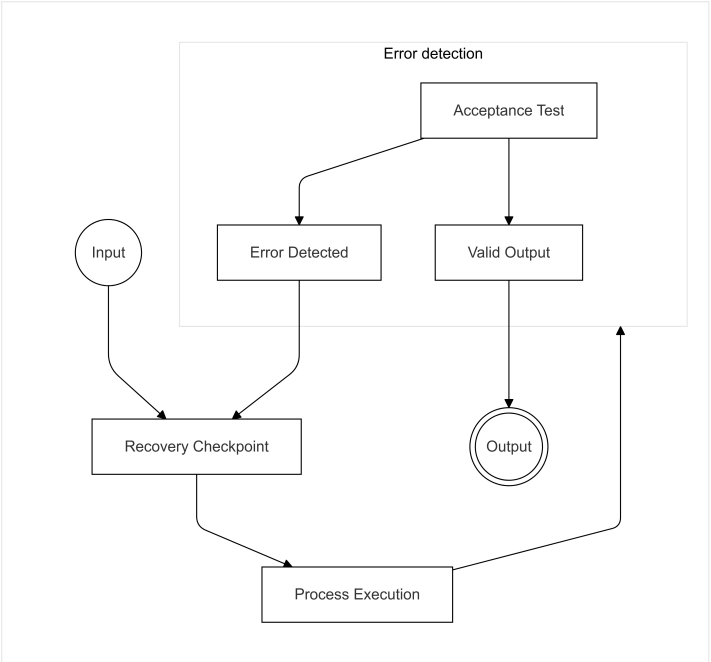

Figure 2: Checkpoint and restart

An alternative to backward recovery is *forward recovery* [12], which is the process of correctly proceeding with execution even if error is detected. This is usually done by creating and storing three (or more copies) of critical data (*data triplication*), performing the same operation on all copies, and using a majority voter to detect errors.

Forward recovery via data triplication is, generally speaking, more computationally demaning than backward recovery, since the same operation(s) has to be performed per each data copy, even during fault-free operation, and the copies have to be compared at certain synchronization points. On the other hand, backward recovery incurs minimal overhead while setting up a checkpoint and only requires additional computation when an error is actually detected.

4.1.4 Considerations for single-version techniques

Single-version fault tolerance techniques are largely easier to develop when compared to the ones discussed in the following section. With this relative simplicity comes the drawback of not being well suited for some situations. While these techniques should work well for transient faults that are unlikely to reoccur, a single version fault-tolerant system will fail when attempting to deal with persistent faults. Even if an error is successfully detected, single version techniques provide no clear way of dealing with errors that stem from, or are heavily influenced by the design of the system. As an example, if the memory region containing a critical function has been permanently damaged, it matters not how many times we attempt to rerun the function, it will always cause an error. For this reason, single version techniques should be reserved for parts of software which are not mission critical.

In order to effectively deal with permanent faults, we need to consider multi-version techniques covered in the following section. A lot of these techniques take inspiration from single version techniques, or even directly build on top of them.

4.2 Multi-version

Multi-version techniques build on the idea of multiple software versions which all meet the same specifications. The rational behind multiple version is that different components should fail differently [19]. These versions are interchangeable in terms of their output, but each version executes differently from the rest, ensuring that no two version share the same resources. Multiple versions of the same software are executed either in sequence or in parallel [19], each utilizing different error detection and recovery methods, to have the highest probability of completing the task successfully.

4.2.1 Recovery blocks

Recovery blocks [21] is a simple form of multi-version programming, expanding upon the idea of "checkpoint and restart". Unlike its single version counterpart, however, recovery blocks does not re-execute the same code again, but instead chooses a different version to try next.

A key advantage of the recovery blocks technique is that, in most cases, the primary version will be the only one to execute [21], allowing alternate versions to prioritize redundancy and safety over performance. This enables the design of backup versions with gradually reduced performance requirements, ensuring robust fallback options without excessive resource consumption.

In some situations, the alternate versions do not necessarily have to produce a correct result, but a result that satisfies the acceptance check [21]. Alternate version can gradually degrade the service in order to accomplish the bare minimum requirements.

The overall success of the recovery block scheme rests to a great extent on the effectiveness of the error detection mechanisms used — especially (but not solely) the acceptance test. The acceptance test must be simple otherwise there will be a significant chance that it will itself contain faults, and so fail to detect some errors, and/or falsely identify some conditions as being erroneous. Moreover, the test will introduce a run-time overhead which could be unacceptable if it is very complex. The development of simple, effective acceptance tests can thus be a difficult task, depending on the actual specification [21].

4.2.2 N-version programming

N-version programming (NVP) [4] extends the multi-version technique by running "N" independent versions in parallel or in sequence, hence "N-version" programming. In this approach, each version meeting the same specifications independently performs the task, and the final outcome is determined through a consensus mechanism that evaluates the results from all N executions.

This consensus is usually achieved through a voting algorithm, which aggregates the outputs from each version and selects the result agreed upon by the majority. Selection algorithms are an entire topic of its own covered well by [1].

This voting approach to handling errors is sometimes referred to as **fault masking**, since we are not necessarily concerned with detecting an error, but rather getting an acceptable output even in the presence of a fault.

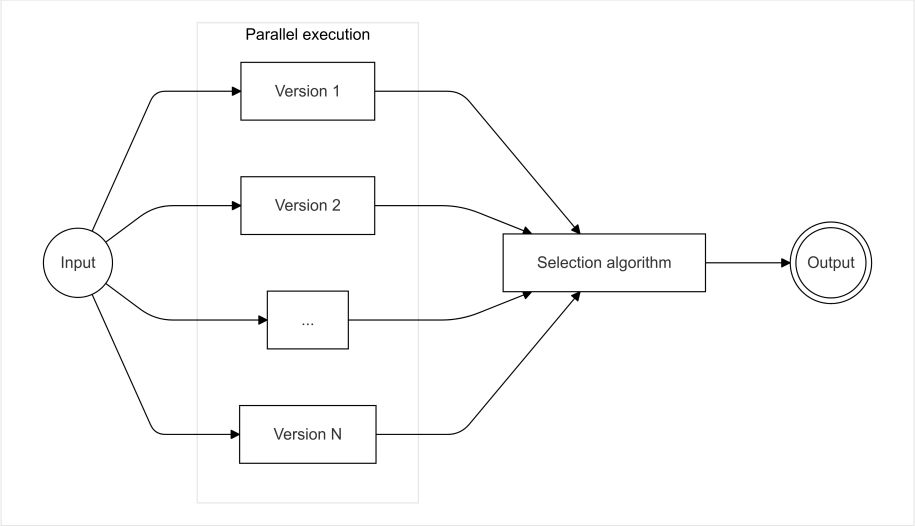

Figure 3: N-Version Programming

The primary drawback of NVP is its requirement to execute all of the versions before determining the final output. This can be highly resource intensive, especially for large or complex tasks, as it requires significant computational power and memory to run multiple versions simultaneously.

For systems with limited resources, such as embedded systems, this approach can be particularly inefficient. The need to allocate resources for each version can strain the system's capabilities, potentially reducing its overall performance and responsiveness. As a result, while NVP enhances fault tolerance and reliability, it may not be suitable for applications where resource constraints are a priority or where processing efficiency is critical.

A consideration for NVP is the possibility of an error not being a random event, but rather a function of the input variables [10]. Therefore, even multiple versions running in parallel could all fail and give erroneous results. This makes the selection algorithm a critical failure point which NVP on its own does not address.

4.2.3 N Self-checking programming

N self-checking programming (NSCP) [11] is an extension of the classic NVP, where on top of executing multiple versions, each version also contains its own independent acceptance test or recovery block, before the results are passed to the selection logic. The selection logic then selects the "topmost" possible version that reports a correct output.

A version-specialized acceptance check is an interesting addition, as it provides the opportunity to take advantage of the version implementation details. We can specifically tailor the check to consider the inner workings of the version to detect errors and possibly even correct them before proceeding to the selection stage.

The drawback here is the increase in complexity over the more simple recovery blocks which uses a shared acceptance check, or the simple N-version approach which opts for masking instead. By creating more acceptance checks we are introducing more opportunities for errors to manifest, while also spend more resource on development.

4.2.4 Considerations for multi-version programming

The primary challenge associated with multi-version programming is the significant effort required to develop, test, and maintain several versions of software that perform the same function. This process can be resource-intensive, leading to increased costs, making it unfeasible for smaller projects or for teams with a limited budget.

To achieve effective multi-version programming, each version must be carefully designed to execute the same task while incorporating distinct failure mechanisms. Ensuring that no two versions fail in the exact same way is very difficult and in practice not always possible.

Research has been conducted into other methods that improve upon the aforementioned multi-version techniques, such as the "t/(n-1)-Variant programming" [22] - extension of NVP which uses fewer comparisons as opposed to NVP or NSCP and can tolerate up to t faults. However, the findings do not conclusively prove that the sharp increase in complexity justifies the marginal benefits this improved technique provides.

