Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

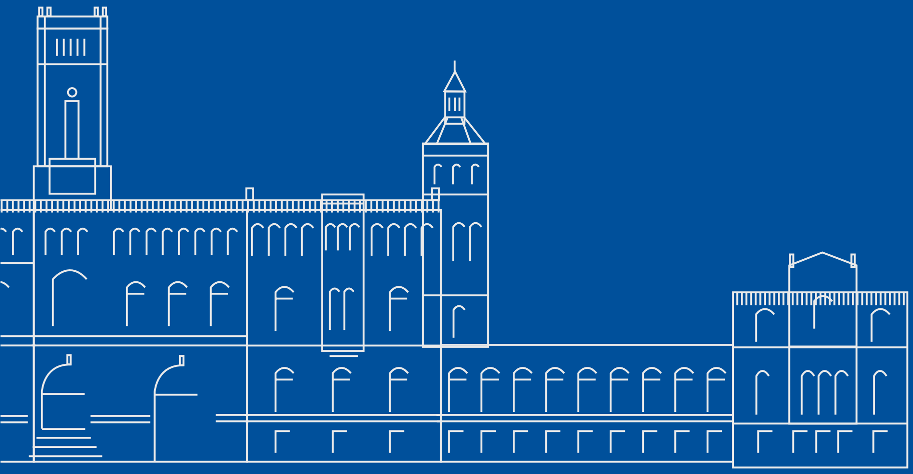
Malte Bargholz

Quantifying Soft-Error Resilience of Embedded RISC-V Systems with Capability-based Memory Protection

Master Thesis in Computer Science

30. Oktober 2020

Please cite as:
Malte Bargholz, “Quantifying Soft-Error Resilience of Embedded RISC-V Systems with Capability-based Memory Protection” Master Thesis, Leibniz Universität Hannover, Institut für Systems Engineering, October 2020.



Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

www.sra.uni-hannover.de

Quantifying Soft-Error Resilience of Embedded RISC-V Systems with Capability-based Memory Protection

Master Thesis in Computer Science

vorgelegt von

Malte Bargholz

geb. am 23. April 1995
in Dannenberg (Elbe)

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer:	**Prof. Dr.-Ing. Bernardo Wagner**
Betreuer:	**Dr.-Ing. Christian Dietrich**
Beginn der Arbeit:	**01. Mai 2020**
Abgabe der Arbeit:	**01. November 2020**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Malte Bargholz)
Hannover, 30. Oktober 2020

ABSTRACT

Shrinking structure sizes and decreasing supply voltages exacerbate the number of transient faults which a device experience during its lifetime. These transient misbehaviors often lead to extensive and expensive failures. In consequence, a multitude of protection schemes has been developed to address the reduced resilience. Most of these incur large overheads, either in hardware or runtime, and are prohibitively expensive for commodity electronics. To avoid these overheads designers often look for added soft-error protection through existing protection schemes such as memory protection. The CHERI protection model protects a system's memory accesses through the use of enhanced pointers called capabilities. To evaluate the impact of CHERI's memory protection on the soft-error resilience of a system, two architectures, one unprotected, and one protected with CHERI, are compared. Their respective soft-error resilience is approximated using fault injection for several workloads. In summary, CHERI reduces the amount of silent data corruption and timeouts which a system experiences. Additionally, it improves the detection of soft-errors by detecting more errors and detecting them faster. Therefore, capability-based memory protection offers effective protection against soft-errors even though it was not explicitly designed for it.

KURZFASSUNG

Schrinkende Strukturgrößen und geringere Versorgungsspannungen verstärken die Menge an transienten Fehlern, welches ein Gerät während seiner Lebenszeit erfährt stark. Dieses kurzzeitige Fehlverhalten führt oft zu weitreichenden und schwerwiegenden Ausfällen. Aus diesem Grund wurden viele verschiedene Schutzmaßnahmen entwickelt, welche die Zuverlässigkeit des Systems erhöhen. Diese Maßnahmen erhöhen jedoch häufig den Aufwand oder die Laufzeit des Systems und sind daher oft zu teuer für Alltagselektronik. Um den erhöhten Aufwand zu vermeiden, wird oft versucht existieren Schutzmaßnahmen, wie zum Beispiel Speicherschutz, für die Zuverlässigkeitserhöhung wiederzuverwenden. Das CHERI Schutzmodell schützt den Speicher eines Systems in dem es erweiteren Zeiger, die man auch Capabilities, nennt verwendet. Um den Einfluss dieses Schutzmodells auf die Zuverlässigkeit eines Systems zu ergründen vergleicht eine durch CHERI geschütze Architektur und eine ungeschützte Architektur. Die Zuverlässigkeit der verglichenen Architekturen wird hierbei durch Fehlerinjektion verglichen. Zusammenfassend verringert CHERI die Menge an undetektierten Datenkorruptionen und Timeout-Fehler eines Systems. Weiterhin führt CHERI auch zu einer besseren Erkennung von transienten Fehler. Diese werden nicht nur häufiger erkannt, sondern im Durchschnitt auch wesentlich schneller. Aus diesem Grund erhöht das CHERI Speicherschutzmodell effektiv die Widerstandsfähigkeit eines Systems ohne spezifisch dafür ausgelegt zu sein.

CONTENTS

Abstract **v**

Kurzfassung **vii**

1 Introduction **1**

2 Fundamentals **3**
- 2.1 A Brief History of Soft-Errors . . . 3
  - 2.1.1 Technology Scaling Effects . . . 4
- 2.2 Faults, Errors, Failures and Resilience . . . 6
  - 2.2.1 Faults, Errors and Failures . . . 6
  - 2.2.2 Dependability and Resilience . . . 7
- 2.3 Fault Forecasting Through Fault Injection . . . 8
- 2.4 Related Work: Protection Schemes . . . 10
  - 2.4.1 Architectural Level . . . 11
  - 2.4.2 Software Level . . . 12
  - 2.4.3 Summary . . . 13
- 2.5 Memory Protection . . . 14
  - 2.5.1 Basic Definitions . . . 14
    - 2.5.1.1 Sandboxing . . . 14
    - 2.5.1.2 Memory Safety . . . 15
  - 2.5.2 The CHERI Protection Model . . . 17
    - 2.5.2.1 Capabilities . . . 17
    - 2.5.2.2 The Capability Hardware Enhanced RISC Instructions (CHERI) hardware architecture . . . 20
    - 2.5.2.3 The CHERI software architecture . . . 21
- 2.6 Summary . . . 23

3 Architecture **25**
- 3.1 Soft-Error Resilience Through Memory Protection . . . 25
- 3.2 The FAIL* Framework . . . 27
- 3.3 CHERI-FAIL: Combining Fault Injection with Memory Protection . . . 30
  - 3.3.1 CHERI-FAIL: Challenges . . . 31
  - 3.3.2 CHERI-FAIL: Extensions . . . 32
    - 3.3.2.1 A virtual fault space . . . 32
    - 3.3.2.2 A bitwise fault space . . . 35

    - 3.3.3 CHERI-FAIL: Integration 38
  - 3.4 Summary 41

4 Analysis **43**
  - 4.1 Fault Model 43
  - 4.2 Benchmarks 44
    - 4.2.1 The fibonacci Benchmark 44
    - 4.2.2 The bubblesort Benchmark 44
    - 4.2.3 Variants 45
  - 4.3 Evaluation Procedure 46
  - 4.4 Results 47
    - 4.4.1 Hypothesis: Reduced Frequency of Unsignaled Content Failures 47
      - 4.4.1.1 Effect of Compiler optimization 48
      - 4.4.1.2 Effect of Instruction Length and memory padding 49
      - 4.4.1.3 Effects of parity-protected capabilities 51
      - 4.4.1.4 Summary 52
    - 4.4.2 Hypothesis: Reduced Frequency of Late Timing Failures 52
      - 4.4.2.1 Effect of optimization 53
      - 4.4.2.2 Effect of instruction length and memory padding 54
      - 4.4.2.3 Effects of parity 55
      - 4.4.2.4 Summary 56
    - 4.4.3 Hypothesis: Improved Detection of Existing Failure Modes 56
      - 4.4.3.1 *TRAP* errors 57
      - 4.4.3.2 Detection Latency 59
      - 4.4.3.3 Summary 60
    - 4.4.4 Summary 60

5 Conclusion **63**

Lists **65**
  - List of Acronyms 65
  - List of Figures 67
  - List of Tables 69
  - List of Listings 71
  - List of Algorithms 73
  - Bibliography 75

INTRODUCTION

1

Soft-errors in integrated circuits have been known to exist for at least the last 30 years [MW79]. They are transient misbehaviors which themselves are manifestations of seemingly random, non-recurring transient currents called faults. As such, they can trigger flip-flops, force a logic transition, or even create permanent feedback loops in the erroneous device [Lay+98]. This low-level corruption can further propagate, for example, if the erroneous flip-flop belongs to a cell of dynamic memory, which is read by an executed program. The program which blindly uses the corrupted values then computes an incorrect result. Untreated and unaccounted for, these soft-errors, therefore, often lead to extensive and expensive system failures [Lyo00].

As the cause of soft-errors researchers first suspected high-energy particle radiation, which only affects devices operating in highly hazardous environments such as satellites in low-earth orbit [BSH75]. However, today it is clear that multiple ionization mechanisms involving different types of radiation exist and can induce the transient current required for soft-errors. Hence, even most commodity electronic devices operating at sea-level will experience multiple errors during their lifecycle.

Additionally, the need for faster and more power-efficient devices has lead to significant decreases in operating voltage, structure size, and increases in operating frequency. All of these changes increase the vulnerability of integrated circuits to soft-errors as the amount of current or charge required to upset the circuit shrinks. For some circuits, the soft-error rate is even the limiting factor for further voltage shrinking [Nar+18].

Over the years, protection against transient errors has been developed at almost any system abstraction level. From special radiation-resistant coatings for semiconductor crystals [CZ85], to redundant logic elements [LV62], error-correcting codes [BB84], and redundant operating systems [DH12], a multitude of protection schemes exist today. Protection at the lower abstraction levels often offer better protection but are significantly more expensive due to additional process steps or multi-redundant hardware. Especially for low-cost devices, this cost is often prohibitively high, which leads to an increased usage of high-level protection approaches.

Memory protection is a high-level protection approach. However, it does not protect a system against soft-errors but instead protects it from invalid access to its dynamic memory. If a program accesses an address for which it has no access rights, the system is notified and can prohibit the invalid access. One form of memory protection is capability-based memory protection. Capabilities are augmented pointers, which in addition to the pointer value include metadata which describes the pointed-to region of memory, for example, its bounds and access rights to it. In a capability-based memory protection scheme, they replace normal pointers and their metadata is validated during each access through them to assert the access's validity.

In addition to protection, these capabilities also provide a form of redundancy. If part of the capability, for example, the pointer value becomes corrupted, they may now be at odds with the additional metadata. This mismatch is subsequently detected by the memory protection system. Most other soft-error protection schemes, such as triple-modular-redundancy, also use redundancy to provide resilience to soft-errors. It can be assumed that the redundancy provided by the capability memory-protection provides similar resilience.

This thesis quantifies the soft-error protection provided by capability-based memory protection. It compares the resilience to soft-errors of two systems: One with capability-based memory protection and one system without any memory protection. More specifically, the CHERI memory-protection model is evaluated by comparing its RISC-V implementation to the unprotected RISC-V architecture. To gauge the additional resilience provided by the memory-protection each system is evaluated by injecting artificial faults into them while they run a suite of micro-benchmarks and recording the resulting soft-errors or system failures.

The rest of this thesis is structured as follows:
Chapter 2 provides a theoretical framework for soft-errors, discusses existing protection schemes, and finally presents the CHERI protection model. Then, Chapter 3 presents, at a high-level, the development of a test framework, which allows gauging CHERI RISC-V's and RISC-V's resilience to faults and their resulting soft-errors for a given workload. Finally, Chapter 4 evaluates both architectures with multiple representative benchmarks to reach a conclusion about their overall resilience to soft-errors in Chapter 5.

FUNDAMENTALS

2

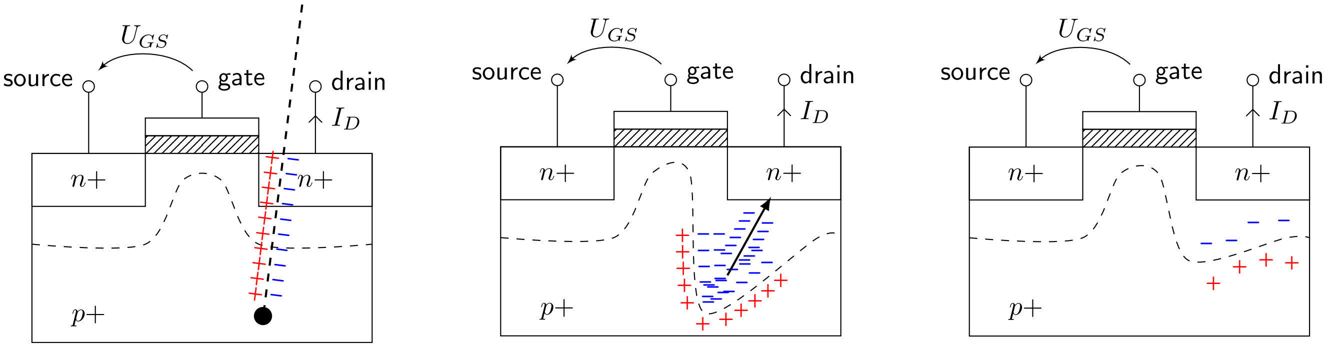
The following chapter gives the reader an overview of the fundamental concepts required to understand this thesis. At first, Section 2.1 touches on the history of soft-errors and their mechanisms. Section 2.2 defines important terms such as fault, error, and failure to arrive at a clear definition of resilience. Section 2.4 gives an overview of soft-error protection mechanisms at different architectural levels, to embed this work into related research. Finally, Section 2.5 defines and describes different levels of memory protection and the CHERI memory protection model that is evaluated in this thesis in effect on the soft-error resilience.

2.1 A Brief History of Soft-Errors

Soft-errors were first discovered in 1975 when researchers of the Hughes Space Communications Company discovered four “anomalies” that occurred in their communications satellites during their 17-year operating period [BSH75]. *Binder et al.* analysis revealed that these abnormal behaviors could not be attributed to the usual charging of the satellites by solar winds. Instead, they argued that these anomalies originated in erroneously triggered flip-flop circuits. They attributed this behavior to high-energy iron particles of cosmic origin, which charged the base-emitter capacitance of critical transistors causing them to change their state. Due to the low failure rate (about one failure in 4 years) and the fact that such heavy, cosmic particles rays are not able to cross into the earth’s atmosphere, the authors dismissed the problem as minor.

It was not until the 1979 landmark paper by May and Woods of Intel [MW79], that the severity of the soft-error problem was recognized. The Intel 2107-series DRAM was found to exhibit a large amount of seemingly random, nonrecurring, single-bit errors, for which the authors coined the term “soft-errors” due to their transient nature. The abnormal behavior was eventually traced back to radioactive contamination of the ceramic packaging material used in these chips. Based on these findings, the authors proposed a different mechanism – ionization by alpha-particles – to explain the soft-errors. Figure 2.1 gives an overview of the ionization mechanism that leads to soft-errors. May and Woods postulated a critical charge Q_{crit}, that must be generated by any ionization mechanism before a soft-error occurs in a particular integrated circuit. Following this discovery “low-alpha manufacturing”, i.e., the usage of material with stringent alpha particle emission rate guarantees, became a standard in the semiconductor industry.

Revising the theory of *Binder et al.*, Ziegler and Landford of IBM hypothesized in 1979 that if alpha particles can induce soft-errors that any cosmic radiation may have a similar effect, even at sea-level [ZL79]. In particular, they proposed an indirect mechanism, in which high-energy particles, especially neutrons, fragment silicon nuclei. These fragments then induce other radiation through nuclear decay and/or charge, which leads to the observed soft-error. At first, the theory



(a) $0 \leq U_{GS} \leq U_{th}$, $I_D = 0$ **(b)** $0 \leq U_{GS} \leq U_{th}$, $I_D \gg 0$ **(c)** $0 \leq U_{GS} \leq U_{th}$, $I_D \approx 0$

Figure 2.1 – A particle strikes a transistor which gate is floating or below the threshold voltage (a), creating a wave of minority and majority charge carriers in its wake. Subsequently, a depletion funnel is formed which collects most majority charge carriers into the nearest depletion region (b) and subsequently the nearest gate, causing a large current spike. Finally, most generated charge carriers have recombined, but a small diffusion current remains (c).

was treated with skepticism by their peers, citing concerns on the separability of soft-error events induced by cosmic radiation and by other sources [Muk08]. To test their hypothesis, the authors then proposed high-altitude testing, which was supposed to increase the occurrence of such high-energy particles. This is due to the earth's dense atmosphere: Most high-energy particles collide and subsequently scatter, on one of the many atmospheric compounds before ever reaching sea-level. Finally, computer repair logs collected by IBM confirmed the proposed mechanism [Nic10] in 1983, putting neutron-induced soft-errors in the focus of research efforts.

In 1995, *Baumann et al.* of Texas Instruments discovered different indirect, neutron-induced soft-error mechanism, which is triggered by low-energy atmospheric neutrons [Bau+95]. Experiments with high-density DRAMs based on the conventional aluminum-based semiconductor process showed a large soft-error rate, which was eventually traced back to boron compounds used as p-type dopants or insulators in the form of Borophosphosilicateglass (BPSG)es. Certain boron isotopes are especially vulnerable to neutron radiation and emit when subjected to such, charged particles and alpha radiation. This discovery leads to the removal of boron additives from semiconductor manufacturing processes, thus eliminating this particular problem [Muk08].