4.3 Data diversity

Faults within the data are the second largest cause of errors [15], with that in mind, it is sometimes not enough to execute different version of a software on the same data in order to get an acceptable output. Instead, *data diversity* [9] might be required to ensure correct execution. Data diversity is an orthogonal method to the previously mentioned design diversity methods. It can be used on its own, or in combination with other fault tolerance methods.

4.3.1 Failure regions

In many situations only very specific conditions will result in an error, we call these specific condition edge-cases. Even with thorough testing, there is no guarantee that all edge-cases will be caught during development, since the *failure domain* (the set of input points that cause program failure [2]) can be extremely small. We can take advantage of this, however, by using "reexpressed" input on a subsequent execution of a procedure, since even small adjustments of the input are likely to move it away from the failure domain.

4.3.2 Data reexpression

Data reexpression is generation of logically equivalent data sets. Any mapping of a program's data that preserves the information content of the data is a valid reexpression algorithm [9]. A simple approximate data reexpression algorithm for a floating-point quantity might alter its value by a small percentage. Similar concept can also be found in *approximate computing* [17]. The allowable percentage by which the data value could be altered would be determined by the application. In applications that process sensor data, for example, the accuracy of the data is often poor and deliberate small changes are unlikely to affect performance [9].

4.4 Previous work

While the analysis so far was mostly theoretical, this section outlines methods already implemented and tested in practice, giving an insight into what we can expect when implementing our own fault-tolerance methods.

4.4.1 EDDI

Error Detection by Duplicated Instructions (EDDI) [14] is a pure software, technique that enhances runtime error detection by duplicating program instructions during compilation. Each original instruction (master instruction) is paired with a duplicate (shadow instruction) that operates on separate registers or variables. At key synchronization points, such as before memory writes or control transfers, the system compares the results of the master and shadow instructions. Any mismatch indicates an error, triggering a fault handler. EDDI leverages idle resources in super-scalar processors by interleaving these redundant instructions to maximize instruction-level parallelism while minimizing performance overhead. It is particularly effective in detecting transient faults, memory corruption, and control-flow deviations without requiring any hardware modifications, achieving over 98% fault tolerance in benchmark programs.

4.4.2 CFCSS

Control Flow Checking by Software Signatures (CFCSS), originally proposed in [13] is a pure software method that embeds control-flow checking logic into a program at compile time. It is capable of detecting faulty branching or jumps within the program.

The program is first divided into basic blocks, each representing a unique node in the control-flow graph. A unique signature is assigned to each node, and additional instructions are inserted to monitor the program's control flow during execution.

At runtime, the control flow is validated using a designated general purpose register (GPR), which holds the expected signature. When control transfers to a new basic block, a new runtime signature is generated using a lightweight XOR-based function that combines the previous and current signatures. This updated signature is then compared with the expected one at the destination block. If a mismatch is detected, it indicates an illegal control transfer, and execution is redirected to an error handler.

CFCSS is particularly advantageous because it requires no dedicated hardware and can operate even in environments without multitasking support. Its effectiveness was validated through fault injection experiments, which showed that while 33.7% of branching faults went undetected in unprotected programs, only 3.1% remained undetected when CFCSS was ap-

plied, demonstrating an order-of-magnitude improvement in error detection coverage. CFCSS introduces moderate code size and execution time overhead (especially in branch-heavy programs).

4.4.3 SWIFT

Software Implemented Fault Tolerance (SWIFT) [16] is a compiler-based technique for detecting transient faults using software-only transformations, without requiring any specialized hardware. It builds upon earlier methods like EDDI by duplicating instructions and inserting validation checks, but improves upon them with several key optimizations to reduce memory and performance overhead. Unlike EDDI, SWIFT excludes memory from the *sphere of replication* (the logical boundary within which all states are logically or physically replicated [12]), assuming protection is already provided by ECC in modern systems. It introduces an enhanced control-flow checking mechanism using software signatures, allowing it to detect illegal control transfers more reliably. Additionally, SWIFT incorporates optimizations such as restricting signature checks to store-containing blocks and eliminating redundant branch validation. These design decisions enable SWIFT to achieve high fault tolerance with lower execution and code size overhead.

4.5 Chosen methods

When considering the fault-tolerance methods to implement, the most crucial factor was the feasibility of implementation. Since we are working with a high-level language, certain methods such as EDDI are not possible to effectively implement, since they require implmenentation at assembly instruction level, however, modified or simplified version of these methods may still be implemented.

Ultimately, checkpoint and restart was chosen as the backbone of fault recovery, as it can be easily combined with other methods. CFCSS, with various modifications to make its implmentation feasible in Rust, was chosen to secure the control flow of the program. Finally, recovery blocks and N-version programming were chosen to facilitate multi-version functionality. Additional miscellaneous fault-tolerance methods were also used, such as variable protection through duplication to support the aforementioned methods.

5 Motivation for Rust

The secondary purposes of this thesis is to analyze the suitability of Rust language for the implementation of fault-tolerant software. Historically, C has been the most dominant language in this problem space, but Rust has numerous benefits which make it a strong competitor.

5.1 Safe by design

The goal of the Rust language is to eliminate the most common software bugs at the compiler level. An example of this is Rust's *ownership and borrowing model* - a static code analysis done by the compiler [5]. It ensures memory safety of the program by explicitly disallowing references to deallocated values, which prevents null pointer dereferencing, and also prevents race conditions by limiting the access to mutable variables. Rust's ownership model is a complex topic and explaining it in full is not within the scope of this thesis, more information can be found in the official Rust book[1].

5.2 Robust error handling

Another major benefit of Rust is its approach to error handling. Rust splits errors into two categories: *Unrecoverable* errors immediately terminate the execution of the program using the *panic!()* macro and *recoverable* errors which are return values of a function using the Result enum [6].

```
fn foo() -> Result<u32, ()> {
    ...
}
fn main() {
    let Ok(res) = foo() else {
        // Function returned with an error
    }
}
```

Figure 4: Rust - error as a value

Using the approach of *error-as-a-value* we can determine if a function executed successfully by simply checking its return value (see figure 4). This

[1] https://doc.rust-lang.org/book/

can act as a very quick and rudimentary form of fault detection, since it can be easily applied to any function.

5.3 Macro system

Rust has a powerful macro system which in essence works as a code preprocessor acting over the abstract syntax tree. Unlike C macros, which only work as direct text replacement, Rust macros allow for arbitrary modification and generation of code before the compilation step. Using Rust macros, various fault tolerance techniques can be implemented. An example of utilizing macros is highlighted in this very thesis, where a Control Flow Checking using Signature Streams (CFCSS) technique is implemented using Rust macro system.

5.4 Low-level control and C integration

Rust demonstrates comparable functionality to C when it comes to low-level control - example being the ability to define custom heap allocator or the ability to use in-line assembly from within Rust. Since Rust uses LLVM[2] (a suite of compiler tools commonly used by C and C++ programming languages) its compiler is able to link against existing C code. We can also easily use C functions within Rust by declaring the function signature (see figure 5). This makes for an easy integration with existing C projects making Rust well suited as a higher level extension. This thesis demonstrates the use of Rust with FreeRTOS codebase written in C[3].

```
unsafe extern "C" {
    fn puts(string: *const c_char, len: usize) -> i32;
}
```

Figure 5: Rust - Using C function within Rust

[2] https://llvm.org/

[3] https://github.com/FreeRTOS/FreeRTOS-Kernel

6 Environment

This section outlines the implementation environment, including hardware constraints, system configuration, and software integration. It also describes key modifications made to support our specific use case.

6.1 RISC-V

The implementation was developed for a 32-bit RISC-V core, running in a simulated environment. The simulation features a single RAM region with a total size of 1MB. Of this, 8KB was allocated for the stack and another 8KB for heap memory. These parameters can be adjusted as needed for testing and experimentation.

A key feature of the RISC-V architecture is its interrupt and exception handling mechanism, which distinguishes between two primary types. *Software exceptions* are triggered by the program itself, often to perform system calls, while *hardware interrupts* are initiated by hardware events, such as timers or peripherals [20]. The combination of these two systems allows for error detection and transferring control for recovery or rollback.

6.2 FreeRTOS and interrupt management

The system runs on the FreeRTOS kernel, a lightweight real-time operating system widely used in embedded systems. FreeRTOS is designed for minimal resource usage, making it suitable for constrained environments like our RISC-V simulation.

FreeRTOS uses *tasks* to run user code. Task is a wrapper around user code which include the specific code's context. Tasks can be instantiated at any point during the program's lifetime. FreeRTOS *scheduler* handles the execution of tasks as well as switching between them using the RISC-V hardware interrupt - *Machine Timer Interrupt*.

In addition to timer-based preemption, FreeRTOS also makes use of software exceptions, such as those triggered by the RISC-V *ecall* instruction. These interrupts are typically used for system-level operations and are routed, along with hardware interrupts, through a shared interrupt handler. In our case, this handler is implemented using a vectored interrupt table to efficiently dispatch control to the correct service routine.

6.3 Rust integration with FreeRTOS

To enable Rust development on top of FreeRTOS, we used the open-source FreeRTOS-Rust library[4]. This library provides Rust bindings to the FreeRTOS C ABI. Modifications had to be done to ensure compatibility with our simulated environment. Notably, the used library required atomic instructions for certain functionality, however, our platform did not support atomic operations. Since we are only simulating single core, hardware multithreading is not a concern, as such, we could safely replace all atomic operations with their non-atomic counterparts.

On top of this, RISC-V Rust library[5] was used to facilitate the compilation of Rust into a RISC-V acceptable binary. This library provided functionality such as designating the program entry point, designating the exception and interrupt handler functions and provided functions to work with RISC-V specific registers. However, the used library also added significant performance overhead in the form of block starting symbol (BSS) clearing (setting all static variables to 0) before the main program start. Clearing BSS is useful to ensure all variables are correctly initialized if a default value is not assigned. However, because Rust does not fundamentally allow for the use of non-instantiated variables this step was not necessary for our use case. By removing this segment we saved significant computation time, reducing the possible area for faults to manifest.

[4] `https://github.com/lobaro/FreeRTOS-rust`
[5] `https://github.com/rust-embedded/riscv`

7 Implementation

The implementation section outlines some selected methods which were implemented using Rust and FreeRTOS and tested on a RISC-V core simulation. The implemented methods were modified to make them feasible to implement in a high level language.

7.1 Checkpoint and restart

Checkpoint and restart system is the backbone of our fault-toleranace framework, facilitating backward recovery. It can be used on its own to provide basic level of redundancy, or it can be combined with other techniques to boost their fault-tolerance potential. The implementation is partially inspired by *libft* C library mentioned in [7].

7.1.1 Creating a checkpoint

```
asm!(
    // Storing registers in a static variable
    "lui   t0, %hi(CHECKPOINT)",
    "addi  t0, t0, %lo(CHECKPOINT)",
    "sw x1,   4(t0)",
    "sw x2,   8(t0)",
    ...
    "sw x31,124(t0)",
    "call set_checkpoint",
    // Restoring registers to their previous state
    "lui   t0, %hi(CHECKPOINT)",
    "addi  t0, t0, %lo(CHECKPOINT)",
    "lw x1,   4(t0)",
    "lw x2,   8(t0)",
    ...
    "lw x31,124(t0)",
)
```

Figure 6: Rust - Creating checkpoint

Checkpoint is generated by creating a *checkpoint context* during which the current state of the application is stored. Namely, the general purpose registers (GPRs) x1 - x31 in RISC-V 32bit (x0 is ommitted as its hardwired

to always be 0, the hardware implementation is responsible for ensuring this and as such this register cannot be written to) are stored into a static memory location before the execution of a *checkpoint block* (the part of code protected by the checkpoint, sometimes referred to as *sphere of recovery* [12]). Additionally, a *checkpoint address* (address of the first recovery instruction, Fig. 6 line 11) is stored along with the GPRs. If we directly used the saved return address (RA) - x1 - to jump back after an exception, the control would be transferred to outside our checkpoint context.

Higher-level programming languages do not directly provide the functionality to manipulate register values. This means assembly instructions have to be used to facilitate the low level access as seen in figure 6.

In order to set the checkpoint address, we utilize a trick (shown in figure 7) whereby calling a function we now have access to the address of the instruction immediately following the function call (figure 6 line 11) saved in the RA register. We store the value of this register along with the registers in the checkpoint variable.

```
1 fn set_checkpoint() {
2     unsafe {
3         asm!(
4             "sw ra, ({0})",
5             in(reg) &(CHECKPOINT.ret_addr),
6         );
7     }
8 }
```

Figure 7: Rust - Set checkpoint

7.1.2 Storing the checkpoint

As mentioned in section 7.1.1, the checkpoint information is stored in a static memory location. This might seem as an arbitrary decision given that storing the checkpoint on the stack or the heap are also an option. Each of the listed approaches comes with its own issues. Using a static memory location limits the number of checkpoints that can be stored. Using the stack for checkpoint storage can be very unreliable as stack corruption can easily occur should the stack pointer manifest a fault. And using the heap is generally slow

and prone to error stemming from pointer corruption or premature memory freeing which could occur in the presence of faults.

Our implementation uses a hybrid approach of using a static memory location for the storage of the *primary checkpoint* (the last checkpoint that was set), while pushing any older checkpoints on the stack. This heightens the chance that at least the primary checkpoint will be available. Since a static memory location is used, we could also store the primary checkpoint in a hardware-hardened memory section. Once the primary checkpoint context ends, the previous checkpoint is popped from the stack and can be used again. This allows for infinite nesting of checkpoints, provided we have enough memory.

7.1.3 Returning to the checkpoint

After setting the checkpoint, the program execution continues as per usual, until either the checkpoint block finishes successfully, or an error is detected. An error can be detected in two ways, an exception is detected by the processor, or an error is detected via checks inserted into the checkpoint block.

If the processor detects an error the exception handler subroutine is called. We can hook into this subroutine and use it to restore the stack pointer to the pre-checkpoint block state and then perform a jump back to the saved checkpoint address.

If an error is detected via checks inserted by the programmer, checkpoint restart can be triggered by simply returning an error (Err()) from within the checkpoint block.

Both forms of error detection and returning to checkpoint result in the restoration of the program context to the program state in which the checkpoint function was called. After the context was restored, checkpoint block will be attempted again if retries are allowed (retries parameter != 0, see Fig. 8 - retries is set to 2), otherwise the entire checkpoint context returns an erroneous value.

7.1.4 Overhead

The implemented checkpoint system incurs minimal instruction overhead. Additional instructions are only inserted when creating the checkpoint context and just before the end of the checkpoint context. This comes out to 79 additional instructions to save and restore program state, plus approximately

```rust
if let Err(Error::MaxRetriesReached(n)) = checkpoint(2, || {
    // Checkpoint block here
    ...
    Ok(())
}) {
    println(&format!("Max retries reached: {}", n));
}
```

Figure 8: Rust - Using the checkpoint system

300 instructions to save and restore old checkpoint and perform various error checks, without factoring in any compiler optimizations.

This overhead is constant for the entire checkpoint block, irrespective of the checkpoint block size. As such, the ratio of overhead instructions to the checkpoint block instructions is solely determined by the size of the protected checkpoint block.

7.1.5 Considerations

Checkpoint system only facilitates basic restart and retry functionality. It can be used on its own when only basic level of protection is required, however, in most cases, checkpoint and restart system is meant to be used as a framework to build upon.

Our implementation of the checkpoint and restart system only saves the minimal required state, which in the case of RISC-V, is the GPRs. This limits the use of the checkpoint system as other memory regions such as stack or heap are not stored. If the checkpoint body has any side-effects, such as modifying global variables, variables outside of the sphere of recovery, these alterations will not be reversed in the case of an error and subsequent checkpoint return. Some techniques which can deal with this issue are outlined in section 7.3.

Another important aspect to mention is that our implementation does not save registers other than the GPRs. This means special registers such as control and status registers (*mtvec*, *mcause* or *mstatus*) are not protected by the checkpoint system. The fault-insertion simulation does not insert faults into these registers, as such their directly protection is not needed. However, errors can still manifest (as if they occurred in the special registers) in ALU after the special registers are read for processing.

7.2 CFCSS

Unlike the usual implementation, which works over assembly instructions, or LLVM's intermediate representation (IR), our implementation takes advantage of Rust's macro system to insert CFCSS directly into program code. An algorithm originally designed for LLVM IR [8] is used, with modifications to make it suitable for our purposes.

Macro is used to designate a *module* as CFCSS protected, each function within the module being designated as a *block* (b_n). A graph is then generated at compile time representing the allowed control flow within the protected module. Each block within the graph is assigned a random unique signature (s_n) and signature difference (d_n), which is calculated by XORing ($\oplus$) the current block's signature and the signature of its predecessor as seen in equation 1.

$$d_n = s_n \oplus s_{n-1} \tag{1}$$

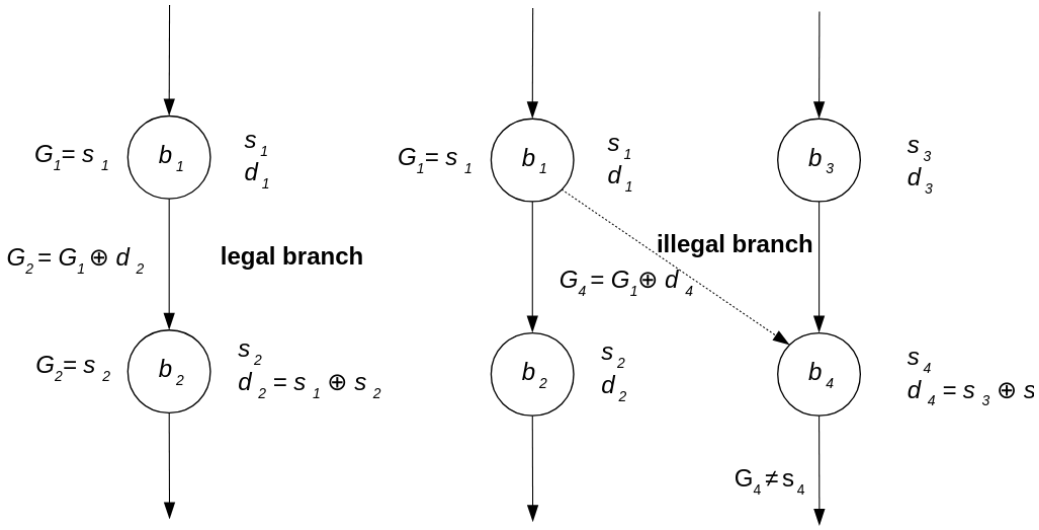

Figure 9: Basic CFCSS [8]