While theoretical work on the soft-error mechanism is abundant, real-world impact data is hard to come by. One highly visible problem case was the "Sun Screen" [Lyo00] bug of 2000, which caused Sun enterprise servers crashed intermittently due to soft-errors in susceptible SRAM caches. Since Sun was the market leader for Unix servers at the time, many high profile companies such as eBay were affected. While other vendors at the time recognized the emerging increased SRAM soft-error rates, Sun failed to account for these in the design of their Unix servers and subsequently lost tens of millions of dollars [Nic10]. In another case in 2005, Hewlett-Packard acknowledged that a large server system installed at Los Alamos National Laboratory was failing frequently because of soft-error events in its parity-protected cache tag array [Mic+05].

2.1.1 Technology Scaling Effects

When soft-errors were first discovered, most semiconductor technologies used a structure size of well over 1 µm. As the customers demanded more functionality with less power consumption and size of electronics, structure density increased dramatically. Moore's law was created which predicted a doubling of transistor count every two years while keeping area constraints the same. To keep up with the increased density, the structure size of modern technology-nodes has decreased to a few nanometers. This change in structure size comes with a problem: The charge needed to upset a transistor cell is directly proportional to the capacitance of its diffusion area, which in turn is proportional to the size of the diffusion area. Thus, a decrease in diffusion area size, or in other words structure size, negatively affects Q_{crit} and increases the soft-error rate. Research as early as 1982 [Pic82], recognized that dense integration posed unique problems for the soft-error rate of integrated circuit. By modeling technology nodes from 4µm to 0.4µm, *Pickel et al.* predicted that soft-error rate would increase with at least $1/\sqrt{K}$, when scaling structure size with a factor K.

Nearly 25 years later, in 2005, studies could not validate this claim [Bau05]. DRAM designs ranging from 1µm to 100nm in structure size instead show a more or less constant soft-error rate when considering the soft-error rate of the whole system. Single bit soft-error rate in DRAM designs even show a 4-5 times reduction for each new technology node. *Baumann* argues that this occurs due to the widespread use of three-dimensional layout techniques in sub-micron DRAM designs, which reduce the vulnerable cross-section significantly. The disparity between whole the system soft-error rate and the single bit soft-error rate can be explained by taking into account the increased density of DRAM designs with each generation. Thus, while DRAM design was originally the most susceptible, it is now one of the most robust devices in terms of soft-error rate.

SRAM designs, on the other hand, show a more or less constant or even decreasing single bit soft-error rate, when considering design ranging below 250 nm structure size. The reduced transistor depletion area cross section has been mostly canceled out by voltage reduction and capacitance reduction, which both decrease Q_{crit}. For larger structure sizes, single bit soft-error was initially increasing, most likely due to the usage BPSG, which has since been eradicated from manufacturing processes. The system-level SRAM soft-error rate is steadily increasing, most likely due to the increased usage of SRAM cells during design. With the advent of new transistor technologies such as Silicon-on-insulator (SOI) or FinFET in technology nodes below 30 nm, a different effect can be observed. While Q_{crit} still decreases with each technology node, Q_{coll}, that is the charge which is collected in a soft-error event, stays constant. Hence, the soft error rate of SRAMs decreases significantly [Nar+18]. This, on the other hand, limits the applicable voltage scaling of modern SRAM cells, as Q_{crit} of these technologies is directly proportional to cell bias. Nonetheless, newer transistor technologies show other failure modes, for example, due to muon radiation [HAR15], and therefore, future developments remain unclear.

Soft-errors, which occur in logic show a tenfold increase in soft-error rate when examining structure size ranging from 180 nm to 90 nm. This may especially be a concern for systems, which protect memory through error correction, i.e., where logic soft-errors are the primary error mechanism. A radiation-induced glitch in logic may only affect the whole system if it propagated from the logic to storage elements (i.e., flip-flops, SRAM, or DRAM). For this, it must occur when the storage element is "latching" or, in other words, clocked. Larger technology nodes are generally driven with lower clock frequencies and higher path delay, which made such a latching glitch exceedingly rare. However, *Baumann* postulated that with increasing frequency and lower path delays that such combinational soft-errors would be latched at an increased rate, and thus make up a significant portion of the whole system soft-error rate. While recent studies confirm the correlation of combinational soft-error rate with frequency [Mah+10], these errors do not make up a significant portion of the system soft-error rate for 32 nm [GSZ09]. Instead, some research even shows a decrease in combinational soft-error rate for designs with a structure size below 20 nm [Mah+14].

2.2 Faults, Errors, Failures and Resilience

The following section provides descriptions and definitions of basic terms relevant to soft-errors, such as faults, errors, failure (Section 2.2.1), and system properties that derive from it, such as dependability or resilience (Section 2.2.2). Note that, while fault, errors, and failures are distinctly defined in this section, the rest of this work uses the term soft-error interchangeably with faults that cause errors and soft-errors, and assumes that a given soft-error always causes a failure and can be observed.

2.2.1 Faults, Errors and Failures

A system that fulfills its intended function (according to a specification) is said to be delivering *correct service* [Avi+04]. If it, on the other hand, deviates from correct service, a system is said to be in failure. Such a *service failure* is the cause of one or more incorrect (internal or external) system (sub-)states called *errors*. Any error is in turn a manifestation of one or more underlying *faults*. Examples of faults include externally induced bit-flips in flip-flops, permanent oxide degradation in transistors, or even software bugs.

Faults can be classified from numerous viewpoints, the most notable of which are: persistence, locality, and dimension. The persistence of a fault may either be permanent, i.e., gate-oxide degradation, or transient, i.e., externally induced bit flips. Additionally, some authors introduce the notion of intermittent faults, which are reoccurring transient faults. The locality of a fault may either be internal, i.e., its cause originates from within the system, or external to the described system boundary. The dimension of a fault may either be hardware, i.e., it originates, and/or affects hardware or software.

Each occurred fault may – at any point in time after its inception – induce an *error*, at which point it is referred to as an *activated fault*. On the other hand, a fault that has not yet caused an error is called a *dormant fault*. Fault dormancy occurs due to a multitude of reasons. Taking a logic circuit as an example, a fault in any of its elements might need a certain set of inputs to propagate to its outputs, which effectively *masks* the fault for certain inputs. A fault can also be masked due to its dependency on the occurrence of a different, but related fault. Furthermore, a fault may be a *tolerated* fault such as a bit flip in an error-corrected SRAM, which is corrected when accessing the faulty cell.

An error is in its inception a manifestation of (one or more) underlying faults [Muk08]. A manifestation need not be directly visible, but it may, in turn, cause other errors which may eventually be visible. This mechanism is called *error propagation*. It can either occur inside a system component (*internal propagation*) or to another component or system (*external propagation*). When an error propagation crosses a system or component boundary it becomes an external fault to the receiving component or system and forms the "chain of threats" as pictured in Figure 2.2. Similar to propagation across component boundaries, the propagation of faults across different abstraction layers can be seen as a form of error propagation. In the context of a processor, a fault and its subsequent failure at the circuit level, e.g., a faulty *AND*-gate, might affect the system at the architectural level, e.g., by saving the wrong result to the result register.

Visible errors are called *detected* errors, if they are indicated by an error signal or message. In contrast, *latent errors* are present but not detected. Additionally, an error might be corrected at any point in the propagation through means of *error correction*. Similarly to faults, errors can be permanent (*hard errors*), or transient called *soft-errors*, which includes errors caused by intermittent faults. While fault and error can be clearly separated, most literature uses these terms interchangeably.

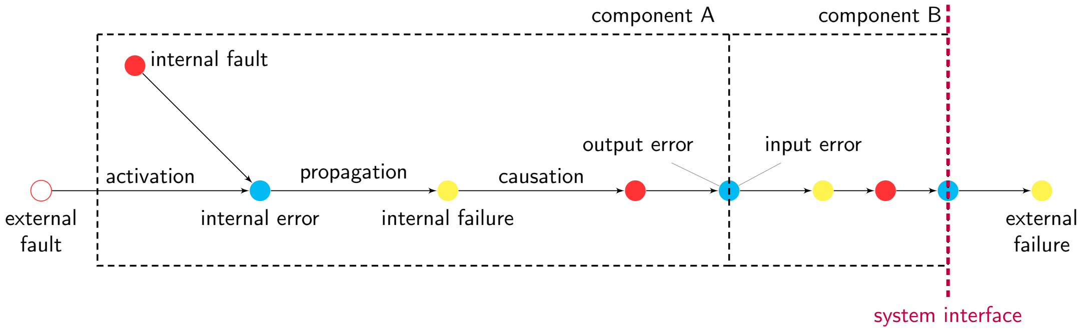


Figure 2.2 – In the "chain of threats" faults activate errors, which propagate to failures, that themselves cause new faults. Errors are propagated across interfaces and can cause failures in the external system behavior or connected components.

This is most likely due to the inherently invisible nature of faults, which only become visible, or manifest, as errors.

A system failure occurs if – and only if – an error manifests at or propagates to an interface at the system boundary and causes deviation from correct service behavior. As a consequence, when talking about errors, one must always consider the scope in which the error occurred. For example, an error in the branch predictor of a processor might propagate along the "chain of threats" until it causes a system failure, i.e., misprediction, but it will never lead to system failure.

The notion of a failure in literature is often used interchangeably with the occurrence of visible errors (e.g., [Muk08]), but when considering resilience to errors it is helpful to separate both. Consequently, system failures can be classified along a failure domain and if applicable, a failure mode.

The domain of a failure, that is the "way" a failure is visible to a user, can be broadly categorized into *content failures*, and *timing failures*. Timing failures can additionally be separated into *early timing failures* and *late timing failures*. Failures that occur when both timing and content of the provided service are incorrect are either *erratic failures* or *halt failures*, i.e., failures which stop any of service of the system. Additionally, each of these failure types can be signaled or unsignaled, which indicates whether the user of the system is able to tell that a failure has occurred previously. This is especially relevant in the context of content failures, which when unsignaled, turn into *silent data corruption failures*. Furthermore, signaled halt failures are often described as *detectable unrecoverable errors*.

Systems, which fail in a consistent way, i.e., don't produce erratic failures, are called *fail-controlled systems*. They only fail in predefined *failure modes* and can be categorized along the most dominant of these modes. *Fail-halt* systems are systems, whose failures are dominated by halt failures. *Fail-silent* systems are systems, whose failures are dominated by silent halt failures. The most interesting failure mode in the context of this work is the *fail-safe* mode, which describes a system that mostly fails with *minor failures*. Such *minor failures* are failures, for which the associated cost of system failures is similar to the benefits of a system that delivers correct service.

2.2.2 Dependability and Resilience

A dependable system is a system, which avoids failures that are more frequent and more severe than acceptable [Avi+04]. Such a system is said to be *resilient* against failures, or the underlying errors and faults respectively. Typically, dependability is split into four to five basic concepts, which must be fulfilled by a dependable system:

1. **Reliability**, i.e., continuity of correct service.
2. **Availability**, i.e., readiness for correct service.
3. **Maintainability**, i.e., the ability to undergo modification and repairs.
4. **Safety**, i.e., the absence of catastrophic consequences of failure.
5. **Integrity**, i.e., the system is free of improper modifications. (optional)

The concepts are often used under the name *RAMS*, to encompass dependable systems. Generally speaking, any extent to which a system possesses resilience or dependability is always probabilistic. Transient faults for example, by their very nature, occur at random and cannot be properly described in a deterministic way.

To attain a resilient or dependable system one usually turns to eradicate their source, i.e., faults. In general four techniques exist: (1) Fault prevention (2) Fault tolerance (3) Fault removal (4) Fault forecasting. *Fault prevention* usually happens at design time. The system is designed in a way that minimizes possible faults, and consequently, minimizes possible failures. *Fault tolerance* on the other hand, happens at runtime, and tries to actively avoid fault induced errors and subsequent failures. It can be split into error detection and recovery. First, a specialized mechanism detects a fault through its manifested error(s), after which a corrective maintenance action tries to remove the underlying fault. A typical example is redundancy: By duplicating part of the system and integrating a special voting mechanism, simple error detection (for dual redundancy) or error correction (for triple redundancy) can be performed. The effectiveness of a fault tolerance mechanism is described by its coverage, which indicates how many faults of a certain class be detected and subsequently corrected through it. *Fault removal* happens at development time and can be split into three parts: Verification, diagnosis, and correction. By verifying a system against its design specification or verification conditions the number of faults may be reduced. Typically, this is done by either static (e.g., model checking) or dynamic (e.g., testing) verification. If a failure is found during verification, the designer must diagnose its underlying fault and finally correct it. Last but not least, *Fault forecasting* evaluates a system's behavior regarding faults. The evaluation can be split into a qualitative evaluation, which identifies failure modes, and a quantitative evaluation, which evaluates in terms of probabilities to which extent failures are possible. Fault forecasting is different from fault removal in that it evaluates the system generally and not with regard to a specification.

For the context of the thesis, I focus on two aspects that increase the resilience of a system. First, a system is more resilient if it shows increased reliability. Such increased reliability could be achieved if the system detects faults earlier or more often and is, therefore, able to correct them in a timely fashion. Secondly, a system is more resilient if it shows increased safety. Such increased safety could be achieved if the system exhibits less catastrophic failure modes or exhibits these failures less frequently. Both these aspects can be evaluated using fault forecasting.

2.3 Fault Forecasting Through Fault Injection

To evaluate a system's overall resilience to transient faults, its failure modes and their corresponding failure rate must be estimated. However, even if failure modes can be successfully modeled, empirical estimation and verification of their corresponding failure rate are infeasible under normal operating conditions [BF93]. Recent studies [Ziv+19] show that even though modern DRAM memory elements exhibit a fault roughly every 3000 hours, this only translates to an uncorrected, i.e., observable, error every $10^1$0 hours. To accelerate system verification, its designers often use a form of *fault forecasting*, to predict possible failures modes and to estimate their corresponding failure rate. One widely known fault forecasting technique is fault injection (FI). Instead of waiting for faults to naturally occur, FI systematically triggers, or in other words *injects*, additional faults in the system under test while it runs one or more workloads to characterize the system's behavior empirically. Specifically, deterministic, run-to-completion workloads are often chosen, because they have a limited and static set of possible points where a fault can occur. While concrete FI implementations differ, most augment an existing target system. A target system, can either be a (modified) physical system or a simulation environment. The FI system assumes complete control over the target system and is able to modify its internal state. Hsueh et al [MTI97] describe a generic FI system with the following components:

- A controller, which orchestrates every step of the fault injection.
- A workload generator, which selects the systems workload and its input.
- A fault injector, which forwards to the system to the injection point and injects the target element.
- A monitor, which observes the workload execution.
- A (optional) data analyzer, which translates the collected data into specific metrics, such as fault coverage.

For the system to produce a valid result, each workload must execute deterministically. This ensures that they have a static set of possible points where a fault can occur. The presented FI system iteratively evaluates this set of *fault points* using a four-step process. First, the target system is reverted to a known, error-less state. Then, after selecting an appropriate workload and the corresponding input, the selected workload is loaded and executed until the target system reaches the fault point. Here, the fault injector injects the faults into the target systems state according to the fault model, e.g., transient single-bit flips, and resumes execution of the target system. Finally, the target system executes the workload until completion or until an exception is encountered, at which point the system state is analyzed and the fault point is considered to be evaluated. This process is repeated until all fault points have been evaluated. Finally, the data analyzer can output relevant metrics, such as the fault coverage for each workload. The FI system as described by Hsueh assumes that only a single fault occurs during each run of the workload. This a reasonable assumption, due to the very low probability of even a single fault. As Schirmeier succinctly puts it in his dissertation [Sch16] :

> "*[...] at current – and tomorrow's – fault rates [...] it is sufficient to inject one fault per FI experiment.*"

Each injection can take place at any given abstraction layer of the system under test. As discussed in Section 2.2.1, due to the error propagation along the "chain-of-threats", a fault in layer n and the resulting failure can and will often cause a new fault in layer $n + 1$, which has a higher abstraction level. Consequently, while the original failure, e.g., a stuck output of an *OR*-gate, is invisible to the user, its propagation in a higher abstraction layer, e.g., a bit flip in a register, is often not. Injecting faults at low abstraction levels provides greater insight into underlying fault mechanisms and allows precise tracing, but is often not feasible. The required computational power and time to model a large and often complex system at this level of detail is too great. Additionally, low-level models of commercial off-the-shelf (COTS) hardware are often not available or no way exist to accurately inject faults at lower abstraction levels [Sch16]. To balance the required simulation effort, FI implementations must carefully choose the abstraction layer at which it injects faults. This fault abstraction implicitly defines a *fault model* of the system, since high-level injection limits possibly faulty elements to these visible at the injected layer. In the context of processors, most FI frameworks, and especially the *FAIL** framework which is used in the context of this thesis, only inject faults at the instruction set architecture (ISA) abstraction level.

For a given injection abstraction level, each possible fault point can be uniquely specified by its location and the time at which it occurs. A fault point's time and location are abstract properties, which are defined in accordance with the target system. To give an example consider an FI experiment, which injects faults into memory elements of a processor. A fault points location in such a system would refer to the memory address and bit injected, while its time would refer to the (absolute or relative) clock cycle during which the injection will take place. Each fault point is located in a two-dimensional *fault space*, which is the space that is defined by all locations and time point combinations during the execution of the target workload. To extract the fault space, a FI experiment will often record a "golden-run", i.e., a fault-less run of the workload, while monitoring accessed elements of the system. The fault space might be non-contiguous, for instance, if the workload does not access certain memory locations. Additionally, a FI experiment might only inject a subset of all fault space points, if certain points are found to be equal in their injection result. This technique is called fault space *pruning*.

2.4 Related Work: Protection Schemes

Research on the influence of memory protection on soft-error resilience is sparse. The *dOSEK* reliable embedded kernel found that enabled memory-protection unit (MPU)-based memory protection halved the experienced unsignaled content failures [Hof+15a] A similar result has also been obtained for the eCos kernel, for which Hoffmann et al. [Hof+14] found that its susceptibility to soft-errors is highly dependent on the enabled hardware protection features. More specifically, enabling both the hardware watchdog and the MPU decreased the percentage of unsignaled content failures by 20 percent. However, no research which evaluates capability-based memory protection in a soft-error context is known to the author. The rest of this section, therefore, gives an overview of existing soft-error protection schemes.

Protection schemes are specific to the type of system that they protect. In this thesis a processor-based system is considered, which next to a load-store RISC processor, of DRAM memory, and a varying amount of memory-mapped I/O devices, the last of which shall not be considered during this thesis. The processor follows the Von-Neumann-architecture, i.e., consists of an execution unit, a control unit, and shared instruction and data memory. The processor state is saved in a register file and the processor might implement a pipelined execution model and caching to accelerate computations.

In total, such a system can be structured into four distinct abstraction layers of which only two are discussed here. The CHERI protection model is a memory protection technique at the architectural- and software level and therefore only memory protection techniques on these two abstraction levels are discussed in the following. At the lowest level of abstraction – the transistor level – a system consists of interconnected transistors and other electrical elements. By abstracting the functionality of transistor groups, logical elements can be derived which are connected in a circuit at the circuit level. Additionally, the system architecture, i.e., their connection to each other, might provide protection against soft-errors. Architectural protection schemes are discussed in Section 2.4.1. Finally, Section 2.4.2 discusses protection schemes, which modify the software that runs on a system.

2.4.1 Architectural Level

The protection at the architectural level can be categorized as follows:

1. Storage, i.e., protection focused on large data structures such as caches or register-files.
2. Computation, i.e., protection that focused on correct arithmetic computation.
3. Control-flow, i.e., protection that focused on the correct execution order of the instruction stream.

Historically, storage-focused approaches were the first to appear in research, most likely due to the increased soft-error rate in the repetitive SRAM or DRAM designs. They rely heavily on a coding/information theory background, to encode the information stored in an efficient but also redundant, or at least easily verifiable, way.

The simplest mitigation strategy – the parity bit – provides single-bit error detection. For each piece of information or byte, a redundant parity bit that is either 1 or 0 is added depending on the number of ones or zeros in the original byte. For example, if the amount of ones in the original byte is even the parity bit is 1 and 0 otherwise. The parity bit is stored in-line with the original data, or in a physically separated parity storage [PGK88]. When reloading data from storage, its parity bit can be calculated again and compared to the stored parity bit to detect any errors. Such a scheme can detect errors up to a hamming distance of 1, but can not correct the error. However, it struggles with Multi-bit upset (MBU), i.e., soft-errors which affect multiple bits in the original data [Kim77; Bar77]. Nonetheless, multiple improvements to the original design, e.g., [Kem80; MJ81], are patented and the parity bits remain in wide use today.

Error-correcting codes (ECCs) are an extension of parity bits. They provide, in addition to detection, correction of soft-errors up to a specified amount of allowed errors. Due to their binary coding nature, their detection and correction capabilities are inherently bounded. For a code word hamming distance of d, they can correct up to n and detect up to $n+1$ errors, iff. $d > 2n+1$ [Li+16]. There are multiple patented ECC schemes, which offer varying amounts of coverage and overhead [BB84; Che89; Del97]. ECC protected systems usually have a timing overhead, which is incurred by the code word loading and checking, in addition to an area overhead incurred by the very same checking circuit. Still, the use of ECC is widespread. Examples include but are not limited to, Reed-Solomon ECC in nearly all consumer media disks (CD, DVD, BluRay) or ECC-RAM, which is the de-facto standard in modern server systems.

Architectural protection, which focuses on computation or control-flow checking is hard to separate. Pure computational approaches rely on the encoding of information, similar to storage protection techniques. One computational protection scheme is arithmetic codes (AN Codes) [Sch10]. AN code is a protection scheme, where for a binary operation each of the operands is multiplied with a constant A before the arithmetic operation is applied. The result of the operation must then be a multiple of A, which can be checked by integer division. If the computation was faulty, e.g., due to a soft-error in the execution unit, the resulting value is most likely not a multiple of A, and the error can be detected. Unfortunately, AN codes do not cover logical operations. Nonetheless, other arithmetic codes, which partially cover logical operations, such as residue codes [Gar66] exist and are used in systems such as the IBM POWER6 processor [San+08]. One other notable computational approach, that does not rely on coding techniques, is the DIVA core [Aus99]. The DIVA core uses partial spatial redundancy. A simple "checker" core runs alongside a complex out-of-order core, that validates both memory accesses and computation of the main core. However, implementations of the DIVA design have proven difficult, due to increased memory and cache pressure [CWA00]. The RESO technique [PF82] instead uses time-redundancy. Each arithmetical operation is calculated twice: Once unmodified and again using shifted versions of the input operands. The results of both operations can be compared and errors can be detected. While spatial-redundancy approaches often only significantly affect the performance of the processor in an error case, time-redundancy approaches introduce a constant overhead (up to 40% [Li+16]) even without any errors.

Hybrid protection schemes, which target both computation and control flow are almost always based on redundancy. One prominent example is the AR-SMT approach [Rot99], which uses the simultaneous multithreading (SMT) feature of modern processors to replicate the instruction stream and provide time-redundancy. At certain points in the execution, each thread's architectural state, i.e., register file, program counter, and caches, is compared to detect soft-errors. This approach can reduce the aforementioned overhead of time-redundancy to 10-30% over a non-duplicated thread in simulations. Spatial redundancy that protects both computation and control flow uses N-modular redundancy and lock stepping. Lock stepping [JSK02] is a mechanism, where the complete processor is duplicated, runs in parallel, and the architectural state of the duplicates are compared cycle-by-cycle. N-modular redundancy incurs an area overhead of N times over the baseline architecture, in addition to any timing overhead of the comparison and synchronization of the duplicated cores. Still, multiple implementations [Woo99; Sle+99] exist. One notable example, which does not use redundancy is the Argus [MBS08] method. Argus is a complete approach, that not only detects computation and control-flow errors, but also data-flow and storage (access) errors. Data-flow and control-flow errors are detected by calculating a signature from the program's control flow graph, which is compared to a precomputed signature. To detect errors in memory (access) Argus uses an ECC inside its caches. Additionally, computational errors are detected through subcheckers for each execution unit.

Protection schemes which target only the control-flow may be: (1) hardware-only, i.e., implemented solely in the processor architecture, (2) software-only, i.e., implemented in COTS architectures by inserting additional instructions, or (3) a hybrid approach, in which the processor implements additional instructions to simplify software-based control-flow checking (CFC) techniques. One hardware-only, that is architectural, the approach is CFCET [RM06]. In CFCET the processor uses execution tracing to compare the executed jumps with a precomputed jump graph, which is generated from the program's source code. If a mismatch is detected, a watchdog resets the processor to the last known good state. While CFC approaches seemed promising at first, recent studies have shown flawed statistical analysis and/or evaluation techniques that lead to the overestimation of these approaches. Instead of providing additional soft-error resilience, they, in fact, increase the vulnerability of the modified architecture by 5% for hardware-based techniques and up to 21% for software-based approaches [RJS19].

2.4.2 Software Level

Protection at the software level can be categorized into three levels:

1. Operating system protection, i.e., protection that targets a program's runtime such as the operating system.

2. Program protection, i.e., protection that targets the executed program, but without considering its algorithm.

3. Algorithm protection, i.e., protection that targets the executed algorithm.

Initially, operating system protection focused on isolation [Hof16]. The isolation concept eventually reached its peak with µ-kernel architecture [Acc+86], in which each operating system service and program run in complete isolation from each other and only communicate over rigid and predefined communication interfaces. Isolation in itself can not protect an operating system from soft-errors but can aid error-detection and hinder error propagation along the “chain-of-threats” (see Section 2.2). In combination with component-based recovery schemes [Dav+07; Dav+19a], such an operating system can successfully detect and recover from a soft-error. PikeOS [Bau+09], which was originally developed in a verification context, bases its protection on a similar isolation-based scheme, in which the trusted hypervisor para-virtualizes multiple isolating operating system instances. The hypervisor itself, however, is explicitly not secured and thus the prime target for soft-errors. L4/Romain [DH12], on the other hand, implements a typical N-modular redundant approach in the well-known L4 µ-kernel. Each thread is executed thrice while proxying any operating system calls to avoid duplicated I/O interaction. Their results are then compared to check for errors.

In recent years, more focus has been put on fault-tolerant operating system design. Artk68-FT [AJJ04] is one such operating system. It uses coding and redundancy techniques to improve its tolerance against soft-errors. Martin Hoffmann focused his dissertation on the constructive dependability of operating systems. He found that a reduction of dynamic state, the usage of direct data structures over indirect counterparts, coding-based protection of critical structures, and system customization through a priori knowledge are the cornerstones of dependable operating systems. The *dOSEK* operating system [Hof+15b] implements these design guidelines and shows improvements to silent data corruption by four orders of magnitude when compared to a COTS operating system with similar features.

Protection at the program level has its origin in the *N-version programming method* [Avi85], in which N (functionally equivalent) versions of the same program are generated from a given specification. The output of each of these programs is then compared to detect soft-errors. This technique is related to the previously discussed spatial and time-redundancy approach of architectural protection. It is in fact, a different kind of redundancy – resource redundancy – in which additional resources, i.e., programmers, are utilized to improve error detection. Today, most protection techniques that target whole programs implement special compilers, which insert additional instructions to protect computation [OSM02b], control-flow [OSM02a] or both [Rei+05]. While all of these show great fault coverage, i.e., they detect faults well, they struggle with a non-negligible performance overhead of $50-200\%$.

Algorithmic protection schemes make use of explicit algorithmic knowledge to develop a fault-tolerant version of the algorithm, which is implementation-independent. *Roy-Chowdhury et al.* [RB96] propose one such algorithm, which protects matrix multiplications by introducing additional data into the matrix that can be checked in the result. Evaluations of the proposed protection scheme have since concluded that this approach introduces an overhead of $10-14\%$, depending on the dimension of the matrix, in addition to the increased code size and algorithmic complexity. Similar algorithms exist for various mathematical problems, such as Fast-Fourier-Transform [RB90] or QR decomposition [JA88].

2.4.3 Summary

In summary, a multitude of protection schemes exist at the architectural and software level of a system. Almost all use some form of redundancy, either in data or computation, and, therefore, induce a large overhead for protected systems. Architectural protection schemes generally fare better in terms of overhead. However, they require potentially expensive and time-extensive extensions of the protected hardware. Software protection schemes, on the other hand, can be adapted by simply extending the program. Sometimes, a fault-tolerant algorithm can even be found, which removes the need for protection schemes altogether. Next, memory protection is discussed, which is a protection scheme for memory access. However, as discussed with fault-tolerant algorithms, it might provide inherent tolerance to soft-errors and can provide a benefits without additional redundancy hardware.

2.5 Memory Protection

The following sections define and describe different levels of memory protection (Section 2.5.1) and present the CHERI memory protection model, a well-known architecture for memory protection that is used for this thesis (Section 2.5.2). Memory protection is used in the context of this thesis to provide a hybrid soft-error resilience scheme, that lies between architectural and software protection.

2.5.1 Basic Definitions

in its most simplistic definition, memory protection describes the control of access to the dynamic memory of a system. In such a system multiple *processes* exist, which have (possibly distinct) access rights associated with them at any given point in their runtime. These access rights are dynamic, i.e., they may change, either by explicit request of the process or by request of the underlying (operating) system. Protection in such a system can take one of two forms: Either sandboxed, that is the virtualized execution of the untrusted process, or execution of an inherently trusted process [Sti12]. In other words, either the executed code is safe-by-design regarding memory access, i.e., *memory-safe*, or the runtime must virtualize its execution to guarantee access control.

2.5.1.1 Sandboxing

The term “sandboxing” originates in a 1993 work of *Wahbe et al.* [Wah+93], but is first defined by *Goldberg et al.* [Gol+96] as:

> We use the term *sandboxing* to describe the concept of confining a helper application to a restricted environment, within which it has free reign.

Instead of providing a process with the complete system environment, a restricted environment is created and subsequently used to execute the process. In addition to memory, such an environment might include other system resources, but for the context of this discussion is it sufficient to assume that a sandboxed process is only restricted in its memory access. This restriction can be transparent for the process, i.e., any invalid access is masked, or non-transparent if the process is notified of the failure when it tries to access restricted resources. Regardless of the transparency, sandboxing can only be detected by the process when accessing invalid memory. Therefore, in the nominal case, it seems to the process as if it had full system access. One consequence of this property is that sandboxing does not protect the process from *semantic errors*, such as out-of-bound array accesses, except where these accesses would affect other processes in the system. Typical implementations of sandboxing often use hardware-based memory isolation techniques, such as virtual memory. Nonetheless, software-based isolation techniques exist [Wah+93] and work through the same basic principle.

Virtual memory is a memory management technique, that provides an idealized abstraction of physical memory addresses to *virtual addresses*. Segmented memory implements such virtual addresses using a combination of segments and offsets [OG65]. A special segment identifier register, which contains the memory segment identifier, is added to the processor or must be passed to each memory access. Therefore, the virtual address consists of a segment identifier and an absolute offset into the segment. The segment identifier is then used, typically by a hardware unit called memory-managment unit (MMU), to query and check the access rights to the requested segment. The MMU stores, in addition to access rights, a base address, and a length, which are checked against the accessed offset. If both tests succeed, the virtual address of the access can be translated to a physical address by adding the offset to the segment base address, and finally, the memory is accessed.

Using segmented memory, a simple protection scheme can be implemented as follows. Each process is assigned a unique segment into which the processes data is loaded. Its corresponding segment identifier is then written to the segment register on process dispatch or continuation. Each process is therefore only able to write its own memory, effectively isolating the processes and implementing a simple memory protection scheme. This concept can be extended, e.g., by splitting code, data, and stack memory of a process into segments with different access permission. Even though today segmented memory is not widely used, virtual memory remains in use, for example as page-based virtual memory or region-based memory protection. Concerning the vulnerability against transient faults, sandboxing methods which rely only on processor internal state registers, e.g., segmented memory, can be protected with less overhead, while sandboxing methods which rely on the large state in volatile memory, e.g., page-based virtual memory, are more difficult to protect [Sti12].

2.5.1.2 Memory Safety

Most commonly, memory safety is a combination of *spatial error safety* and *temporal error safety*. The authors of the MemSafe [SB13] project define these terms as follows:

> A *spatial error* is a violation caused by dereferencing a pointer that refers to an address outside the bounds of its “referent”. Examples include indexing beyond the bounds of an array, dereferencing pointers obtained from invalid pointer arithmetic, and dereferencing uninitialized, NULL or “manufactured” pointers.
>
> A *temporal error* is a violation caused by using a pointer whose referent has been deallocated (e.g., by calling the `free` standard library function) and is no longer a valid memory object.

If a program, or more often its programming language, does not commit spatial or temporal memory errors, then it is *memory safe*. Most modern languages are memory-safe by design, e.g., Ada, C# or OCaml, while most languages which target, or are at least most widely used for, embedded system development, e.g., C or C++, are *not* memory-safe. Often, memory safety is derived by providing strong type-safety at the language level, i.e., each reference represents a strongly typed *capability* to access a memory area in limited ways defined by the type of the referenced data. Such languages often do not allow the creation of arbitrary pointers, as it is required for controlling memory-mapped devices in embedded systems, since the type system would be unable to reason about the referenced data. The strongly-typed programming language Rust [Rus] is one notable exception. It combines both memory-safe and memory-unsafe code with a clearly defined interface between both to allow the type-system to remain sound. Still, memory-unsafe languages (especially C) remain in wide use today, thus substantial research effort has been put into extending their memory safety guarantees. These efforts can be broadly categorized into three abstraction levels (low to high):

1. Hardware-level or instruction-set level, i.e., the memory safety checks are performed by the hardware, which might receive additional information through inserted or modified instructions.

2. Compiler-level, i.e., an intermediate representation of the code is extended or transpired to implement memory-safety.

3. Language-level, i.e., the unsafe language is extended to perform runtime checks, which implement memory-safety.

Low-level approaches generally incur a large overhead but provide broader applicability than high-level approaches, which often target only one specific language or usage case. However, high-level approaches can be significantly faster due to the additional information available at the source code level. Nonetheless, language-level approaches often require non-trivial changes to the source code, for example in the form of annotations, which might lead to non-trivial resources, i.e., programming time, overhead.

HardBound [Dev+08] is a hardware-level processor extension, which provides a new architectural primitive – the bounded pointer – that enables spatial memory safety through hardware/software cooperation. This bounded pointer (often referred to as a "fat-pointer" in other work) leaves the original pointer intact but amends it with additional base and bound information, which are set by the software on pointer creation. The additional information is kept completely invisible and separate from the original pointer and is maintained and propagated by the hardware after its creation. To differentiate a raw pointer, i.e., not-yet-annotated pointer, from its amended counterpart HardBound keeps separate tag storage, which contains a bit for each memory word to distinguish the different types of pointers. An evaluation, which assumes that the hardware extension takes one additional micro-operation per instruction to complete the bound checking and forwarding, shows an average runtime overhead of 10%, which goes up to 23% for some benchmarks, and an average memory overhead of 55%, which goes up to 200% for some benchmarks. The CHERI protection model, which is used in the context of this thesis, uses a similar approach, but significantly expands the software-side of HardBound and can provide additional temporal memory security. It is discussed in Section 2.5.2.

SoftBound [Nag+09] is a compiler-level approach, which applies the HardBound fat-pointer approach as a low-level virtual machine (LLVM) transformation. Instead of explicitly inserting source code level bound instructions, SoftBound, instead, inserts additional instruction at the LLVM intermediate representation level to implement the bounded pointer primitive. Therefore, no changes to the program's source code are required. Its evaluation shows an average runtime overhead of 93% when checking both loads and stores, but 54% if only writes are checked. Checking only writes is an optimization which the authors propose to minimize the induced overhead, while still retaining memory safety in most cases. In addition to the runtime overhead, SoftBound incurs up to 300% (average 87%) overhead. Other optimization techniques, such as categorizing pointers by their usage, as done in the CCured project [NMW02], can decrease overhead. The overhead, nonetheless, remains significant.

Retrofitting unsafe languages with memory safety-by-design is the language-level approach that the authors of Cyclone [Jim+02] propose. Cyclone is a safe dialect of the C programming language, which guarantees memory safety through a combination of intra-procedural analysis, source code annotation, and runtime checks. As a dialect, Cyclone is not directly compatible with existing C source code, but the authors estimate that only around $10 - 20\%$ of existing source lines must be changed to port a typical application. Cyclone incurs an overhead of up to 242% when compared to the baseline C implementation.