A runtime signature G_n is also created for the given protected module, initially set to $G_0 = 0$. This signature is updated at runtime before the execution of every block by XORing it with the signature difference of the destination block (equation 2). Using XOR to update the runtime signature

is favorable, since XOR operation can be undone by simply being repeated, which is desirable when returning from a block.

$$G_n = G_{n-1} \oplus d_n \tag{2}$$

To ensure correct control flow, the newly calculated runtime signature must be equal to the function signature, otherwise an invalid branching occurred.

$$G_n = s_n \tag{3}$$

This approach is, however, insufficient in the presence of so-called *fan-in* block. If both b_1 and b_3 have an edge to b_4 a problem arises where (provided that $s_1 \neq s_3$) either $G_4 = G_1 \oplus d_4$ is true or $G_4 = G_3 \oplus d_4$.

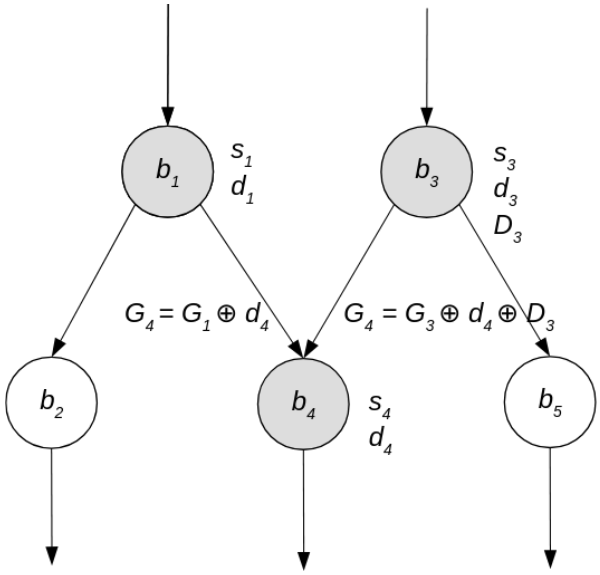

Figure 10: CFCSS with fan-in block [8]

We need to introduce another constant called signature adjuster (D_n) calculated for each predecessor of a fan-in node, and apply it while updating the runtime signature. The adjuster is generated by XORing the signature of the current node (s_n) with the signature of the successor (s_{n+1}). This gives us the actual equation (4) for updating runtime signature.

$$G_n = G_{n-1} \oplus d_n \oplus D_n \tag{4}$$

7.2.1 Macro implementation

The Rust macro generates static variables (G_n, D_n, s_n, d_n) during compile time and inserts additional code before and after the actual function code within the protected module (see figure 11). The instructions inserted after the function block are equivalent to the ones inserted before, with the exception of restoring *last adjuster* from stack instead of updating and storing it.

```
1 RUNTIME_SIGNATURE ^= #dif_name;
2 if is_fan_in {
3     RUNTIME_SIGNATURE ^= RUNTIME_ADJUSTER;
4 }
5 if #sig_name != RUNTIME_SIGNATURE {
6     return_to_checkpoint(ERROR_MSG);
7 }
8 if func_has_adjuster {
9     RUNTIME_ADJUSTER = #adj_name;
10 }
```

Figure 11: Rust - CFCSS instructions inserted before function block

This macro can be applied as shown in figure 12 by specifying a module as *#[cfcss_root]*. An entry function is designated with a special macro *#[entry]*, this macro embeds additional instruction that reset the runtime signature and adjuster of the module, allowing for repeated call of the entry point. An entry function can only be a function that is not called from within another function in the protected module (module being a collection of functions designated as seen in Fig. 12 line 2).

7.2.2 Overhead

Embedding new instruction during compile time will understandably have performance impact on the execution of the program. Each function will incur a execution overhead, as multiple operations and checks must be carried out before the actual function body executes. Approximately 50 additional assembly instructions are inserted before the function block and 50 more after the function block, without using any compiler optimizations.

However, due to the nature of the implementation, the developer is in full control of just how much overhead is incurred. Since the number of

```rust
#[cfcss_root]
pub mod protected_block {
    fn function_0();
    fn function_1();
    #[entry]
    pub fn entry_function() {
        function_0();
        function_1();
    }
}
```

Figure 12: Rust - Using the CFCSS macro

additional instructions is constant per function definition, we can either split our protected module into fewer function to have less overhead, but at the cost of lower fault-tolerance. Or we can fragment our code into as many function blocks as possible, and thereby increasing the number of control flow checks performed.

7.2.3 Considerations

Since our implementation of CFCSS utilizes a purely code based approach it naturally comes with several limitations. Unlike the traditional approach, which implements CFCSS at IR level by extending the LLVM pipeline we do not have access to the underlying instructions, which limits the possible density of the control flow checks. An IR approach is able to precisely insert instructions after every branch and jump operations, and also has more granular control over the actual instructions being inserted. This means our implementation is more prone to control flow errors. Although not within the extent of this thesis, further improvements could be made by directly extending Rust's LLVM pipeline and integrating a CFCSS method at IR level.

Additionally, CFCSS is incompatible with the aforementioned checkpoint and restart system. As explained in section 7.1.5, our checkpoint and restart system does not restore context of static variables, which CFCSS relies on for proper functioning.

7.3 Protection of variables

Various unforeseeable circumstances can lead to the corruption of program variables, from direct corruption of registers storing the variables to indirect side-effect originating from errors within function that work with said variables. This section focuses on various techniques implemented to ensure the reliability of critical program variables with varying degree of protection depending on the use case.

7.3.1 Copy and commit

Copy and commit is a method inspired by a common practice in database transaction handling, where a temporary copy of the variable is created and modified instead of the original value. Upon an explicit commit, the original variable is overwritten with the modified copy. This pattern resembles *libft* [7] variable protection and is particularly useful in combination with checkpoint and restart to avoid unintended side-effects. In the Rust example shown in Figure 13, a CopyCommit wrapper is used to hold a temporary copy of the value. Inside the bar function, the value is incremented, but the change only takes effect on the original data when commit() is called. This allows for greater control over when and how mutations are finalized.

```
fn bar(mut a: CopyCommit<u32>) {
    *a += 1;
    a.commit();
}

fn foo() {
    let mut a = 1;
    bar(CopyCommit::new(&mut a));
}
```

Figure 13: Rust - Copy and commit example

While this method gives us more control over when variables are modified and helps prevent unwanted side-effects, it inherently does not provide any fault tolerance against direct corruption of variables. As such, it should only be reserved for non-critical parts of the program, and or combined with other variable protection techniques.

7.3.2 Multiple redundant variables

Another way we can secure a variable is creating multiple copies of it and applying any modification to all the copies. The modified copies are then compared against each other to detect any mismatch. With two copies of a variable, we can detect an error but cannot correct it, since we cannot tell which one of the copies is faulty - we can use this to trigger backward recovery via the checkpoint system. With three copies, we can detect an error but also perform correction (if at least two copies of the variable are equal) effectively performing forward recovery and avoiding rollback. The more copies we create the more likely we are to have a majority of the variables be equal, therefore the more confident we can be in our fault-tolerance.

```
// Creates 3 identical copies
let mut a = MultiVar::new(0);
// Applies the edit function to each of the copies and
    performs equality check
if let Err(e) = a.edit(|ptr_a| {
    *ptr_a += 1;
}) {
    // Error while updating variable
}
```

Figure 14: Rust - Multiple variable redundancy

In our implementation, a MultiVar wrapper around a variable is used to automatically instantiate three separate copies of the variable as seen in Figure 14. By calling the edit function on the wrapper, we map any modification to all the copies of the variable. The edit function also includes an internal checks which tries to correct the variables in a case of a mismatch. If correction is not possible this function returns an error.

Unfortunately, even if the stored variables are correct, faults can still manifest on the ALU directly while the variable is being read, effectively behaving as if the stored variable was erroneous. Due to this, variable protection alone can never give us complete certainty in any computed result.

7.4 Multiversion

The implemented multiversion techniques are culmination of most of the techniques discussed so far.