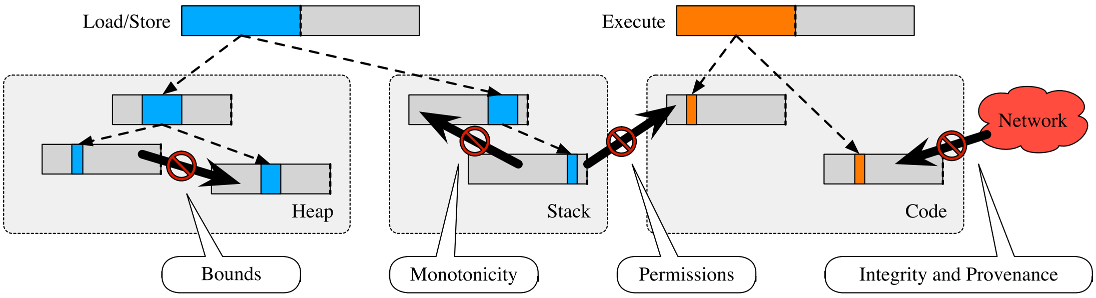


Figure 2.3 – Visualization of CHERIs protection properties.

2.5.2 The CHERI Protection Model

The CHERI protection model [Woo+14] is a joint effort of the Cambridge Computer Laboratory and SRI International's Computer Science Laboratory to develop a hybrid protection model in a hardware-software co-design approach for RISC architectures. It is a generally applicable extension to reduced instruction set computers (RISCs), which builts on top of an existing virtual-memory model. CHERI is a hybrid protection model, that allows both capability-unaware and capability-aware code to run side by side, and thus allows incremental adoption. This design choice is also reflected in the software stack of CHERI: The compiler has two distinct modes and generates either *pure-capability* code, that exclusively uses capabilities, or *hybrid-capability* code, which relies on manual annotation of pointers to use them as capabilities. It provides a new security primitive – a capability – that mediates access to and in protection (sub-)domains within an address space. In other words, a process's virtual address space becomes a capability address space, within which all reachable capabilities determine which protection domains or memory it can interact with.

When using the classification used in Section 2.5.1.2, CHERI is a multi-level memory safety approach. Its checks are performed in hardware and it provides additional safety for legacy, or memory-unsafe, code through its default capabilities. Therefore, it is a hardware-level memory safety approach. On the other hand, CHERI relies on its software stack (discussed in Section 2.5.2.3), especially the modified LLVM compiler, to make use of the additional security primitives and considerably improve upon the memory safety of CHERI enabled programs. Therefore, it is also a compiler-level memory safety approach. One important distinction to existing compiler-level approaches, such as SoftBound, is that this software-stack is not included in the Trusted Computing Base (TCB). Illegal modifications or usage of capabilities, especially widening processes capabilities, will always lead to a hardware exception, meaning that a malicious compiler can never exceed the permission which the program receives from its runtime. In its most basic version CHERI provides *spatial memory safety* and compartmentalization, although recent advances have extended it with *temporal memory safety* in certain usage cases. The following sections try to give a compacted overview of the core principles of the CHERI architecture (Section 2.5.2.1), how it can be mapped into hardware (Section 2.5.2.2) and how its complimenting software stack is built(Section 2.5.2.3). They are based on the CHERI-ISA description [Wat+19b], which may be used as a reference.

2.5.2.1 Capabilities

A capability can be seen as an *unforgeable* token of authority through which access is mediated. A protection domain then refers to a set of capabilities that allow both data access and control flow within a virtual address region. Capabilities are modeled after pointers, with additional metadata to protect their enforce the CHERI protection model. From an architectural perspective, it is a hardware data type and can be destructured into its referenced virtual address, its bounds, and additional protective metadata (see Table 2.1). The additional metadata ensures six properties: (1) integrity, (2) pointer-provenance, (3) monotonicty, (4) bound checking, (5) permissions and (6) encapsulation (see Figure 2.3).

CHERI ensures pointer *integrity* through the out-of-band tag bit. When a capability is modified illegally, either through oversight or malevolence, its tag bit is cleared and any subsequent use of the capability fails.

Capabilities also provide *pointer-provenance*, i.e., they ensure the origin, or provenance, of each pointer [Dav+19b]. More specifically, by only allowing newly-created capabilities to be derived from existing capabilities, each capability must have a valid provenance, if the initial capability state has provenance. On system boot, CHERI-enabled processors derive default capabilities that span the entire address space, and from which the embedded system or operating system can derive capabilities for its protection domains.

Furthermore, capability creation must always be *monotonic*, that is any newly created capability may only narrow the permissions or range of an existing capability. Together with tagged integrity protection, this provides *unforgability* of capabilities, which is the foundation of the CHERI protection model.

Where a capability is used to access memory, *bounds checking* limits, in addition to its permissions, the extent of memory that can be accessed. The region triple $(base, address, length)$ fully specifies the address space region to which a capability has, albeit restricted, access. While a capability referenced address might move outside its bounds, attempts to dereference out-of-bound capabilities will throw to an exception. Typically, these bounds originate in an allocation event, where a language runtime sets them dynamically.

Additionally, a capability is limited by their *permissions*, which restricts the way it is used. For example, a compiler might restrict a capability so that it can only be used for execution, by granting only the “execute” permission. A capability that points to an array in memory might, on the other hand, have read and write permissions. Specific CHERI implementations, especially when used in conjunction with a custom compiler, might furthermore implement language features (such as C’s `const` specifier) directly through capability permissions. Last but not least, capabilities permissions might allow it to use the seal/unseal mechanism, used to provide immutable capabilities. Available permissions may vary between CHERI implementations. Table 2.2 gives an overview of a selection of available permission for the MIPS implementation of CHERI.

Capabilities may either be *unsealed*, i.e., modifiable and dereferencable, or *sealed*, i.e., non-modifiable and non-dereferencable. CHERI provides pointer *encapsulation* through this sealing

Field name	Width [bit]	Purpose
Tag	1	Integrity bit, stored out-of-band
Permissions mask	∼	Permissions available through this capability.
SW permissions mask	∼	Extended permissions for use in software.
Flags	∼	Architecture specific capability flags.
Address	64	The virtual address, which the capability references.
Object type	64	Used for sealed capabilities.
Base	64	The base address of the capability’s memory region.
Length	64	The length (in bytes) of the capability’s memory region.

Table 2.1 – The CHERI capability data type. “∼” refers to a variable sized field.

Name	Bounds	Use
Global	-	Allow this capability to be modified by capabilities that do not have *Permit_Store_Local_Capability*
Permit_Execute	Region	Allow this capability to fetch and execute instructions
Permit_Load	Region	Allow this capability to load words from memory
Permit_Store	Region	Allow this capability to store words to memory
Permit_Load_Capability	-	Allow this capability to load other capabilities with valid tags
Permit_Store_Capability	-	Allow this capability to store other capabilities with valid tags.
Permit_Seal	Object type	Allow this capability to authorize sealing of another capability, where `other.otype == self.address`
Permit_CCall	-	Allow this sealed capability to be used to transition between protection domains.
Permit_Unseal	Object type	Allow this capability to authorize unsealing of another capability, where `other.otype == self.address`
Access_System_Registers	-	Allow access to privileged processor registers, such as registers for interrupt management or MMU configuration.

Table 2.2 – Selection of permission bits in the MIPS implementation of the CHERI protection model. Adopted from [Wat+19b].

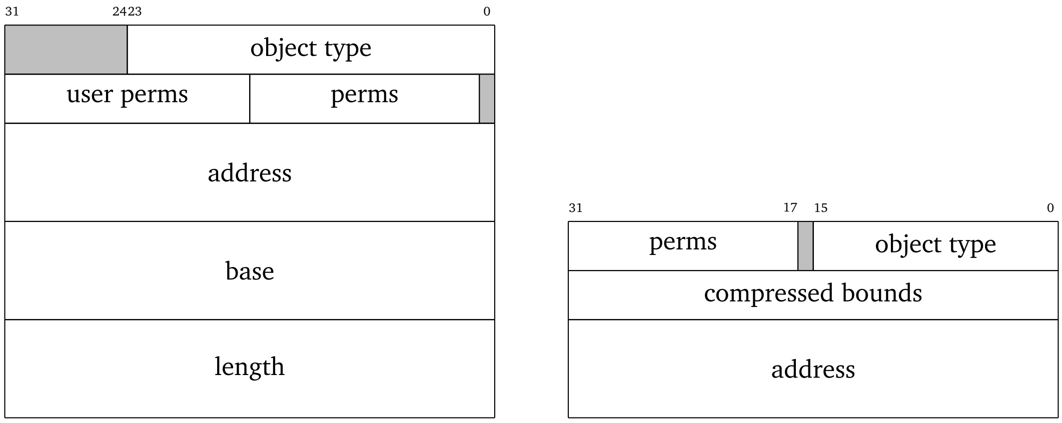
mechanism. Unsealed capabilities, when passed to capability-aware load or store instruction, might be dereferenced to mediate memory access inside the virtual address space. Sealed capabilities, on the other hand, may be used as object capabilities that can be invoked and unsealed. To seal or unseal a capability, the (un-)sealing capability must have the *(un-)seal* permission and a matching object type with the capability to be (un-)sealed. A capability may also be invoked through special capability-aware `call` and `return` instructions to provide, which allows the implementation of an effective inline address space compartmentalization. A domain crossing in the CHERI compartmentalization model always requires a pair of two capabilities with matching object types to initialize the target protection domain. More specially, any protection domain may be uniquely described and initialized from two capabilities: The data capability, which is used to derive any memory capabilities inside the domain, and the control capability which specifies the domain's entry point and execution bound. When a domain crossing is executed, the CHERI-enabled processor installs the provided pair of capabilities *atomically* to the corresponding architectural registers (see Section 2.5.2.2) and execution continues in the target domain. A similar scheme is used during exception handling: By installing a pair of capabilities to be used when an exception occurs, CHERI allows an exception handler to break the monotonicity guarantee and hold privileged access, even when the domain in which the exception occurred has limited access. Recently, CHERI added support for *sentrys*, which allow domain crossings similar to the previously mentioned call and return mechanism, but instead uses only one code capability to facilitate the crossing. Only the code capability is changed and consequently jumped to. Finally, while sealed capabilities are mostly used to implement protection domains, they may also be used by a software-stack to construct arbitrary protected references.

The CHERI protection model does not provide a general way to revoke or invalidate, an existing capability without modifying each instance of it. However, when it is coupled with a virtual memory system, revocation is still possible. By invalidating the address which is referenced by the capability and instead of mapping the data at a different location in the virtual address space, a capability may be invalidated. Subsequently, a modified capability, with different permissions can be created for the new virtual memory address. Other techniques, for example, the usage of capabilities tagged with an identifier and a global capability table or memory scanning approaches, exist too [Xia+19]. They have been successfully used to implement *temporal memory safety* for heap memory in the FreeBSD operating system [Fil+20]. Nonetheless, revocation is not a primary design goal of the CHERI protection model and must be retrofitted if it is needed.

2.5.2.2 The CHERI hardware architecture

Similar to a classical load-store architecture, capabilities may be stored in memory or held in capability-enabled registers when they are used or modified. Their in-memory or in-register representation might vary considerably from the data type structure presented previously (see Figure 2.4). In the 64 bit MIPS implementation of the CHERI protection model, for example, each capability is 256 bit long with one additional tag bit, which is stored separately (see Figure 2.4a). Of these 256 bits, 192 bits are reserved for the region triple, and the remaining 64 bits are shared among the permissions and object type fields (31 bits and 24 bits respectively), while the flags fields remain unused. In total, a stored capability, therefore, takes four times the storage of a normal 64-bit pointer. Modern implementations of the CHERI protection, e.g., CHERI-RISCV, instead use the CHERI Concentrate [Woo+19] representation (see Figure 2.4b). It uses a floating-point format to coalesce the region triple and reduce available permissions and object types, to fit a 64-bit capability into 128-bit (or 32-bit capabilities into 64-bit) memory or register space reducing the storage overhead. In comparison to the uncompressed representation, CHERI concentrate requires stronger alignment of capabilities in memory to compensate for compression. Still, CHERI Concentrate halves the storage overhead of the protection model.

Architectures, which implement the CHERI protection model, are free to explicitly split the register set into capability-only and normal registers or provide a merged register set, that supports both general-purpose values and capabilities in registers. The program counter capability (PCC) replaces the default program counter register regardless of this decision. A capability enabled program counter allows the CHERI protection model to constrain the control flow by restricting instruction fetch access, and finally, provide compartmentalization. Similarly, other control-flow related registers must be extended, such as the exception program counter, if they are present in the



(a) 256 bit CHERI capabilities.

(b) 128 bit CHERIConcentrate capabilities.

Figure 2.4 – Compressed (128 bit) and uncompressed (256 bit) capability representations.

architecture. Furthermore, CHERI extends the architecture with a default data capability (DDC) register, which holds the default capability that is used for legacy, i.e., non-capability, loads, and stores.

The CHERI protection model extends the base instruction set with several instructions. These can be used to modify, create, or use capabilities. By adding explicit instructions, which only target capabilities, a key design choice of the CHERI protection model, *intentionality*, is achieved. Additionally, capability-aware instruction are *unprivileged*. This does not only reduce context switch overhead but also allows processes to create protection sub-domains without operating system interaction. Nonetheless, it is also the reason why the integrity of capabilities must be protected in memory or registers. Due to the unprivileged nature of capability-aware instructions, it is expected that capabilities are also stored in unprivileged memory. Thus, they may be modified by capability-unaware stores, or arithmetic register operations and must be protected by the aforementioned tag mechanism. The authors of HardBound rely on tagged memory for the same reason, with one important distinction: It allows arbitrary capability creation and, therefore, a capability in HardBound does not constitute a protection domain like in CHERI.

Due to different underlying design principles of the extended architectures, specific implementations of the CHERI architecture vary in micro-architectural implementation details. For example, while ARM processors avoid pointer modification exceptions and instead defer such exceptions to access time, RISC-V architectures always raise precise exceptions. The differences between implementations can be (mostly) reduced to three design decisions:

1. Should the register set be split or merged, or are both variants supported?
2. How should the architecture treat capability errors?
3. How are capability-aware instructions encoded?

Currently, four implementations exist with varying degrees of completeness. They are summarized in Table 2.3. MIPS is the first architecture for which the CHERI protection model was implemented and is therefore the most mature adoption. It supports a split register set, in which all capabilities are managed by an additional co-processor, called CP-1, which runs in parallel to normal processor execution. Due to the vast opcode space in MIPS, CHERI instructions were given their own encoding, completely separate from normal load/store encoding. When a capability error occurs, the capability co-processor raises a synchronous exception which interrupts the current program flow. RISC-V follows similar concepts, except that it supports both split and merged register sets. The ARM implementation on the other hand only supports merged register sets. Additionally, it does not raise precise exceptions when an invalid capability modification occurs. Instead, the capability is simply marked as corrupted (by clearing its tag bit) and execution continues. When the corrupted capability is subsequently used, an exception occurs and the program is notified. Both the RISC-V and the ARM implementation use a so-called “capability-mode” extension, in which the opcode for a capability-aware load/store instruction and a capability-unaware load/store instruction is the same. The processor is extended with an additional mode-bit, the capability-mode bit, which indicates whether the currently processed instruction is interpreted as a capability-aware instruction or not. Finally, an early sketch of a CHERI implementation for x86-64 exists, which uses merged register sets and a capability mode.

2.5.2.3 The CHERI software architecture

To use the previously described primitives to provide memory safety in software, CHERI implements a versatile software stack. More specifically, CHERI’s software stack discussed here is used to provide

ISA	Status	Register file	Error Mechanism	Shared Opcodes
MIPS	Complete	Split	Exception	✗
ARM	Experimental	Merged	Invalidate	✓
RISC-V	Draft	Both	Exception	✓
x86-64	Sketch	Merged	Undecided	✓

Table 2.3 – Existing CHERI implementations, adapted from [WSWMN19]

memory safety to two historically unsafe languages: C and C++. At the lowest level, the software stack is a modified version of the LLVM compiler and linker, on top of which various support libraries, such as a hardened standard library are implemented. Then, a CHERI-enabled operating system is stacked on top, which may reuse the high-level primitives provided by the hardened support libraries. Finally, a CHERI-enabled program may run as the last layer of the software stack. Two basic protection modes are implemented by the CHERI software stack: Fine-grained memory safety and scalable software compartmentalization. Only the implementation of fine-grained memory safety is discussed here as it is the relevant technique for this thesis, but [WSWMN19] may be used to gain more insight into the compartmentalization mechanism.

Fine-grained memory safety is provided by the compiler and the runtime system and works by replacing C/C++ pointers, either explicitly or implicitly, with capabilities. Two new compilations modes are defined: pure-capability mode and hybrid-capability mode. *Pure-capability* mode uses capabilities for all C/C++ pointers including implied pointers, such as return address or stack pointer. Trying to dereference an integer, that is a legacy pointer, will lead to an exception, in the same way, that using a capability in an invalid way would. Pure-capability compiled code is usually incompatible with legacy code due to different pointer sizes. *Hybrid-capability* mode, on the other hand, uses explicit language-level annotation for pointers to distinguish between pointers which are backed by a capability and unprotected pointers. Pointers are by default unprotected and use the DDC, when dereferenced. Therefore, pointer size is unchanged (in the default case), and hybrid-capability compiled code is binary compatible with both pure-capability and legacy code. As such, it can be either used as a bridging interface between both variants, or when the capability system proves too restrictive to implement the required functionality (e.g., bootloaders, early kernel startup).

When using the pure-capability mode, capability creation is deferred to the runtime, which will derive valid capabilities either at runtime, e.g., when a dynamic pointer is allocated, or during load time, e.g., when a global capability is created. Additionally, a C/C++ runtime might use load-time capabilities to protect runtime internal structures, such as the procedure linkage table (PLT), or thread-local storage (TLS), from corruption. Furthermore, the PCC is usually the only capability, which holds execute permissions and access to the code region of the loaded program. The DDC usually contains the `NULL` capability, an always invalid capability, such that any legacy pointer access will throw an exception.

The described memory safety techniques have been successfully used to implement fully memory safe operating systems: Notable examples are:

CheriBSD [Cheb], a port of the FreeBSD operating system, which supports both hybrid and pure-capability mode compiled binaries in its userspace and provides compartmentalization and memory safety.

CheriFreeRTOS, a port of the FreeRTOS operating system, which is compiled in pure-capability mode to provide memory safety.

CheriOS [Chea], a clean-slate, single-address space µ-kernel design which uses CHERIs compartmentalization techniques to efficiently implement its design primitives.

2.6 Summary

This chapter first defined the terms fault, error, and failure in the context of this thesis. Next, it used them to define a resilience concept that is based on the *RAMS* principles. It then presented fault injection, a fault forecasting technique, that can be used to evaluate the dependability and resilience of a system. Furthermore, related work was presented that mostly focused on existing soft-error protection schemes due to the lack of research, which used memory protection as a method for soft-error protection. Finally, memory protection was defined and the CHERI memory protection model was discussed in detail. It is a hybrid protection scheme that provides its memory protection through a combination of hardware extensions and software augmentation. Similar to the presented soft-error protection schemes, it works between the architectural and software level of an architecture. Its influence on soft-error resilience is further evaluated in the rest of this thesis.

ARCHITECTURE 3

After laying the theoretical foundation, this chapter first outlines – from a theoretical standpoint – how extending an architecture with memory protection might improve the soft-error resilience in Section 3.1. Next, Section 3.2 presents the fault injection framework FAIL*, that forms the basis on which the soft-error resilience of an architecture can be evaluated. Finally, Section 3.3 presents the concrete memory protection architecture – CHERI RISC-V – and its integration into the FAIL* framework.

3.1 Soft-Error Resilience Through Memory Protection

To understand how a system, which employs memory protection, might have improved soft-error resilience, consider what improved resilience constitutes. Generally speaking, a system can be considered to be more resilient than another system, if it adheres to stricter guarantees regarding the *RAMS* principles (see Section 2.2.1). For example, if such a system provides better fault detection, i.e., detects faults earlier, it can be considered to be more *reliable* than the baseline system. In consequence, it is more resilient, as it allows for failures to be detected and corrected in a more timely fashion. Similarly, a system, which by design exhibits less catastrophic modes of failures or exhibits such failures less frequently, may be considered to be *safer* and, thus, more resilient than a comparable system. In the context of this thesis, a system is defined to be more resilient against soft-errors if it either:

1. Exhibits reduced frequency of specific failure modes *or*
2. Improves detection of existing failure modes

Therefore, to assess a system's resiliency, its failure modes must be considered in greater detail. A system, as described in the previous chapter has six different failure modes: early timing failures, late timing failures, erratic failures, halt failures, unsignaled content failures and signaled content failures. Of these six failure modes, two are considered in greater details as they provide opportunity for improvement through memory protection: Unsignaled content failures and late timing failures.

Unsignaled content failures are system failures in which the system silently corrupts content or, in other words, data contained in a system. These system failures are often called silent data corruption. For a processor-based system most of its corruptible data is usually contained in an attached random-access memory, while only a small fraction is loaded to its registers at any given moment. Depending on the system architecture an executed instruction may operate directly on data contained in memory and registers (register-memory architecture), or it may only operate on register data (load-store architecture). The memory protection architecture evaluated in the context of this thesis is a load-store architecture, and only such architectures will be considered from now on. To access data contained in memory, a load-store processor must follow a two-step process: First a pointer to the data is constructed in a register, usually through arithmetic computation from an existing pointer, before it is loaded by passing the constructed pointer to a dedicated *load instruction*. Similarly, such a processor must first construct a destination pointer, before writing register data to memory by passing the pointer to a dedicated *store instruction*. For load-store architectures data may either be corrupted *directly*, when a fault affects the data stored in memory, or *indirectly*, when the pointer used to access it is corrupted by a fault and subsequently the wrong data is accessed and propagated. While direct data corruption is not influenced by memory protection, indirect corruption, i.e., constructing or accessing an invalid pointer, can be detected and, therefore, prevented. Memory-protected systems often impose additional restrictions on pointer creation, such as monotocity (see Section 2.5.2.1), which prevent arbitrary pointer construction. Additionally, these systems introduce additional pointer metadata, such as the intended pointer usage, which further restricts the space of faulty, but valid, pointers. Unprotected architectures, on the other hand, allow arbitrary pointer construction, and therefore, are only able to detect indirect corruption when the corrupted pointer leads to a generally invalid, e.g., misaligned, access. As long as the faulty pointer still refers to a valid, albeit incorrect, memory location and if the system does not have a mechanism to detect invalid memory accesses the pointer may be used to silently corrupt data in the unprotected system. In summary, memory-protected systems impose restrictions on pointer creation and usage and, thus, should in theory provide improved protection, or at the least detection, from otherwise silent data corruption.

Next, consider late timing failures. Late timing failures, or as they are more commonly called *timeouts*, may either occur due to corrupted data or due to corrupted control flow. Corrupted input data leads to timeouts when it is used to bound the computation of a system. To give an example, consider a system which processes a different amount of elements during each run. First, it determines the size of its work queue and saves this information to a faulty storage element, which instead saves a much higher value, e.g., by flipping the most-significant bit. When the system subsequently tries to use this information to process queued elements, e.g., as a loop bound, its computation will take much longer than expected. Therefore, the system will not deliver the result in time, which constitutes a late timing failure. Similarly, corrupted control flow can induce a late timing failure. For example, if the system reaches the code section that outputs its results never or heavily delayed, it might fail to deliver its service in time or, in other words, exhibit a late timing failure. To understand how a fault can induce erroneous control flow, consider a processors function call mechanism. Before continuing execution in the callee, its caller first saves certain registers, sets up the callees call frame and finally stores the correct return address to the stack. If a fault happens in the storage element that backs the stack memory, a return address can be corrupted and consequently lead to an erroneous control flow when the callee returns control to the caller. Unprotected systems will accept the faulty return address and continue execution, given that the faulty address still refers to a generally valid memory location. This is especially problematic, when a processor architecture (as it is the case for ARM A32 [Arma], MIPS [Mip] and AVR [Avr]) consider an instruction that is of value zero to be the no-operation instruction. If such a processor jumps to a zero-initialized, but unused section of memory due to a faulty return address, it will silently continue execution, with no failure indication to the user, until a timing failure occurs. Memory-protected systems, however, can offer some protection against timeout failures. As previously discussed, memory protection schemes often restrict pointer creation or modification, and thus should prevent modifications to saved return address through corrupted pointer to stack variables. Corruptions of variables which bound the computation of a system, such as a loop counters, will be protected similarly. Memory-protected systems may additionally restrict the range of valid call and return addresses by augmenting the program counter, as done in the previously discussed CHERI capability architecture. Here, the program counter is restricted by a bound, that restricts possible jump offsets from the current program location. Transitions outside this range require a special compartment call, and hence the set of possible incorrect, but valid, return addresses is further restricted. In conclusion, memory-protected system will protect control flow information saved in memory due to imposed restrictions on pointer creation and subsequently restrict possible late timeout failures by detecting error conditions early.

3.2 The FAIL* Framework

This section describes the FAIL* framework, a versatile FI *meta*-framework [Sch+15], developed to aid architecture-independent full fault space evaluation of complete systems. It is used in this thesis to evaluate both an unprotected system and its memory protected counterpart in regard to transient faults. FAIL* is an acronym for ***FA**ult **I**njection **L**everaged* and its asterisks stands for its versatility with regard to its target architecture. By default, FAIL* supports both *simulation-based* FI and *hybrid* FI backends, which modify the target state through a JTAG-based approach. For the rest of this chapter, however, focus is put on the simulation-based backends of the FAIL* framework, since the architectures depicted in this thesis are evaluated in a simulation environment. Originally, FAIL* was developed since existing FI tools lacked *generability*, i.e., target independent experiments, or were too

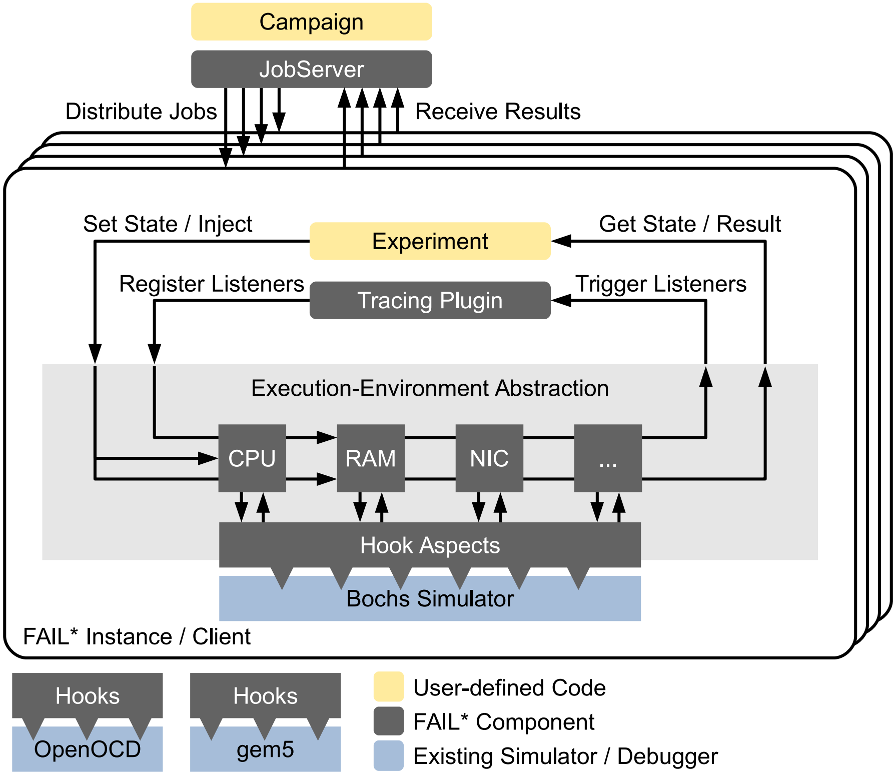


Figure 3.1 – Structure of the FAIL* plumbing layer, or in other words, its FI abstraction. Taken from [Sch16].

specialized, i.e., only supported a single architecture with non-portable experiments [Sch16]. FAIL* strives to balance both needs, and create an abstraction of the target – the execution environment abstraction (EEA) –, which provides fine-grained target state access while allowing experiment sharing across backends.

On top of the EEA layer, FAIL* builds a parallelized client server architecture, which manipulates the target architecture through its EEA interface (shown in Figure 3.1). In its essence, each client is a version of the targets architectural simulator that is extended through the EEA layer and controlled by a user-defined experiment procedure. Both the extended architectural simulator and the experiment procedure run in different co-routines, and are cooperatively scheduled. The EEA exposes two distinct interfaces to the experiment procedure. First, a target state interface is available through the EEA. It allows complete state dumps and restores, as well as fine-grained deep state access, e.g., setting a bit in a CPU register. Secondly, an event interface is exposed that allows the experiment procedure to wait for events by registering *listeners*, such as reaching a certain instruction, or passing a set amount of simulator time. Additionally, the EEA facilitates FAIL* cooperative scheduling approach: An experiment may resume execution of the extended simulator at any point, however, it is only returned control, when a previously registered listener is triggered or the workload execution finishes. A control-flow example is shown in Figure 3.2. While user-defined, each experiment procedure usually either *traces* the simulators execution or *injects* (one or more) faults during its execution and records the result. This duality between tracing and injecting is inherent to fault injection: Before a program can be injected its set of possible fault points, i.e., the fault space, must be discovered, by *tracing* a “golden-run” of the program (see Section 2.3). Nonetheless, how a execution trace maps to possible injection points, or in which way these are subsequently injected, depends entirely on the user-defined experiment procedure. To ease the transition for new users FAIL* provides a generic tracing and injection experiment procedure, called the *assessment layer*. In contrast to the previously discussed *plumbing layer* [Sch16] it supports both memory and register

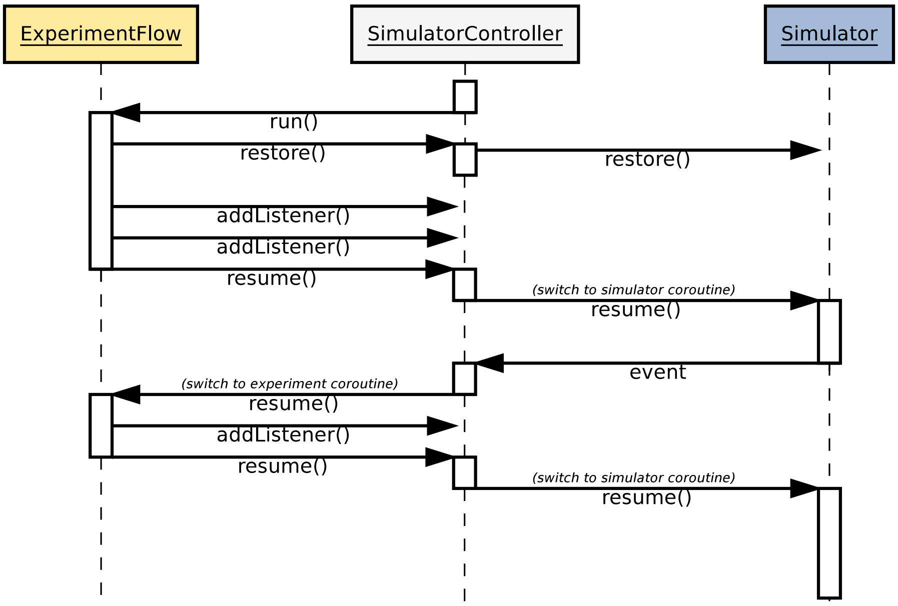


Figure 3.2 – Sequence diagram of a typical control-flow between an experiment procedure and the simulator which has been extended with an EEA layer. Taken from [Sch16].

injection, and injects bits according to a uniform single-bit flip or uniform byte burst flip model. It stores the tracing information, the fault space and fault experiments results in a central database for easy parallel access during injection, and to simplify subsequent result analysis. A generic fault injection experiment, a *campaign*, then consists of four distinct steps, not considering the result analysis (shown in Figure 3.3):

First a "golden-run" trace of the workload is generated by running the simulator, while all relevant state access, i.e., memory accesses, executed instructions or for example serial output, is recorded. Typically, only part of the workload is traced. For example, it might be unwanted to inject the workloads startup code, since it is not part of the fault model. Consequently, two special instructions are defined – the `start_marker` and `stop_marker` –, which instruct FAIL* to start and stop the tracing respectively. Additionally, a dump of the target state is saved by the FAIL* client when it starts tracing workload execution. It is used during the injection phase of the campaign to restore the system to a known good state.

Secondly, the information collected by tracing the "golden-run" is used to extract possible fault points through a separate program called *importer*. Multiple importer implementations exist that differ in the type of possible fault points they import into the database for later injection. One may, for example, only convert recorded memory access to fault points, while another importer may only deal with access to certain registers. They, in other words, each explore a different part of the traces fault space and import it into the database. The choice of importer remains with the user, allowing for full flexibility.

Thirdly, the set of possible fault points is reduced, i.e., *pruned*, to avoid redundant injection and enable full fault space evaluation even for large workloads. A set of planned injections, or *pilots*, is generated which each evaluate a group of similar behaved faults. This technique is based on the observation, that in any fault space multiple fault points may result in the same behavior when injected. To give an example, consider a fault occurring in an arbitrary byte of memory. Such a fault can only be activated and cause a failure, if the memory location holds a value, i.e., was previously written, and is read, i.e., used, after the fault occurs. The time interval during which the fault can be activated forms an equivalence interval. It does not matter, at which time during the interval the fault happens, it will always be activated by the read at the end of its equivalence

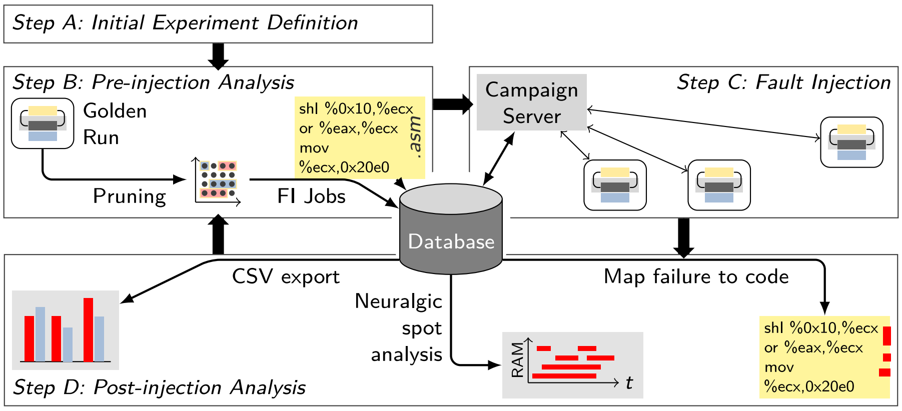


Figure 3.3 – Structure of FAIL* assessment layer. It consists of four steps – tracing, import, pruning and injection –, which store their (intermediary) results in the internal database. Finally, during post-injection analysis, this database can be queried to extract results for the FI campaign. Taken from [Sch16].

interval. Consequently, all fault points inside this equivalence interval can be considered to be equal in their injection result, and only one candidate must actually be injected to record the effect of all fault points. All other fault points are pruned, and are henceforth considered to be invisible to the injection experiment procedure. Their existence is, nonetheless, recorded in the database so that each injection result may be weighted according to its equivalence interval, when calculating failure metrics during data analysis. In addition to equivalent faults, a fault may be pruned if it is overwritten after it occurs, i.e., if its equivalence interval ends with a write instead of a read. In this case the faulty value is lost and the fault is rendered ineffective. After evaluating all possible fault points for their effectiveness, the *pruner* writes each effective fault to the central database, thus building a list of scheduled injections. Note that the presented pruning technique, called *def/use* pruning, is only one of many possible techniques. For example, Schirmeier presents a heuristic-based pruning approach [Sch16], which additionally groups fault points according to their architectural state.