7.4.1 Recovery blocks

Recover blocks have been implemented by taking advantage of the checkpoint system. Each call to a version is wrapped in a checkpoint context, ensuring we get back some result, even if incorrect. As seen in Figure 15, if the version returns successfully, we immediately return the value, otherwise we try another version.

```
1 pub fn fib_rec_blocks(x: u8) -> Result<u32, &'static str> {
2     for i in 0..3 {
3         if let Ok(res) = checkpoint(0, || version(i, x)) {
4             return Ok(res);
5         }
6     }
7     Err("Versions exhausted")
8 }
```

Figure 15: Rust - Recovery blocks

The utilized version selection (line 3) is a simple branching logic (Fig. 16) where the higher index version tend to employ stronger protection.

```
1 fn version(v: usize, x: u8) -> Result<u32, &'static str> {
2     let res = match v {
3         0 => fib_v1(x),
4         1 => fib_v2(x),
5         2 => fib_v3(x),
6         _ => return Err("Unknown version"),
7     };
8     Ok(res)
9 }
```

Figure 16: Rust - Version selection

Important thing to mention is the lack of any verification of the result. In the case of recovery blocks, we only care about the function executing without

obvious errors. The return value could be corrupted by an undetectable bit flip somewhere in the function variables. Recovery blocks alone does not deal with this issue.

7.4.2 N-version programming

As opposed to recovery blocks, which only executes one version at a time until one returns correctly, the implemented NVP techniqe executes all available version in sequence. The results from all of the versions are then passed into a voter function (Fig. 17, line 12), which was implemented using Boyer-Moore majority vote algorithm [6].

```
1  pub fn fib_n_version(x: u8) -> Result<u32, &'static str> {
2      let mut results = [0; 3];
3      for i in 0..3 {
4          checkpoint(0, || {
5              let Ok(res) = version(i, x) else {
6                  return Err(checkpoint::Error::Retry);
7              };
8              results[i] = res;
9              Ok(())
10         });
11     }
12     find_majority(&results)
13 }
```

Figure 17: Rust - N-version programming

The used version select is indentical to the one used with recovery blocks and can be seen in Fig. 16.

7.4.3 Versions

The protection used in the individual versions varied widely throughout developement and testing, but as a rule of thumb, the lower index version usually employed fewer, or no, fault tolerance techniques. Primary version usually employed no protection and the alternate versions utilized combination of variable protection via duplication or triplication, repeated calcuation and or complete function rewrites using alternate algorithms.

[6] https://en.wikipedia.org/wiki/Boyer-Moore_majority_vote_algorithm

8 Testing

Testing of the implementation was done on Hardisc[7] (simulated using ModelSim) - a hardened RISC-V core with the ability to statically insert faults during runtime of the program. The form of the fault is a single bit-flip inserted approximately every 2500 CPU cycles. Depending on the duration of the benchmark, up to 100 faults could be inserted per benchmark. For the purpose of narrowing down the scope of the thesis, faults were only inserted in the general purpose registers and the execution pipeline (RFC, ALU, MDU, DP, TP).

Notably, random access memory (RAM) was exempt from FIs during our testing. Should a non-transient error occur in the part of RAM which holds the program instructions there would be no way to properly recover from it using software-only fault tolerance. Usually, RAM and CPU duplication is used, effectively running the same program twice in parallel with set synchronization points. This method is outside of the scope of software implemented fault tolerance. Hardisc also provides hardware-implemented RAM protection which we can take advantage of.

The methods outlined in the implementation section, however, are in theory capable of detecting and recovering from certain faults in RAM. For example, using multiversion programming can provide tolerance even in the case of non-transient error occurring in one of the versions. Due to general unreliability of this approach, however, the testing on RAM FIs was not conducted.

8.1 Benchmarks

A set of benchmarks was selected to evaluate the effectiveness of the implemented fault tolerance techniques. These benchmarks primarily consist of common arithmetic operations and data manipulation tasks representative of typical embedded system workloads.

In addition to standard benchmarks, one unique benchmark - *Nested Function Calls* (NFC) was included. This benchmark involves a series of functions invoking one another in a nested manner, creating intertwined control-flow graph. It was specifically chosen to assess the performance impact of CFCSS, which is particularly sensitive to complex function call

[7] `https://github.com/janomach/the-hardisc`

hierarchies.

Benchmark programs report results in one of the following ways, successful execution (0), unrecoverable error (1) and incorrect result (2) written to the platform specific output register (EXIT_REG - 0x80000004). Any other return status code is considered an unknown error. The reliability of the fault tolerance was measured as a sum of successful executions and correctly reported unrecoverable errors divided by the overall number of executions (25 per benchmark). Each benchmark was given equal amount of time to execute (100 000 CPU cycles), if the time threshold is reached the program ends in a timeout. Notably, a timeout does not necessarily mean the benchmark itself did not execute correctly. It is very common for the FreeRTOS scheduler to be the source of an error leading to a timeout. FreeRTOS can take upwards of 30 000 CPU cycles to initialize the scheduler and being the execution of the first task. During this time window, the program is exceptionally prone to faults resulting in unrecoverable errors, as our implemented fault tolerance approach does not directly modify the FreeRTOS codebase or instructions being executed. As such, in the case of a timeout, the output logs are examined to determine if the benchmark succeeded.

Fib sequence	**Correct**	**Errors Reported**	**Timeouts**
Unprotected	10	5	8
Protected	14	9	3

Table 2: Fibonacci sequence benchmark statistics

As seen in Table 2 the overall fault detection of the protected benchmark went up significantly. The increase in correct results was not exceptionally high, likely due to the fact that errors mostly occurred outside of the variables and registers that would directly alter the computation result. Rather, most faults resulted in errors manifesting in various supporting subroutines. Evidence of this is the nearly doubled number of reported errors when fault tolerance was enabled.

Matrix dot product	**Correct**	**Errors Reported**	**Timeouts**
Unprotected	16	0	9
Protected	18	7	0

Table 3: Matrix dot product benchmark statistics

Similar trend can be observed in the case of Matrix benchmark (see Table 3). The correct results difference is even lower, likely due to the fact that the matrix dot product test requires less CPU instructions, and therefore is a relatively smaller time window for faults to manifest. However, the overall error reporting went up significantly, proving our fault detection methods are good at catching error that would otherwise result in unexpected program termination.

Bubblesort	**Correct**	**Errors Reported**	**Timeouts**
Unprotected	15	2	8
Protected	18	7	0

Table 4: Bubblesort benchmark statistics

Bubblesort benchmark also matches the previous observations. It further confirms that while our implementation only has marginal impact on the correctness of the result, the overall fault detection and ability to avoid timeouts is greatly improved with protection enabled.

8.2 Testing with delayed FI

As previously mentioned, the initial startup of FreeRTOS scheduler took up a significant portion of the program runtime. During initialization, only basic fault tolerance is possible without directly modifying FreeROTS source code. For that reason, we also attempted a single round of testing with unprotected scheduler and delayed FI to give FreeRTOS enough time to initialize, before starting fault insertion. This ensures faults would only occur in the part of program directly under our control.

For this specific test, we also lowered the FI saturation down to one fault every 4000 CPU cycles. To ensure faults would still affect our tests, the benchmark was adjusted to take more time, by increasing the Fibonacci number being computed. We executed the benchmark 100 times, with an equal split between protected and unprotected variants.

Fibonacci (FI Delay)	**Correct**	**Errors Reported**	**Timeouts**
Unprotected	23	23	4
Protected	34	13	5

Table 5: Fibonacci benchmark with delayed FI

The findings from this test can be seen in Fig. 5. The rate of timeout and unexpected program terminations became roughly equivalent, reinforcing our suspicion that timeouts mostly occur as a result of fault manifesting in the scheduler subroutines. The rate of successful executions, however, rose up significantly in the protected benchmark - 47.82% increase over the unprotected benchmark, compared to the increase of approx. 20% in the benchmarks without FI delay. This further solidifies the claim that our fault tolerance methods, when implemented properly and in sufficient amount, can increase the success rate of the program.

Matrix (FI Delay)	**Correct**	**Errors Reported**	**Timeouts**
Unprotected	10	36	4
Protected	28	12	10

Table 6: Matrix benchmark with delayed FI

We also ran the matrix benchmark under the same FI conditions, including an initial delay. Figure 6 shows some interesting results: the protected version produced correct outputs much more often - an impressive 180% improvement over the unprotected one. However, we also saw an unexpected increase in timeouts for the protected version, which did not happen with the other benchmarks.

A likely reason for the timeouts is the added complexity of the protection mechanism in this particular test. For example, if the checkpoint system is set to retry too many times, it could cause the program to take much longer to finish. In some cases, the program might not have crashed - it could have just needed more time to complete its computations. If we allowed a longer execution time, the number of timeouts might have gone down. In future work, we could explore adding software watchdogs that monitor whether the program is actually stuck or still working. This would let us adjust time limits more intelligently, avoiding endless waits while still giving the program a fair chance to finish.

8.3 Performance impact

Performance impact of the implemented fault-tolerance methods was measured without employing any compiler optimizations on a fault-free version of the simulated core. In the presence of faults, the effective performance

could vary unexpected, influenced by the randomness of the FI. As such, there is not much point in measuring performance in the presence of faults.

The test were measured in CPU cycles using the RISC-V *mcycle* and *mcycleh* registers.