Lastly, a central server and multiple parallel clients are launched to process the scheduled injections. Each client receives a subset of all scheduled injections from the central server, which it subsequently processes in serial. During each injection it follows a simple process: First the simulator is rewound to the workloads `start_marker` by restoring the state dumped in the first campaign step. Then, it is forwarded until it reaches the instruction at which the fault shall occur. At this point the architectural state is injected through the EEA in accordance with the selected injection technique. Finally, the simulators' execution is continued, either until it reaches the end of the workload, or until the simulator stops execution due to an erroneous condition, such as a trap. To indicate a successful or failed run, the workload jumps to one of two special instructions called `ok_marker`, and `fail_marker`, which indicate a valid or invalid result respectively. The injection's result is subsequently recorded in the central database and the client repeats the process for the next scheduled injection. After all scheduled injections have been processed, the central server instructs all connected clients to quit before shutting down.

After all injection results have been collected, the developer can continue with the (optional) *post-injection analysis*. Each result contains, in addition to its outcome, a detailed account of the injected fault and its result. In total, it contains information on the injected element and the precise offset within it, the injections' outcome with additional information, such as the addresses which caused a subsequent error condition, and the simulator time when the result occurred. Through this detailed information, each result can be linked, not only to its fault point, but also to the event which resulted in the fault point. This allows for extremely fine-grained injection analysis, such as mapping injection results to source code lines. FAIL* additionally allows calculation of well-known FI metrics such as *fault coverage*. However, this metric is often unfit when comparing differently-sized fault spaces, and thus, FAIL* also supports calculation of *absolute error counts*. This metric is independent of fault space size and thus can be used to compare different implementations, or even architectures, for their resiliency to transient faults.

3.3 CHERI-FAIL: Combining Fault Injection with Memory Protection

After discussing the theoretical improvements that a memory-protected architecture might offer in regard to transient faults, and discussing the FAIL* framework, this section presents the integration of a memory-protected architecture – CHERI RISC-V – into the FAIL* framework to validate the hypotheses made in Section 3.1. Together, with the existing integration of RISC-V into the FAIL* framework, a comprehensive evaluation platform – CHERI-FAIL – is formed, which allows conclusion on the effectiveness of the protection schemes provided by the CHERI protection model.

The CHERI protection model is selected to evaluate the presented hypotheses because it offers both protection of pointers and control-flow. Additionally, multiple well-tested implementations exist, e.g., CHERI MIPS or CHERI RISC-V, and have been used in numerous research projects. For each implementation a complete software-stack, including a C-compiler and assembler, is available, which further simplifies development of representative workloads. Furthermore, its protection works at the hardware-level (see Section 2.5.1.2) and requires no runtime libraries. Loading additional, potentially large, runtimes libraries increases both the execution time and the memory footprint and, therefore, the set of possible faults significantly. Finally, cycle-accurate simulators exist for each CHERI implementation which can be integrated into the FAIL* framework.

Specifically, the RISC-V implementation of CHERI in addition to a baseline RISC-V implementation is selected for comparison. Firstly, both are load-store architectures and should exhibit the discussed decrease in unsignaled content failures and late timing failures (see Section 3.1). Furthermore, both architectures have a cycle-accurate simulator based on the Sail architectural description language [Armb]. In fact, the CHERI RISC-V simulator is an extension of the RISC-V simulator aiding comparability. Finally, an example integration of the 32-bit variant of the RISC-V simulator was developed during Marcel Budoj's master thesis [Bud20] and can be used as the basis for the integration of the CHERI RISC-V architecture into the FAIL* framework.

The rest of this section is structured as follows:
First, Section 3.3.1 outlines the challenges, which prevent a straight-forward integration of CHERI RISC-V into FAIL*. Next, Section 3.3.2 discusses two extensions to the FAIL* framework aimed to overcome the previously presented problems. Section 3.3.3 presents the final integration of CHERI RISC-V into the FAIL* framework.

3.3.1 CHERI-FAIL: Challenges

To understand the challenges that come with integrating CHERI RISC-V into FAIL*, consider its main difference to the unprotected RISC-V architecture: The tagged memory-architecture. A tagged memory architecture is an often used technique that annotates each byte of memory with an additional *tag* bit. CHERI uses this bit to ensure the integrity of its capabilities (see Section 2.5.2.1). In short, each legitimately created capability, e.g., through valid modification of an existing capability, has a set tag bit. Explicit writing of the tag bit is disallowed, which makes it impossible to craft valid capabilities by memory manipulation. While each tag bit can be stored in-line with the data memory, CHERI RISC-V chooses to store the tags bits out-of-line in a seperate bit-addressed memory, which shares the the address space of the data memory. Due to its addressing scheme and access width this implementation choice complicates CHERI's integration into FAIL*.

First, consider the addressing scheme used for the tag memory. During a memory access, it is indexed with the data memory address to read or write its corresponding tag bit. Consequently, the address space of the tag memory and the data memory are shared. In other words, for each valid memory address, two pieces of information exist: its data and its tag bit. During trace import, i.e., when exploring the traces fault space, FAIL* uniquely identifies each fault point by its memory address and time of fault. To identify registers in this scheme, each register is assigned an unused address of the memory address space. Often, the begin of the memory address space is unused, or *reserved* for device I/O and thus not injected, and can be safely re-purposed to identify the limited set of registers available in an architecture. Consequently, register and memory fault points will never overlap and can be uniquely identified by their address and time of occurrence. However, with tag memory such overlaps are inevitable. During a memory access that loads a capability from memory both the tag memory and the data memory are read. Therefore, two distinct fault points should be imported. However, both are accessed with the same memory address and at the same time. Their respective fault points are the same and the importer, and more importantly, the injection mechanism, has no way to distinguish the two fault points for the different types of memory. To solve this overlap, either each fault point must be additionally identified by the type of memory to which it belongs, or a unique mapping must be found, which translate their respective addresses to a globally unique, virtual, memory address. Section 3.3.2 discusses the creation of this unique mapping.

Secondly, consider the width of each access in the data and tag memory. By default, FAIL* assumes a byte-addressed data memory, and therefore, a byte-addressed fault space. In other words, one byte of memory belongs to each fault point discovered during import. Furthermore, each access tracked during tracing is assumed to be aligned and of byte-length, i.e., reads *n* full bytes. This is a reasonable assumption for most modern architecture, given that misaligned access is discouraged in most modern architectures (see [Inta]), or even optional (see [Wat+19a]), i.e., leads to a system trap. Similar to their respective data memory access, accesses to tag memory are byte-aligned. However, only a single bit belongs to each "byte-address". FAIL*, in contrast, assumes that each memory address in the tag memory contains one byte of information and will inject each bit of the fault point. Of these injections only one will have an effect when injected, while the others do not change the systems behavior. Effectively, seven artificial no-effect faults are added to the fault space for each accessed tag, which skews the obtained FI results. Consequently, a way must be found to track fault points at bit-granularity, or mark actually accessed and, thus, possibly faulty, bits of each address. Section 3.3.2 discusses a masking scheme, which tracks fault points at bit-granularity, while retaining a byte-addressed fault space.

3.3.2 CHERI-FAIL: Extensions

In the following two extensions to the FAIL* framework are presented, which solve the previously discussed integration problems of CHERI RISC-V. First, in Section 3.3.2.1, a *virtual* fault space is discussed which allows a unique mapping of architectural elements to addresses even when considering multiple types of memory with shared address spaces. Then, Section 3.3.2.2 discusses how this abstraction was extended to support fault points with bit-granularity.

3.3.2.1 A virtual fault space

An important step of each FI experiment is the discovery of possible faults, or in other words, the workloads *fault space*. For a given system architecture, the set of possible faults for a given workload is defined by all architectural storage elements, that are used during the execution of the workload. Each of these storage elements can experience a fault at any time during the workloads' execution.

During trace import FAIL*, consequently, identifies the set of possible faults, which can occur in an architectural element, by their time of occurrence, and an abstract *fault space address* which uniquely identifies the architectural element. The accumulation of these individual *fault points* forms the two-dimensional fault space (see Section 2.3) of the workload, which can be further processed, or pruned. Finally, given a fault model, each abstract point in this fault space can be mapped to one or more concrete injections and be injected. The amount of concrete injections depends on the amount of data stored in its architectural element, or in other words, the fault points size. For example, a fault point might represent a one-byte chunk of data memory, and produce eight concrete injections for a uniformly-distributed single-bit flip fault model. Note that, while the size of each fault point need not be uniform, FAIL* assumes that each fault point is byte-sized,

i.e., represents one or more byte of storage. The fault space abstraction has several benefits for the pruning strategies used by FAIL*, however, it needs a unique mapping from architectural elements to fault space addresses. To give an intuition on possible fault space addresses for architectural elements, three systems with increasing degrees of complexity are discussed in the following.
First consider a system, in which a fault only occurs in used bytes of the data memory. In such a system, each possibly faulty byte of data memory is an architectural element, which by design is uniquely identified by its memory address. Each fault space address, therefore, *is* the memory address of the corresponding byte of data memory.
Next consider a system, in which a fault can occur in used bytes of data memory and registers. Given that a unique register identifier exists, a fault point can be uniquely mapped to either a register or a byte of memory. However, a globally unique mapping is required to model faults in bytes of registers and memory simultaneously. FAIL* implements this globally unique mapping, by assigning each register a unique identifier and using this identifier as the fault space address of register fault points. This effectively maps all register fault points to the beginning of the memory fault space, shadowing all possible memory fault points in the overwritten range. However, most architectures have non-contiguous address space by default [Intb], for example due to memory-mapped device registers. Shadowing such an unused region constitutes no loss of fault space.
Finally, consider a system, which has multiple types of memory, e.g., tag and data memory, such as CHERI RISC-V. Reusing the memory address as a fault space address is not possible due to the overlap. A simple memory access, which also accesses the corresponding tag, would result in two identical fault points due to the shared addressing between tag and data memory.

This aliasing can be mitigated by either transforming the memory address depending on the class of architectural element it belongs to, or by extending the fault point model to include this class. Extending the fault point model, however, complicates the previously discussed fault space processing, such as pruning, significantly and is therefore not pursued in this thesis. Instead, each physical memory or register address space, is transformed or, in other words, mapped into distinct sections of an artificial global fault address space.

Inside this *virtual* fault space, each class of architectural elements is mapped to a distinct, i.e., non-overlapping, address range, or fault space *area*. Consequently, each element's global fault space address consists of both the area's offset in the fault space and its original address. Even if two elements initially shared an address, their respective global fault space address will be unique in this scheme, as long as the original address was uniquely occupied in its area. Each distinct fault space area manages a set of architectural elements, or *fault space elements*, for each of which it must provide an, at least locally, unique address. During fault space generation, this *relative* address is used by the virtual fault space to *encode* the element to a globally unique fault space address by adding it to the area's offset. Similarly, during injection the fault space can be used to *decode* an existing fault space address to return the fault space element associated with it, and inject it.

The virtual fault space implementation in FAIL* follows this hierarchical fault space structure and is shown in Figure 3.4. It consists of three classes – `space`, `area` and `element` – , which implement a global fault space, a fault space area and an (injectable) fault space element respectively. However, their respective implementations are dependent on the systems configuration and must be subclassed to fit its architectural requirements. Typically, each architecture will provide a single subclass of the `space` class which provides the `create_areas()` method. During construction, the `space` class will call this method to query the architectures available fault space areas and their respective sizes to calculate their offsets in the virtual space. When decoding a global fault space address during injection, it then uses this calculated offset to derive the elements relative address before passing it to the respective area for final address decoding. Finally, the `space` class provides a method to query an area by its canonical name. It is used by each importer during the trace import

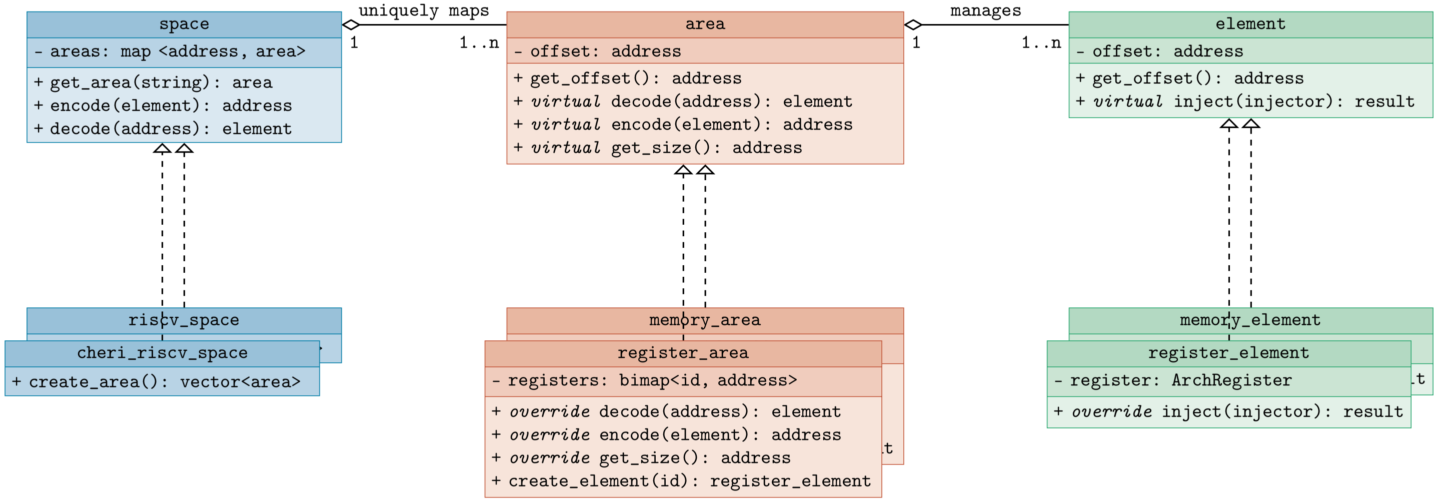

space
- areas: map <address, area>
+ get_area(string): area
+ encode(element): address
+ decode(address): element
uniquely maps
1
1..n
area
- offset: address
+ get_offset(): address
+ *virtual* decode(address): element
+ *virtual* encode(element): address
+ *virtual* get_size(): address
manages
1
1..n
element
- offset: address
+ get_offset(): address
+ *virtual* inject(injector): result
riscv_space
cheri_riscv_space
+ create_area(): vector<area>
memory_area
register_area
- registers: bimap<id, address>
+ *override* decode(address): element
+ *override* encode(element): address
+ *override* get_size(): address
+ create_element(id): register_element
memory_element
register_element
- register: ArchRegister
+ *override* inject(injector): result

Figure 3.4 – The class diagram for the virtual fault space implementation.

to get a reference to its corresponding fault space area. Next, `area` must be sub-classed for each class of architectural elements in the target system. By default, however, it only provides a default implementation for its `encode()` method. Since each element has access to its own offset (via its `get_offset()` method), its global fault space address can be trivially calculated by adding this offset to the areas offset. Nonetheless, if this assumption is invalid for a specific area `encode()` can be overwritten to implement a custom encoding scheme. Besides providing an encoding scheme, each subclass of `area` must provide implementations for a total of four methods: First, it must provide the size of its address space via the `get_size()` method, so that it can receive a correctly-sized address range in the global fault space. Secondly, it must provide a canonical name so that it can be queried through the fault spaces `get_area()` method during trace import. Thirdly, it must provide a `create_element()` method, which is tailored to the importer which will use the area to create fault points. While the `MemoryImporter` creates fault points by passing an absolute memory address to the area, a `RegisterImporter` will instead pass a register identifier and a byte offset within the register. Each call to `create_element()` creates an instance of the `element` class, or an area specific subclass of it, which abstracts an architectural element. Making it injectable through a common interface. Fourthly, each area must provide a mapping of previously created `element` instances to a locally unique address through the `decode()` method. Each subclass of `area` must provide its own `element` subclass, which is self-contained and able to perform an injection of its corresponding architectural element. This injection mechanism is accessed through `element`'s virtual `inject()` function, which takes an injector. Each fault model comes with its own injector, which modifies the architectural element accordingly.

In summary, the virtual space implementation allows the creation of unique elements in a virtual fault space during import, which can be transformed into a globally unique fault space address. This address can then be used during injection to retrieve an abstract architectural element, which can be injected transparently according to a fault model.

3.3.2.2 A bitwise fault space

By default, FAIL* uses byte-sized fault points with a length of one. For the originally considered systems, e.g., Intel IA-32 and ARM A32, memory reads and writes are naturally byte-aligned in length. In other words, a byte of memory or register can only ever be read or written in its completion. For these systems, tracking fault points at byte granularity has no adverse effect. Consequently, FAIL* imports byte-sized fault points, builds equivalence classes during pruning at byte granularity, and schedules injection at byte granularity. Only when processing the actual injection during the final experiment phase, a fault model is applied to inject specific bits of each byte of storage. FAIL* supports both a uniform single-bit fault model, in which each fault point leads to eight concrete injections, and an eight bit burst fault model, in which each pilot leads to a single injection that flips all bits of the injected byte.

However, not all systems have memory writes and reads that are byte-aligned in length, have registers which byte-aligned in length or even write registers at a byte granularity. For example, CHERI RISC-V's tag memory, is byte-addressed, however, each write or read will only ever access one bit of information. To map each bit of information into a byte-addressed fault space, it must be appended with seven unused bits, that while tracked through the pruning and injection step of the FI experiment, have no effect when injected. Consequently, creating a byte-sized fault point for a single bit of tag memory does not only result in unnecessary work in the import and pruning steps of the experiment and cause injections, which have no effect, it also artificially inflates the fault space and skews the FI experiments results.