Test	**Unprotected (cycles)**	**Protected (cycles)**	**Increase (%)**
Fibonacci	30734	34983	13.82%
Matrix	9871	11133	12.78%
NFC	9889	13126	32.73%
Bubblesort	15228	16773	10.14%
CRC	8278	10614	28.21%

Table 7: Execution time comparison between unprotected and protected tests

As expected, looking at Table 7 we see an increase in CPU cycles it takes to execute the protected version as opposed to the unprotected one. Fibonacci, Matrix and Bubblesort (group one) test only incurred roughly 12% increase, while both NFC and CRC (group two) incurred much more substantial increase. The main difference between these two groups of tests is their flow graph complexity. Group one has linear control flow and mostly consists of 1-4 function call, meaning very little time is spent on CFCSS checking. Group two has more complex control flow resulting in a lot of CFCSS checks.

Performance impact is one of the most important aspects to consider when it comes to software-implemented fault tolerance. During testing, we observed certain complex fault tolerance techniques (such as NVP) to sometimes result in worse outcomes, even compared to completely unprotected alternatives. This is likely the direct result of increased execution time, as the more time it takes the program to execute the higher the chance of faults occuring and manifesting into errors.

However, if performance is not a concern and the sole focus is getting the correct result we can use increased execution times to our advantage. By padding our program with no operation (NOP) instructions we could, in theory, decrease the possibility of error manifesting during the execution of an actual program instruction. This approach is yet to be tested, but if effective could incorporated as an extension of the LLVM pipeline.

8.4 Size impact & compiler optimizations

To evaluate the memory overhead introduced by the fault tolerance mechanisms, we compared the size of the compiled binaries with and without protection under two conditions: with no compiler optimizations and with standard Rust *release profile* optimizations enabled, more details on the used optimizations can be found in the official Rust book[8]. As expected, enabling protection introduces additional instructions and data structures, resulting in larger binary sizes. The impact is more pronounced in the unoptimized build, while optimized builds show a more moderate increase due to compiler optimizations reducing overall code size.

Compiler optimizations?	**Unprotected**	**Protected**	**Size increase**
NO	23164	32847	41.8%
YES	8467	9216	8.84%

Table 8: Binary size increase with and without protection

However, using the compiler optimizations comes with some risks. When examining the performance of select compiler optimized benchmarks, fault tolerance was noticeably lacking. Upon examining the compiled binary, it was determined that the compiler optimizations reorganized the order of some instruction blocks, effectively negating the effects of the implemented fault tolerance. Compiler optimizations make it harder to reason about the final form of the compiled binary, making them possibly unsuitable for fault tolerant software.

[8] `https://doc.rust-lang.org/cargo/reference/profiles.html`

9 Conclusion

The purpose of this thesis was to analyze, implement, and test various software-based fault tolerance methods to improve the reliability on embedded systems.

Several prominent software-only fault tolerance techniques were analyzed, including general single and multi-version programming methods, but also specific techniques such as EDDI, CFCSS and SWIFT. From this analysis, a subset of techniques was selected for implementation based on their practicality, fault coverage, and relevance to embedded systems. These included checkpoint and restart mechanisms, control-flow checking, variable protection strategies, and some forms of multi-version programming.

The selected techniques were implemented in Rust and tested within a FreeRTOS-based environment on a simulated 32-bit RISC-V platform. Faults were injected into selected processor components, including general-purpose registers, control units, and the ALU, to simulate common transient error scenarios. RAM was excluded from fault injection due to the limitations of software-only recovery mechanisms.

Checkpoint and restart was implemented as means to facilitate recovery and act as a building block for multi-version techniques. Checkpoint and restart proved to be one of the most reliable fault tolerance methods as it could be added to any part of the software with ease and it served as good protection against timeouts, ensuring control could always be transferred back to the block from which the checkpoint was created.

CFCSS was implemented to examine Rust's capability to implmenent techniques that are usually only thought of as "instruction level". Rust's macro system was used for this purpose and it proved to be a feasible alternative to what would otherwise have to be implemented as compiler level protection. Due to nature of Rust macros and syntax, however, CFCSS could not be applied to the entire application, but only to designated modules, limiting the coverage. CFCSS also proved to be not entirely compatible with checkpoint and restart due to side-effects in static variables required by CFCSS. Future research could be conducted into the implementation of CFCSS directly into Rust's compiler or LLVM pipeline, as well as adding full compatibility with checkpoint and restart by securing the static variables.

Various multi-version techniques were also tried for select benchmarks where multi-version programming was fasible. Recovery blocks was implemented for Fibonacci sequence benchmark, and various multi-version-

inspired forms of protection were used for the other benchmark. Implementation of multi-version techniques proved relatively easy thanks to Rust's high-level concepts such as generic variables. However, the longer execution time of the mutli-version benchmarks often outweight any benefit, given the increased time window for faults to manifest. Further research needs to be conducted to ascertain the benefits of multi-version programming.

The software-implemented fault tolerance techniques did increase the success rate of our benchmark marginally. They also increased the reporting rate of errors and significantly decreased timeouts stemming from FreeRTOS scheduler. We can say, with reasonable certainty, that the aforementioned methods did contribute positively towards the fault detection and tolerance of the system.

Future work could explore hybrid approaches of combining code-level fault-tolerance techniques with compiler implemented ones. By extending the compiler pipeline of Rust, techniques such as CFCSS or SWIFT could be implemented at IR level and further boost the fault detection and tolerance of the software.

10 Resumé

S rastúcim dopytom po výkonnom softvéri pre zabudované systémy prichádza aj náročná úloha zabezpečiť ich bezchybnú funkčnosť. Zložitosť systémov neustále rastie - či už ide o hardvér alebo softvér - čo znamená aj viac miest, kde sa môžu objaviť chyby.

Chyby môžu byť spôsobené vonkajšími faktormi, napríklad žiarením vo vesmíre, ale aj chybami pri vývoji softvéru. Keďže softvér a hardvér sa stáva čoraz zložitejším, je nepravdepodobné, že sa nám v blízkej budúcnosti podarí vytvoriť úplne bezchybné systémy. Preto je odolnosť voči chybám dôležitou súčasťou návrhu softvéru, najmä v kritických systémoch, kde by zlyhanie mohlo ohroziť ľudské životy. Zároveň však platí, že zvýšená spoľahlivosť často znamená kompromis - napríklad v podobe nižšieho výkonu alebo vyššej náročnosti vývoja.

Tradične sa odolnosť voči chybám riešila hardvérovo, napríklad duplikáciou alebo použitím špeciálnych, odolných komponentov. Tento prístup je však finančne aj časovo veľmi náročný, keďže si vyžaduje výrobu špecializovaného hardvéru. V posledných rokoch sa preto čoraz častejšie využíva lacnejšie riešenie - použitie bežne dostupných súčiastok v kombinácii so softvérovými technikami, ktoré zabezpečujú odolnosť voči chybám.

Táto práca sa zameriava na analýzu rôznych softvérových metód na zvýšenie odolnosti voči chybám. Preskúmame ich výhody a nevýhody a implementujeme praktickú ukážku bežiacu na FreeRTOS a RISC-V jadre, ktorá implementuje vybrané metódy v programovacom jazyku Rust a overí ich účinnosť.

Pre zabezpečenie konzistencie a jednotnosti pojmov používaných v tejto práci sa opierame o terminológiu definovanú v publikácii [3]. Nižšie uvádzame vysvetlenie najdôležitejších a najčastejšie používaných pojmov. Ostatné špecifické termíny budú vysvetlené priamo v texte, keď sa prvýkrát objavia.

Služba označuje funkciu, ktorú systém poskytuje používateľovi. Napríklad: diaľkový teplomer, ktorý zobrazuje správnu teplotu. **Systém** predstavuje všetky súčasti potrebné na poskytovanie tejto služby - teda procesor s vnútornými komponentmi, vstupno-výstupné zariadenia, senzory a príslušný softvér.

Všetko, čo je súčasťou systému, sa nachádza v rámci tzv. **hranice systému**. Ide o pomyselnú čiaru, ktorá oddeľuje náš systém od jeho okolia.

Zlyhanie je situácia, keď systém prestane správne poskytovať službu. Ide o prechod zo stavu správneho fungovania do nesprávneho [3]. Napríklad, ak

prestane fungovať vstupno-výstupné zariadenie, ktoré je kľúčové pre činnosť systému. Zlyhanie je dôsledkom **chyby** (error), čo je odchýlka od správneho stavu systému - napríklad neočakávaná zmena výstupnej hodnoty.

Chyba býva spôsobená **poruchou** (fault), teda skutočnou alebo predpokladanou príčinou chyby [12]. Porucha môže byť dlho neaktívna (latentná), až kým sa neprejaví a nespôsobí chybu [3]. Príkladom môže byť tzv. bit-flip v pamäti spôsobený žiarením.

Odolnosť voči poruchám (angl. *fault tolerance*) je schopnosť systému pokračovať vo svojej činnosti aj v prípade, že sa vyskytne porucha. Napríklad systém môže správne fungovať aj s jedným nefunkčným senzorom. Ak systém nedokáže plne splniť všetky svoje funkcie, ale stále je aspoň čiastočne použiteľný, hovoríme o **degradácii služby** - napríklad ak sa zníži frekvencia meraní alebo sa poskytujú len približné údaje.

Zvládanie chýb a porúch je kľúčovým cieľom tejto práce. V nasledujúcich kapitolách sa pozrieme na konkrétne typy porúch, ktoré sa môžu vyskytnúť, a na metódy, ktoré nám môžu pomôcť zvládnuť ich dopady.

Softvérové techniky zamerané na zvýšenie spoľahlivosti systémov možno rozdeliť do viacerých kategórií na základe ich architektúry a spôsobu implementácie. Jedným zo základných rozdelení je podľa toho, či systém používa jednu alebo viacero verzií kódu na zabezpečenie odolnosti voči chybám. Na tomto základe rozoznávame:

Jedno-verziové metódy (Single-Version)

Tieto metódy sú navrhnuté tak, aby fungovali v rámci jednej verzie softvéru. Ich cieľom je rozpoznať chyby počas behu programu a reagovať na ne vhodným spôsobom, bez potreby redundantného vykonávania viacerých verzií [19].

- **Modularita** - Rozdelenie programu na nezávislé moduly a zabezpečenie, že chyby z jedného modulu nepresihnu do žiadneho iného.
- **Detekcia chýb (Error Detection)** - Monitorovanie behu programu s cieľom odhaliť nezvyčajné správanie alebo neplatné hodnoty, ktoré môžu naznačovať výskyt chyby [19].
- **Obnova po chybe (Fault Recovery)** - Mechanizmy umožňujúce návrat do bezpečného stavu, ako napríklad kontrolné body (check-

points) a návrat do známeho funkčného stavu. Delia sa na *backward recovery* a *forward recovery* [12].

Viac-verziové metódy (Multi-Version)

Tieto metódy využívajú viaceré nezávislé verzie softvéru alebo algoritmov, ktoré vykonávajú rovnakú úlohu. Porovnávaním výsledkov týchto verzií je možné odhaliť chyby a zabezpečiť korektné výstupy [19].

- **Recovery Blocks** [21] - Používajú hlavnú verziu kódu spolu s alternatívnymi blokmi, ktoré sa vykonajú, ak hlavná verzia zlyhá pri vopred definovanej kontrole správnosti (acceptance test).
- **N-Version Programming** [4] - Viacero verzií programu je vyvíjaných nezávisle a spúšťaných paralelne. Výsledky sú porovnávané a systém sa riadi podľa väčšinového rozhodnutia.
- **N Self-checking Programming** [11] - Variácia NVP s pridanými obnovovacími a akceptačnými technikami v každej verzii.

Dátová diverzita (Data Diversity)

Na rozdiel od kódovej redundancie sa dátová diverzita zameriava na zmenu vstupných dát rôznymi spôsobmi (napr. reexpresia), čím sa znižuje pravdepodobnosť, že chyba ovplyvní všetky varianty rovnakým spôsobom. Táto technika je často využívaná ako doplnok k iným formám redundancie [9].

Predchádzajúce práce a inšpirácie

V tejto práci boli analyzované a čiastočne implementované aj niektoré známe softvérové techniky z predchádzajúceho výskumu:

- **EDDI (Error Detection by Duplicated Instructions)** [14] - Softvérová metóda založená na duplikácii inštrukcií a porovnávaní výsledkov na detekciu chýb v behu programu.
- **CFCSS (Control Flow Checking by Software Signatures)** [13] - Technika overovania správnosti tokov riadenia pomocou generovania a porovnávania podpisov pre jednotlivé časti kódu.

- **SWIFT (Software-Implemented Fault Tolerance)**[16] - Pokročilý prístup kombinujúci duplikáciu inštrukcií, kontrolu tokov riadenia a optimalizácie zamerané na zníženie režijných nákladov.

Tieto metódy tvoria základ pre návrh a výber konkrétnych prístupov, ktoré boli implementované a testované v praktickej časti tejto práce.

Programovacý jazyk Rust

Ako už bolo spomínané, implementácia bola uskutočnená pomocou programovacieho jazyka Rust. Rust je moderný jazyk, ktorý bol v tejto práci zvolený pre implementáciu odolnosti voči chybám kvôli svojim jedinečným vlastnostiam. Je navrhnutý s dôrazom na bezpečnosť už pri preklade kódu - napríklad pomocou tzv. *ownership and borrowing* modelu [5], ktorý zabraňuje typickým chybám ako dereferencovanie neplatného ukazovateľa alebo súbežný prístup k premennej. Okrem toho ponúka robustný systém spracovania chýb, kde každá funkcia môže vracať informáciu o chybe ako hodnotu, čím umožňuje jednoduchú detekciu zlyhaní bez nutnosti špeciálnych nástrojov. Rust tiež disponuje pokročilým makro systémom, ktorý umožňuje generovanie a úpravu kódu pred samotnou kompiláciou - čo sa v tejto práci využilo pri implementácii techniky CFCSS. Zároveň ponúka nízkoúrovňový prístup porovnateľný s jazykom C a jednoduchú integráciu s existujúcimi C knižnicami, čo umožnilo prepojenie Rust kódu s FreeRTOS, ktorý je napísaný práve v C. Vďaka týmto vlastnostiam je Rust silným kandidátom pre vývoj spoľahlivého softvéru v zabudovaných systémoch. Viac o Rust-e sa dá nájsť v oficiálnej dokumentácii[9].

Prostredie

Táto práca bola vyvíjaná pre 32-bitové RISC-V jadro bežiace v simulovanom prostredí s 1MB pamäťou RAM, z ktorej bolo 8KB vyhradených pre zásobník a ďalších 8KB pre haldu. Systém využíva jadro FreeRTOS, ktoré je vhodné pre prostredia s obmedzenými zdrojmi. FreeRTOS spravuje spúšťanie úloh (tasks) pomocou plánovača a hardvérového prerušenia typu *Machine Timer Interrupt*. Na systémové operácie sa používajú softvérové výnimky spúšťané cez inštrukciu `ecall`. Všetky prerušenia sú obsluhované pomocou

[9] `https://doc.rust-lang.org/book/`

vektorovej tabuľky, ktorá efektívne presmeruje vykonávanie do správnej obslužnej rutiny [20]. Na integráciu Rustu s FreeRTOS sme použili knižnicu `FreeRTOS-Rust`[10], ktorá poskytuje prepojenie s C kódom. Keďže náš simulovaný systém nevyžaduje podporu atómových inštrukcií, ktoré pôvodná knižnica vyžadovala, boli tieto časti nahradené ne-atómovými verziami. Okrem toho sme použili `riscv`[11] knižnicu pre Rust, ktorá zabezpečuje preklad pre RISC-V platformu a poskytuje nástroje na prácu s registrami a obsluhou prerušení. Z dôvodu výkonovej optimalizácie sme odstránili časť kódu, ktorá pred spustením programu nulovala statické premenné (BSS segment), keďže Rust garantuje inicializáciu premenných už pri kompilácii. Tým sa znížilo zaťaženie procesora a obmedzila sa pravdepodobnosť výskytu chýb počas štartu programu.

Implementácia

Jednou z hlavných súčastí navrhnutého systému odolnosti voči chybám je mechanizmus *checkpoint and restart*, ktorý umožňuje obnoviť program do predchádzajúceho stabilného stavu po zistení chyby. Funguje tak, že si počas vykonávania programu uloží aktuálny stav registrov (x1 až x31) a návratovú adresu do špeciálnej pamäťovej oblasti, čím vytvorí tzv. checkpoint. Po detekcii chyby - či už prostredníctvom výnimky procesora alebo softvérového overenia - sa systém vráti späť na tento bod a pokúsi sa znova vykonať daný úsek kódu. Implementácia využíva statickú pamäť pre posledný checkpoint a zásobník pre staršie, čo umožňuje vnorené checkpointy. Výhodou tohto riešenia je nízka režijná záťaž (približne 400 inštrukcií bez optimalizácií) a flexibilita pri kombinácii s inými technikami. Nevýhodou však je, že systém neukladá dáta mimo registrov, teda napríklad zásobník, heap alebo globálne premenné. Ak sa v checkpoint blokoch vykonajú zmeny mimo sféru zotavenia, tieto sa pri návrate späť nevrátia do pôvodného stavu. Preto je vhodné tento mechanizmus kombinovať s doplnkovými technikami ochrany premenných (viď sekcia 7.3). Celkovo checkpoint systém slúži ako základná infraštruktúra pre budovanie robustnejších stratégií zotavenia.