A different, but related, problem arises for the tag memory of CHERI RISC-V's capability registers. Instead of storing the tag bits in-line with the capabilities values, all tag bits are instead mapped to a shared, bit-packed, virtual tag register. A write to this register, which is artificially generated when any byte of a capability register is written, will only ever write a single bit of the tag register. More specifically, it will write the tag bit which belongs to the written capability register. To track such a write in a byte-addressed fault space, the whole byte in which the written bit is contained must be considered a fault point. This however is detrimental to the experiment, even if only a single bit was written of the byte in the tag register, the tag bit of seven other registers would be injected during the injection phase of the experiment. Therefore, any results gathered from such an injection campaign would provide seven result that provide no meaningful insight, and again, skew the FI experiments results.

To tackle this problem, one might consider implementing a bit-addressed fault space. In such a fault space each fault point only refers to one bit of storage. However, this inflates the fault space by a factor of eight for uniformly-distributed single-bit faults, which increases runtime and storage requirements for tracing, import and pruning. Instead a mask-based approach is implemented, which means in addition to the fault space address and its time of occurrence, each fault point saves a bit-mask to indicate accessed bits. For each bit that is accessed during the read or write, the mask contains a one bit at the corresponding bit position. Each importer sets the mask according

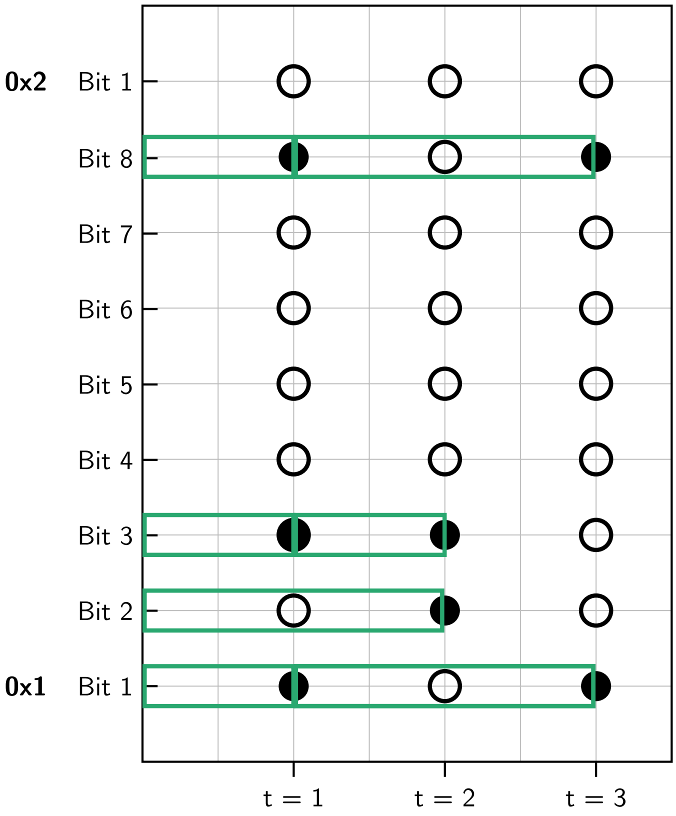


Figure 3.5 – Example for equivalence intervals in a masked byte-addressed fault space. Each interval is indicated by a green color.

```
input: $U = \{(address, mask, time), \ldots\}$ previous accesses $address$, $mask$ of the new access
output: $T = \{(mask, time), \ldots\}$ all accesses, which form an equivalence interval
  $T \leftarrow \varnothing$
  for all $(a, m, t) \in U$ where $a = address$ sorted descending by $t$ do
    if $mask \wedge m \neq 0$ then
      $T \leftarrow T \cup \{(m \wedge mask, t)\}$
      $mask \leftarrow mask \oplus m$
    end if
  end for
```

Algorithm 3.1 – FINDMATCHING

to the tracing information it received and the type of architectural element. For example, a fault point which refers tag memory will always have only the least-significant bit set, while a fault point referencing the virtual tag register, will have a bit set in its mask that corresponds to the injected capability register's tag bit. During pruning, this mask is then used to track reads and writes and subsequently build equivalence classes at bit-granularity. Additionally, during injection, only the bits marked as accessed in the fault point are injected.

Byte-sized reads and writes can then be treated similarly to the original fault space model by simply setting all bits in the mask. Thus, no additional information, beside their mask, must be tracked in the database for these fault points, which reduces the performance impact in comparison to a bit-addressed fault space. However, finding equivalence intervals is more complex in this fault space model. For a simple byte-addressed fault space with byte-sized fault points, each interval begins when an address is accessed and ends when it is accessed again at a later time. An algorithm to find these access intervals, when each such access is potentially masked, in contrast, must consider each accesses non-masked bits. Next, consider the example given in Figure 3.5. Here, the memory access at time $t = 1$, accesses the first three and the last bit of the fault point at fault space address `0x1`. As the memory access at $t = 2$ subsequently accesses bits two and three of the same byte of fault space, two equivalence intervals are created, which span with a length of 1. The access of bits one and eight, however, only create an equivalence interval when they are accessed at $t = 3$.

Consequently, when deriving equivalence intervals from tracing information, the implementation of this mask concept in FAIL* tracks accessed addresses and their respective not-yet-accessed-again bits. When an address is accessed the importer iterates the access masks of previous accesses to find equivalence intervals that will be closed by this access. Algorithm 3.1 is used to find matching previous accesses. It takes the set U, which contains previously recorded accesses, with corresponding address, time of occurrence and access mask, and the newly accessed address and access mask, which is used as a search mask. For each previous access, which matches the newly accessed address, it determines the overlap in their access mask. If they overlap, the overlap and the time of occurrence of the matching access is recorded. Additionally, the overlap is cleared from the newly accessed mask, which acts as search mask. This process continues until no further previous accesses have been recorded or the search mask has been cleared completely. If bits remain in the search mask, FAIL* returns an additional artificial $(mask, time)$-tuple which contains the remaining bits of the search mask, and the workloads' start time, i.e., the time at the encounter of *start_trace* marker (see Section 3.2), to emulate a write access outside the trace.

After deriving all matching previous accesses the importer validates them, combines their access mask, and uses the algorithm presented in Algorithm 3.2 to find all accesses, which are now shadowed by the new access. Going back to Figure 3.5, the access at $t = 3$ would purge the interval at $t = 1$, since all of its accessed bits have since been accessed or, in other words, shadowed. To determine

```
input: $U = \{(address, mask, time), \ldots\}$ previous accesses; $address$, $mask$, $time$ of the new access
output: $U'$ all accesses with uncompleted equivalence intervals
  $U' \leftarrow \varnothing$
  for all $(a, m, t) \in U$ where $a = address$ sorted descending by $t$ do
    $overlap \leftarrow m \wedge mask$
    $m' \leftarrow m \oplus overlap$
    $mask \leftarrow mask \oplus overlap$
    if $m' \neq 0$ then
      $U' \leftarrow U' \cup \{(a, m', t)\}$
    end if
  end for
  $U' \leftarrow U' \cup \{(address, mask, time)\}$
```

Algorithm 3.2 – PURGEMATCHING

which accesses must be purged the algorithm again iterates over all previous accesses and determines their overlap with the combined mask of all valid intervals. The overlapping bits are then cleared from the combined mask and the previous accesses' mask. If there are still bits left in the access mask of the previous access, it is then added to the set of still uncompleted accesses. Finally, the algorithm adds a new access to the list of recorded accesses, which contains the combined mask and the address and time of the current access.

To sum up, instead of assuming that an accessed byte is always read or written completely, a mask can be supplied, which marks accessed bits. This enables tracking of faults at bit granularity without the drawback of a purely bit-addressed fault space. This, mask-based, approach has been successfully used in this thesis to enable both tag bit fault points in memory and in capability registers.

3.3.3 CHERI-FAIL: Integration

After discussing the required extension to FAIL* to support the CHERI architecture, this section outlines the concrete integration of CHERI RISC-V into the FAIL* framework. Its integration is based on and inspired by Schotbruch [Bud20], which integrates a 32-bit RISC-V simulator into FAIL*. However, due to its recent development, a subset of its implementation, most notably its automatic interface generation, is unused in this thesis. Other parts have been extended or simplified. Similar to Schotbruch, the integration of CHERI RISC-V is based on its simulation model written in the Sail architectural description language [Armb]. To simulate such a model, the Sail compiler first translates the description into C code, which can be integrated into the FAIL* framework. A similar translation to OCaml code is available too, however, it is unused for this thesis due to its inherent incompatibility and simulation speed.

Both 32-bit and 64-bit variants of a RISC-V processor, which implements the *IMAC* specification, can be generated. Similarly, the CHERI RISC-V model can be generated for 32-bit and 64-bit instruction length, with 64-bit and 128-bit capabilities respectively. The capabilities are compressed with the CHERI Concrentrate representation (see Section 2.5.2.2). Additionally, it implements a similar *IMAC* specification, since its model is an extension of the RISC-V model.

Generally speaking, to integrate a new architectural simulator into the FAIL* framework, it must be adapted to the existing EEA interface. It consists of two distinct sub-interfaces (see Section 3.2), which must be implemented separately.

The first interface – the event-listener interface – is responsible for tracing architectural events. Registered listeners are tracked in the EEA abstraction, however, each simulator must call event callback functions exposed by the EEA abstraction at the appropriate times to notify it of occurred events. However, since the C simulator is generated from the Sail model, these calls must be integrated into the Sail model itself. For this, the Sail foreign-function interface is used, which allows calls to external C functions directly from the Sail model. To cover all possible registered listeners, the simulator must call a total of seven different event callback functions during its execution.

First, the simulator notifies FAIL* of any memory accesses that occur in the simulator. For RISC-V, it passes the type of access which occurred, i.e., read or write, the accessed address and the length of the access to the callback function. To distinguish between accesses to tag and data memory, this interface is extended for CHERI RISC-V to include the type of memory that has been accessed. This type is then saved to the trace, and can be used during import to map the memory access to the correct fault space area.

Additionally, it notifies FAIL* of any interrupts and traps, which have occurred in the processor. By default, only the pending interrupt or trap number is passed to FAIL*. In addition to the event callback, both models are extended with additional traps to signal error conditions in the model execution to FAIL*. Furthermore, the CHERI RISC-V is modified to map any internal error which occurs due to an invalid capability to an unused trap number. This eases later analysis of its failure modes.

Furthermore, it notifies FAIL* when the executed workload has written characters to the host-target interface (HTIF) available in RISC-V. FAIL* allows tracking of the output generated by a workload to further verify its correct execution.

Likewise, it notifies FAIL* when it reaches a new instruction and after it executed the instruction. This is used to track listeners, which wait for the execution to reach a certain instruction or, in other words, breakpoint listeners. The callback after instruction execution is used to track if the executed instruction was a jump instruction, which is required for certain pruning strategies. Both RISC-V and CHERI RISC-V use a opcode table to distinguish normal instructions from jump instructions.

The last callback function does not notify FAIL* of any event, but instead provides a synchronization point between FAIL* and the simulator. It switches the control flow back to the experiment co-routine (see Section 3.2 and Figure 3.2), and allows any previously queued save or restore requests to run. Careful consideration must be applied to the timing of its call: It must be called after FAIL* has been notified that an instruction has been reached, but before any processing of the current instruction has started. This ensures, that the correct state is saved, when the `start_marker` is reached and no previous state pollutes the simulators execution, when its state is restored.

The second interface of the EEA layer is the state interface. It facilitates access to the simulators, and consequently, the simulated processors internal state. The state of a Sail model can be split up into the simulated data and tag memory, and the processors registers.

First, consider the simulated memory. The Sail compiler provides a standard library, which provides often used functionality to the generated C simulator. Most notably, this standard library implements both data and tag memory as dynamically allocated linked-list structure. Both memories are exposed to the Sail model through a pair of read and write function, and can be accessed from the model through the foreign function call interface of Sail. FAIL*, consequently, uses the same functions to read and modify the simulated memory during injection. To save and restore the state of simulated memory, FAIL* simply serializes the linked-list structure to a file.

Secondly, consider the processors internal and external registers. Each register declared in a Sail model is a global, statically-typed variable. By default, Sail supports a multitude of data types, ranging from simple integers to bit-vectors and complex structs. During model compilation, the Sail compiler allocates a similarly named, albeit differently typed, static variable in the generated C code for each register declaration. This global variable can then be accessed in the EEA layer. However, special consideration must be given to the type of generated C variable.

Figure 3.6 shows three Sail register declarations and the corresponding generated C code. Depending on the size and type of the register, the Sail compiler will either emit a register which is backed by a primitive type (`test`), a register which is backed by an arbitrary precision number type (`test_long`), or a register which is backed by a struct (`test_comp`).

Internally, the EEA layer uses unsigned integer types to pass register values and thus, any Sail register value must be converted to an integer value before it can be handled by FAIL*. Registers, which are backed by primitive types, are unsigned integers and can therefore be read or written by simple assignment. Most registers declared in the RISC-V and CHERI RISC-V model fall into this category. This includes registers, which are internal to the model and must be saved or restored but cannot be injected. Struct registers, such as CHERI RISC-V capability register, on the other hand, must be serialized to an unsigned integer representation before passing through the EEA layer. Instead of implementing this conversion from scratch, CHERI RISC-V EEA implementation instead uses memory serialization functions from the CHERI RISC-Vs Sail model to convert the capability registers before access. However, since the tag bits are saved separately during memory reads and writes, they must also be considered separately for capability registers. Instead of storing them in-line with the register value all tag bits are mapped into a separate *virtual* tag register. This register can then be read and written, and thus injected, like any other register. Similarly to the memory state, each such register is serialized to and from a file during state save and restore respectively.

In addition to its adaption for the EEA layer of FAIL*, CHERI RISC-V requires modifications to the fault space exploration, i.e., the trace importing, of FAIL*. Both RISC-V and CHERI RISC-V use the virtual fault space abstraction discussed in Section 3.3.2.1. Their fault space contains three concrete fault space areas, a `register_area` and two `memory_areas`, which abstract registers and data or tag memory respectively. For each memory event, the importer then creates a fault space element in the corresponding memory area and the accessed address. It then records its globally unique address and access mask for the injection procedure. Similarly, the register importer creates a fault space element in the register area, which references the accessed register. For CHERI RISC-V it additionally inserts an artificial access to the virtual tag register, which only reads or writes the tag bit that corresponds to the accessed register. This single-bit access is facilitated by setting the fault points masks accordingly (see Section 3.3.2.2).

In summary, the integration of CHERI RISC-V into FAIL* largely follows the existing integration of the RISC-V Sail model. However, the additional tag bit for both register and memory required special consideration. Nonetheless, a successful integration was possible due to the virtual fault space extension presented in Section 3.3.2.1 and the additional granularity for the fault space presented in Section 3.3.2.2.

```
/* Sail model code */

register test      : bits(64)
register test_long : bits(128)
struct Complex = {
    real: bits(64),
    imag: bits(64)
}
register test_comp : Complex
```

```
/* Generated C code */

uint64_t ztest;
lbits ztest_long;
struct zComplex {
    uint64_t zreal;
    uint64_t zimag;
};
struct zComplex ztest_comp;
```

Figure 3.6 – Sail register declarations with the corresponding generated C code for differently typed registers. `lbits` is a multi precision number type defined by the Sail standard library.

3.4 Summary

This chapter first outlined, how a memory-protected architecture could improve the soft-error resilience of a system. Then, it presented the FAIL* FI framework, which can be used to evaluate the soft-error resilience of an architecture. Next, it presented the concrete memory protection architecture – CHERI RISC-V – and its integration into the FAIL* framework. Together, with the existing integration of RISC-V, this form a comprehensive evaluation platform on which the memory-protected architecture can be evaluated. However, CHERI RISC-V's tagged memory architecture posed unique challenges for its integration. These have been tackled by developing a virtual fault space abstraction, which can abstract into a globally unique fault space regardless of their corresponding architectural element. Additionally, FAIL* is extended to support fault point at bit-granularity instead of byte-granularity to track writes to tag bits in registers and memory without skewing the FI results.

ANALYSIS

4

This chapter evaluates the memory-protected CHERI RISC-V and its unprotected counterpart RISC-V to compare their resilience against soft-errors. First, Section 4.1 defines the fault model used for the evaluation. Next, Section 4.2 and Section 4.2.3 present two micro benchmarks, with various variants, designed to exercise the spatial memory safety and control-flow protection of CHERI. In combination, they cover large parts of the system's functionality, and allow conclusion on its resilience. They are then evaluated according to Section 4.3 to characterize each architectures behavior. Finally, Section 4.4 discusses the results to validate or refute the hypotheses made in Section 3.1.

4.1 Fault Model

For this evaluation, I, first and foremost, assume that faults can only occur or first become visible at the ISA level of an architecture. Furthermore, only faults which occur in the system's data memory or its registers including their respective tag bits are considered. Control logic, on the other hand, is assumed to be resistant against faults due to its inherent masking of transient irregularities [Hen+13; DW11].

From all available registers, only consider general-purpose registers are considered. More specifically, RISC-V's control-and-status registers (CSRs) are considered to be immune against faults. They often do not act as mere storage elements and instead forward architectural state to the user. Although research shows that SRAM and DRAM memories have different soft-error rates [Sla11] they vary significantly for different architectural parameters and implementations. Therefore, it is assumed for this thesis that a fault in either type of system memory is equally likely and their resulting failures can be combined without additional weighting to reach conclusions for the complete systems.

Furthermore, the evaluation follows the "single-fault assumption" postulated by Schirmeier [Sch16]. In short, it is assumed that during each system run, only a single fault will happen, which is transient and is only visible for a single cycle.

Each fault leads to a, possibly erroneous, system state, which are classified along three classes of errors.

1. *SDC* error, meaning that no error has been reported but the system produced a wrong result.

2. *TIMEOUT* error, meaning that no error has been reported, but the system failed to produce a result within a predetermined timeframe.

3. *TRAP* error, meaning that the system reported an unexpected trap event.

Translating each error class to failure modes is simple for most error classes. *SDC* errors correspond to unsignaled content failures, and *TIMEOUT* errors correspond to late timing failures. Finally, *TRAP* errors associated failure mode is dependent on the failure model of the system. Assuming a fault tolerant system, such errors could be recovered from and they therefore do not constitute any failure. However, for non-tolerant system they correspond to unsignaled halt failures.