Implementácia CFCSS v tejto práci využíva Rust makrá namiesto klasického prístupu na úrovni inštrukcií alebo LLVM IR. Každá funkcia v chránenom module je považovaná za blok s jedinečným podpisom, pričom kontrolný graf

[10] `https://github.com/lobaro/FreeRTOS-rust`
[11] `https://github.com/rust-embedded/riscv`

medzi blokmi sa vytvára počas prekladu. Na základe tohto grafu sa počas behu sleduje tzv. runtime podpis, ktorý sa aktualizuje pri každom skoku podľa rozdielu medzi blokmi. Ak runtime podpis nesúhlasí s očakávaným podpisom bloku, deteguje sa chyba v riadení toku programu. Problémom sú tzv. fan-in bloky (viacero predchodcov), pre ktoré sa zavádzajú tzv. adjustéry podpisov. Implementácia v Ruste generuje všetky potrebné kontrolné premenne počas prekladu a vkladá inštrukcie pred aj za každý blok. Približne 100 dodatočných inštrukcií je vložených na každú funkciu, čo môže ovplyvniť výkon. Výhodou je však kontrola nad rozsahom ochrany - čím viac funkcií, tým vyššia miera kontroly, ale aj vyššia režijná záťaž. Táto verzia CFCSS však nie je taká presná ako implementácia na úrovni LLVM IR, kde je možné presnejšie vkladať kontrolné inštrukcie po každom skoku. Navyše, CFCSS nie je kompatibilný s mechanizmom checkpoint and restart, pretože závisí na statických premenných, ktoré nie sú obnovované pri návrate na checkpoint, čím by došlo k narušeniu kontrolného mechanizmu.

Testovanie

Testovanie implementovaných techník bolo vykonané pomocou nástroja Hardisc[12] - simulátora odolného RISC-V jadra so schopnosťou vkladať poruchy vo forme bit-flipov počas behu programu. Po úvodnom oneskorení 36 000 cyklov (na inicializáciu FreeRTOS) boli chyby vkladané každých približne 2500 cyklov, pričom sa mohlo v rámci jedného benchmarku objaviť až 100 porúch. Poruchy boli vkladané iba do registrov a častí vykonávacej pipeline (GPR, ALU, RFC, MDU, DP, TP), pamäť RAM bola z testovania vynechaná, keďže softvérové mechanizmy nie sú schopné opraviť trvalé chyby v inštrukciách uložených v RAM. Testovanie prebiehalo na piatich benchmarkoch, z ktorých každý reprezentoval bežné operácie v embedded systémoch (napr. výpočty, triedenie), pričom NFC (Nested Function Calls) bol špeciálne zvolený na testovanie techniky CFCSS. Výsledky ukázali, že zapnutá ochrana výrazne zvýšila schopnosť detekovať chyby a minimalizovala počet neobslúžených chýb a timeoutov, aj keď zlepšenie v správnosti výpočtov bolo mierne. Merania výkonu prebehli na bezchybnom jadre a ukázali nárast počtu CPU cyklov o 10-33 %, pričom vyšší nárast bol zaznamenaný pri benchmarkoch s komplexnejšou štruktúrou riadenia programu. Nakoniec sa hodnotil aj dopad na veľkosť binárneho súboru - v neoptimalizovanej verzii narastol o vyše 40 %,

[12] `https://github.com/janomach/the-hardisc`

zatiaľ čo pri použití kompilátorových optimalizácií len o 9 %. Avšak optimalizácie môžu narušiť správne fungovanie ochranných mechanizmov, keďže môžu zmeniť poradie inštrukcií. Preto je ich použitie v kontexte odolného softvéru potrebné starostlivo zvážiť.

Zhodnotenie

Cieľom tejto práce bolo analyzovať, implementovať a testovať rôzne softvérové metódy tolerancie na chyby s cieľom zlepšiť spoľahlivosť embedded systémov. Po analýze niekoľkých techník, ako sú generalizované metódy jedného a viacerých verzií, CFCSS, EDDI a SWIFT, boli vybrané na implementáciu techniky ako checkpoint and restart mechanizmy, overenie integrity riadiaceho toku, stratégie ochrany premenných a multiverzné programovanie. Tieto metódy boli implementované v Rust-e a testované na platforme 32-bitového RISC-V simulátora v prostredí FreeRTOS.

Implementácia checkpoint and restart sa ukázala ako veľmi spoľahlivá a efektívna na obnovu po chybách. CFCSS, realizovaný pomocou Rust makier, bol úspešný, no limitovaný len na určité moduly. Vyskytli sa problémy s kompatibilitou CFCSS a checkpoint and restart kvôli statickým premenným. Multi-verzné techniky boli implementované, ale často spôsobovali vyššie časové nároky bez výrazného zlepšenia.

Záverom, softvérové techniky tolerancie na chyby zlepšili úspešnosť testov, znížili počet timeoutov a zvýšili detekciu chýb. Dá sa teda povedať, že implementované softvérové zabezpečenie má pozitívny vplyv na korektnosť programu. Budúci výskum by mohol zahŕňať hybridné prístupy kombinujúce techniky softvérového zabezpečenia implementovaného v kóde, so zabezpečením implementovaným priamo do Rust kompilátoru alebo LLVM.

References

[1] A Aljarbouh. "Selection of the optimal set of versions of N-version software using the ant colony optimization". In: *Journal of Physics: Conference Series* 2094.3 (Nov. 2021), p. 032026. DOI: 10.1088/1742-6596/2094/3/032026. URL: https://dx.doi.org/10.1088/1742-6596/2094/3/032026.

[2] P.E. Ammann and J.C. Knight. "Data diversity: an approach to software fault tolerance". In: *IEEE Transactions on Computers* 37.4 (1988), pp. 418–425. DOI: 10.1109/12.2185.

[3] A. Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[4] Algirdas Avizienis. "The Methodology of N-Version Programming". In: *Software Fault Tolerance*. Ed. by Michael R. Lyu. John Wiley and Sons, 2005. Chap. 2. URL: https://www.cse.cuhk.edu.hk/~lyu/book/sft/pdf/chap2.pdf.

[5] The Rust Foundation. URL: https://doc.rust-lang.org/1.8.0/book/ownership.html.

[6] The Rust Foundation. URL: https://doc.rust-lang.org/book/ch09-00-error-handling.html.

[7] YENNUN HUANG and CHANDRA KINTALA. "Software Fault Tolerance in the Application Layer". In: *Software Fault Tolerance*. Ed. by Michael R. Lyu. John Wiley and Sons, 2005. Chap. 10. URL: https://www.cse.cuhk.edu.hk/~lyu/book/sft/pdf/chap10.pdf.

[8] Benjamin James Jeffrey Goeders. URL: https://coast-compiler.readthedocs.io/en/latest/cfcss.html.

[9] John C. Knight. *Software fault tolerance using data diversity*. Work of the US Gov. Public Use Permitted. 1991. URL: https://ntrs.nasa.gov/citations/19910016332.

[10] J.H. Lala and L.S. Alger. "Hardware and software fault tolerance: a unified architectural approach". In: *[1988] The Eighteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*. 1988, pp. 240–245. DOI: 10.1109/FTCS.1988.5326.

[11] Jean-Claude Laprie et al. “Definition and Analysis of Hardware and Software-Fault-Tolerant Architectures”. In: *Computer* 23 (Aug. 1990), pp. 39–51. DOI: 10.1109/2.56851.

[12] Shubu Mukherjee, ed. *ARCHITECTURE DESIGN FOR SOFT ERRORS*. Elsevier, 2008. URL: http://ndl.ethernet.edu.et/bitstream/123456789/8181/1/166.pdf.

[13] N. Oh, P.P. Shirvani, and E.J. McCluskey. “Control-flow checking by software signatures”. In: *IEEE Transactions on Reliability* 51.1 (2002), pp. 111–122. DOI: 10.1109/24.994926.

[14] N. Oh, P.P. Shirvani, and E.J. McCluskey. “Error detection by duplicated instructions in super-scalar processors”. In: *IEEE Transactions on Reliability* 51.1 (2002), pp. 63–75. DOI: 10.1109/24.994913.

[15] Lorraine Prokop. *Historical Aerospace Software Errors Categorized to Influence Fault Tolerance*. Work of the US Gov. Public Use Permitted. 2023. URL: https://ntrs.nasa.gov/citations/20230001295.

[16] G.A. Reis et al. “SWIFT: software implemented fault tolerance”. In: *International Symposium on Code Generation and Optimization*. 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34.

[17] Kaushik Roy. “Approximate computing for energy-efficient error-resilient multimedia systems”. In: *2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. 2013, pp. 5–6. DOI: 10.1109/DDECS.2013.6549776.

[18] Schuette and Shen. “Processor Control Flow Monitoring Using Signatured Instruction Streams”. In: *IEEE Transactions on Computers* C-36.3 (1987), pp. 264–276. DOI: 10.1109/TC.1987.1676899.

[19] Wilfredo Torres-Pomales. *Software Fault Tolerance: A Tutorial*. Work of the US Gov. Public Use Permitted. 2000. URL: https://ntrs.nasa.gov/citations/20000120144.

[20] Andrew Waterman and RISC-V Foundation Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. 2019. URL: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view.

[21] John Wiley and Sons. *Software Fault Tolerance*. Ed. by Michael R. Lyu. 2005. URL: https://www.cse.cuhk.edu.hk/~lyu/book/sft/.

[22] Jie Xu and B. Randell. “Software fault tolerance: t/(n-1)-variant programming”. In: *IEEE Transactions on Reliability* 46.1 (1997), pp. 60–68. DOI: 10.1109/24.589928.