4.2 Benchmarks

To evaluate each architecture thoroughly, two micro benchmarks written in the C programming language, with a total of five different implementations, are presented that each exercise a specific aspect of the target. Together, they provide an overview of the system's behavior for different classes of programs under the influence of soft-errors and allow conclusions for the real-world soft-error resilience of the evaluated architectures. In addition to the distinct implementations of each benchmark, four compile-time variants of each benchmark, which differ in compilation flags and/or memory layout, are generated to increase the generalizability of the results. In the following, both benchmarks and their respective implementation variants are discussed in Section 4.2.1 and Section 4.2.2 respectively. Finally, Section 4.2.3 elaborates on the compile-time variants, which are generated for each implementation.

4.2.1 The fibonacci Benchmark

The fibonacci benchmark computes the eights Fibonacci number. Two implementations of the algorithm are evaluated, one of which calculates the result through recursion (`fib`), and the other through iteration (`fib-iter`).

Generally, recursive computations allocate a multitude of stack frames due to repeated function invocation, whereas iterative computations often only allocate a single stack frame. Each allocated stack frame contains, next to the invocated functions' arguments and spilled registers, control-flow related information such as the return address and a pointer to the previous stack frame. Transient faults that affect the stack memory can, therefore, affect the control flow of the affected system by corrupting saved frame pointer or return addresses. This is especially true, if the stack mostly contains control-flow related information, i.e., the recursive function takes few arguments and keeps a small local state which may needs to be spilled before the next recursive invocation.

The fibonacci algorithm is a simple and stateless recursive algorithm. In its recursive implementation, the recursive function takes only a single argument and keeps no local state. Therefore, its allocated stack memory contains mostly control-flow related information. The iterative implementation of the same algorithm provides a baseline, which calculates the same result without keeping any control-flow related information on the stack.
Expected Result: The recursive fibonacci implementation will show fewer *TIMEOUT* errors than the iterative implementation.

4.2.2 The bubblesort Benchmark

The bubblesort benchmark sorts a sequence of numbers according to the bubble sort algorithm. It repeatedly loops over the sequence and swaps two adjacent entries if the value of the latter is lower than at the former. This is continued until the end of the sequence is reached, and then repeated from the front until no swaps have been performed for the full length of the sequence and, therefore, the sequence has been sorted. The resulting sequence is then compared to a presorted version of the input sequence, to check if it was sorted correctly by the algorithm.

The input array contains ten numbers ranging from $(0, 2^{32} - 1)$ for 32-bit systems and $(0, 2^{64} - 1)$ for 64-bit systems, and is, similarly to the presorted array, generated pseudo-randomly.

Three different implementations of this algorithm exist, which keep the same sorting algorithm but choose a different implementation of the data structure which back the abstract sequence.

The *static* variant implements the sequence as a C array. To swap two elements, it simply stores the former element's value in a temporary variable, moves the latters value into the formers array entry and finally moves the temporary variable into the latter array entry. The array is statically allocated and filled at compile time.

The *single* variant, on the other hand, implements the sequence as a single-linked list. Two elements can then be swapped, by modifying their and potentially their neighbors' forward link pointers to change the structure of the sequence.

Finally, the *double* variant, implements the sequence as a double-linked list. In contrast to the single-linked list implementation, both the forward and backward pointer of both nodes, and potentially these of their neighbors', must be swapped.

Both the single and the double variant allocate their linked-list dynamically at the start of the workload. However, the allocation is not injected during the evaluation, as to not skew the results in comparison to the baseline.

While the static variant uses no, or very few pointers to access the array's values, both the single and the double variant handle multiple pointers to list elements with varying levels of indirection. In other words: The level of indirection, i.e., the use of pointers, increases with each implementation. As discussed in Section 3.1, indirect memory accesses should be better protected in CHERI.
Expected Result: The single and double variant will show fewer *SDC* errors than the static implementation. Additionally, the double variant will show even fewer *SDC* errors than the single variant.

4.2.3 Variants

Each of the previously discussed benchmark implementations is evaluated in four compile-time variants. The first two relate to the compiler's optimization flags. Typically, benchmarks evaluated with FAIL* are compiled without compiler optimization. This makes it easy to map assembler instructions back to source-code lines, when performing the post-injection analysis. Additionally, it prevents the compiler from optimizing the program's structure and which skews the benchmark's fault space. However, the compiler used to compile the benchmarks for CHERI RISC-V (and consequently also for RISC-V) is `llvm-cheri`, a Clang-based research compiler. Without optimization, it generates correct but very inefficient code for CHERI-enabled architectures, which artificially inflates the fault space of these architectures. For some benchmark variants this inflation is up to 440 percent when comparing CHERI RISC-V to RISC-V. Consequently, to aid comparability, each benchmark is built with and without enabled optimization to evaluate the impact of the default code generation.

The other two variants modify the memory layout of the compiled binary. A pointer in a non-memory protected architecture that has been corrupted by a transient fault can still be used to access data, if the newly pointed to address is still a valid, albeit incorrect, address. If it points to non-accessible or invalid memory addresses, e.g., device memory, even an unprotected architecture is able to detect the invalid access. For a given pointer, the number of incorrect but valid corrupted addresses is directly proportional to the size of memory available to its program. Assuming that typically, only that data section of a program is both readable and writeable, this pool of corrupted but valid pointers is comparatively small for small input data sizes, and thus for the presented benchmarks.

However, real-world implementations of the presented benchmarks would be embedded into a larger system, and thus have a much larger pool of corrupted, but valid, pointer available. Consequently, evaluating each algorithm in a micro-benchmark scenario will yield less silent data corruption, than it would when implemented in a commercial system. To consider this effect, each benchmark is evaluated in a padded version, where its data section has been artifically enlarged through link-time modifications, and as an unpadded version where the original memory layout is preserved.

4.3 Evaluation Procedure

Through implementation and compile-time variants, a total of twenty different benchmarks are evaluated. Each benchmark is evaluated for the baseline RISC-V architecture and its memory-protected variant CHERI RISC-V in both 32-bit and 64-bit configurations. Evaluating both instruction lengths is of interest because the CHERIConcentrate capability encoding (see Figure 2.4b) is dependent on the instruction length. While the length of capability is always fixed to twice the instruction length of the architecture, the size of its internal fields is not. For example, its permission field varies in the number of permissions supported by the specific CHERI RISC-V implementation. Additionally, the length of its compressed bounds fields can be varied, where shorter bound encodings require stricter alignments and, thus, increase object granularity. This changes the heap layout of the workload since its objects must fulfill stricter alignment guarantees. However, by shortening the bounds, additional permission bits can be added or the range of object types can be extended. Therefore the precise allocation of bits in the capability encoding is able to change the runtime behavior of the workload and must be evaluated. Additionally, an extended version of the CHERI RISC-V architecture, henceforth called CHERI-P that uses parity to protect capabilities in memory and registers is evaluated in both instruction length configurations. It is discussed in greater detail in Section 4.4.1.3

Both memory and register, including their corresponding tag bits, are injected separately through the FAIL* framework. For each benchmark the complete fault space is evaluated. However, to speed up the evaluation, its size is reduced before injection with the *def/use pruning* strategy discussed in Section 3.2. For each of these injected faults the resulting error class and the delay between the injection and the resulting system state – the detection delay – are recorded in a database. To account for the pruning, each result is then weighted with the length of its equivalence class. Finally, the weighted results are grouped by their error class and accumulated to form each error class's *absolute error count*. The fault coverage of each system is not compared since it is an unreliable metric when comparing workloads with different fault space sizes [Sch16].

For CHERI RISC-V, each benchmark is compiled in *pure-capability* mode, i.e., all pointers are capabilities (see Section 2.5.2). Furthermore, all benchmarks are compiled for the RISC-V's `medany` code model, are not position-independent and statically linked without any standard library. They are, instead, amended with a small boot code that only initializes the heap and stack memory before calling the main benchmark function. For CHERI RISC-V this bootcode, additionally, performs the necessary initialization of the capability system, such as setting up the DDC registers and PCC registers. Finally, optimized variants of the benchmarks are compiled with loop-unrolling and automatic function inlining disabled. This way, any optimization by the compiler do not dramatically alter the structure of the program, and skew the results. Additionally, with loop-unrolling enabled, the compiler sometimes resolves the bubblesort algorithm statically on RISC-V, which collapses the fault space of the benchmark.

All experiments are conducted on a 48-core (96 hardware threads) Intel Xeon Gold 6252 machine clocked at 2.10 GHz with 374 GiB of main memory.

4.4 Results

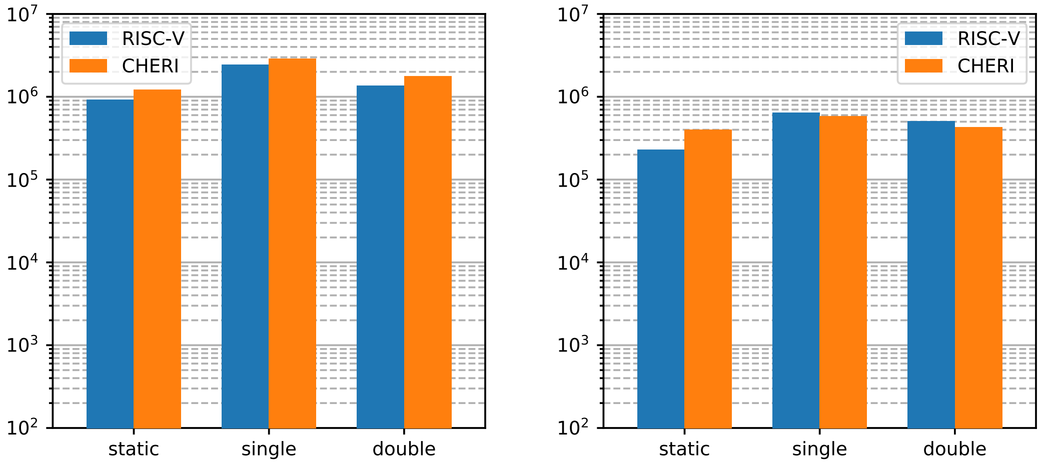
In the following, the results of the evaluation are presented and used to reject or validate the hypotheses introduced in Section 3.1. First, it is evaluated if CHERI RISC-V exhibits a reduced frequency of two failure modes. Section 4.4.1 discusses if and when the added protection measures of CHERI RISC-V lead to less unsignaled content failures. It specifically analyses, how different levels of indirection, compilation flags, and architectural parameters affect the failure distribution for the `bubblesort` benchmark. Next, Section 4.4.2 explores the effect of CHERI RISC-V's control-flow protective measures with regards to late timing failures. Finally, Section 4.4.3 evaluates if CHERI RISC-V offers improved detection of specific failure modes.

4.4.1 Hypothesis: Reduced Frequency of Unsignaled Content Failures

As discussed in Section 3.1, memory-protected architectures generally better protect programs that use a lot of indirection when accessing data. Consequently, CHERI RISC-V programs that make heavy use of indirection should exhibit less unsignaled content failures, in comparison to programs that use few or no indirection.

Figure 4.1a shows the number of faults that result in *SDC* errors for each architecture and each of the three implementation variants of the bubblesort benchmark. All variants are built without optimizations or memory padding. Additionally, the 32-bit variant of each architecture is evaluated. The figure shows combined absolute error counts, i.e., the number of *SDC* errors originating from register and memory faults have been summed up and are weighted by the length of their equivalence interval.

For the static variant, CHERI RISC-V shows a 31 percent increase in *SDC* errors when combining both fault classes and in comparison to RISC-V. However, when considering register faults separately,



(a) Number of *SDC* errors for each variant compiled *without* optimizations

(b) Number of *SDC* errors for each variant compiled *with* optimizations

Figure 4.1 – Number of faults that result in a *SDC* error for each bubblesort implementation variant and architecture. All variants are built without memory padding and evaluated on 32-bit architecture variants of CHERI RISC-V and RISC-V.

SDC errors decrease by 32 percent. The overall increase is, therefore, mostly due to memory faults, which show a 36 percent increase in *SDC* errors.

The single benchmark variant, similarly, shows an overall increase of 19 percent in *SDC* errors, when considering combined faults. However, in comparison to the static variant the overhead of CHERI RISC-V is reduced by nearly half, most likely due to the additional indirection in the single variant. That being said, both architectures show a general increase in *SDC* errors for the single variant (150 percent on average), most likely due to the added complexity of the linked-list handling. Regarding the error difference between memory and register faults, CHERI RISC-V shows an increase of both *SDC* errors which originate from register faults (+13 percent) and memory faults (+19 percent) in comparison to RISC-V for the single variant.

Although the level of indirection is increased, the double variant shows a larger overall increase of 30 percent in *SDC* errors than the single variant, when considering combined faults in comparison to RISC-V. Nonentheless, the overhead is still smaller than that of the static variant. Again, the added complexity of the double linked-list increases the number of *SDC* errors by 47 percent on average. This increase is smaller than the single variant since the bubble sort algorithm requires fewer traversals of its list to swap adjacent elements. Similarly to the static variant, *SDC* errors caused by register faults are decreased by 22 percent, and *SDC* errors caused by memory faults increased by 30 percent.

The unexpected inversion in *SDC* error overhead with increasing indirection and the overall increase in *SDC* errors for CHERI RISC-V can be explained with the large deviation of fault space size between the architectures. While the complete fault space for the static variant in RISC-V consists of 1,736,768 unique fault points, the same variant has 6,539,752 unique faults points in CHERI RISC-V. In other words, compiling the static variant for CHERI RISC-V enlarges its fault space by a factor of 3.77.

The single and double variants show a similar increase. Moreover, 70 percent of the additional fault points for CHERI RISC-V are tolerated faults, i.e., result in the *OK* error when injected. One possible explanation for the large fault space increase with mostly no-effect faults could be that the compiler generates highly redundant, but easily (human-)verifiable code, for CHERI RISC-V. For a research compiler, like `llvm-cheri`, this is often convenient during its development, since the emitted code is often verified manually. The RISC-V compiler, on the other hand, is used in production and has been tested extensively. It, therefore, emits less readable or verifiable, but better-optimized code by default.

Next, in Section 4.4.1.1, discusses one mitigation for the fault space inflation – compiler optimization –, and its effect on the relative increase of *SDC* errors when comparing CHERI RISC-V to RISC-V. Section 4.4.1.2 explores the effect of the workload's memory layout and the instruction length of the evaluated architecture on the number of *SDC* errors. Finally, Section 4.4.1.3 discusses the CHERI-P architecture, which extends CHERI RISC-V with parity-protected capabilities, and its effect on the frequency of unsignaled content failures.

4.4.1.1 Effect of Compiler optimization

Benchmarks compiled for CHERI RISC-V have a much larger fault space than the same benchmark compiled for RISC-V. Table 4.1 summarizes this size increase for each variant of the bubblesort benchmark. It presents the fault space sizes for the same benchmark, once with compiler optimizations disabled and once with compiler optimizations enabled (marked (opt.)). Additionally, the relative increase for each CHERI RISC-V variant is recorded in a separate column (marked Diff.). For unoptimized variants of the bubblesort benchmark, compiling the same benchmark for CHERI RISC-V increases the fault space by 250 percent on average. The overhead is the highest for

Variant	RISC-V	CHERI	Diff. [%]	RISC-V (opt.)	CHERI (opt.)	Diff. [%]
static	1,736,768	6,539,752	+276.5	381,888	1,051,960	+175.5
single	5,404,032	17,930,930	+231.8	1,101,536	1,762,670	+60.0
double	5,597,888	19,119,132	+241.5	1,854,848	2,832,000	+52.7

Table 4.1 – Number of unique fault points for RISC-V and CHERI RISC-V for optimzed and non-optimized variants of the bubblesort benchmark.

the static benchmark variant. However, all variants generate a percentual increase which is close to the calculated average. The optimized compilation of each benchmark, on the other hand, only shows a 96 percent increase in fault space size on average. Of the three, the static variant stands out with a 175 percent increase in fault space size for CHERI RISC-V, whereas the other two only show a 55 percent average increase. With optimizations, the compiler emits completely stackless code for the static bubblesort implementation for RISC-V, but is unable to do so when compiling for CHERI RISC-V. For RISC-V, memory access is, therefore, restricted to the benchmarks sorting input, and significantly reduced in comparison to CHERI RISC-V and its non-optimized compilation. Hence, the size of its fault space is unproportionally decreased, which explains the large overhead for CHERI RISC-V. Nonetheless, compiling the benchmarks with optimizations still halves the fault space size overhead on average.

Going back to Figure 4.1b, the number of faults that result in *SDC* errors is shown for all benchmarks with enabled optimizations. For the static variant, the previously recorded increase in *SDC* errors is even larger (+75 percent) when it is compiled with enabled optimizations. This can be attributed to the collapsed fault space of its RISC-V variant. However, both the single and double variant now show an overall decrease in *SDC* errors (9 percent and 16 percent respectively). More importantly, this overall decrease in *SDC* errors increases with the amount of indirection, as predicted in Section 3.1.

In summary, compiler optimizations are an effective mitigation technique for the fault space inflation of CHERI RISC-V. Additionally, a comparable fault space size is important when comparing different architectures. For roughly equally sized fault space CHERI RISC-V shows an overall decrease in *SDC* errors for two of the three benchmark variants. Furthermore, this decrease is higher for a workload, which uses more indirection. To conclude CHERI RISC-V shows less unsignaled content failures for workloads which use indirection, however, its benefits can be masked by compilation artifacts.

4.4.1.2 Effect of Instruction Length and memory padding

As discussed in Section 4.2.3, running the same benchmark on different instruction length variants of CHERI RISC-V can change its runtime behavior and memory layout. The CHERI Concretate encoding supports various allocations of the available capability bits to its internal fields. Most importantly, the 32-bit variant of CHERI RISC-V allocates fewer bits to the capabilities' bounds than its 64-bit counterpart. Consequently, it requires stronger alignment for objects accessed through capabilities. This affects the heap layout and, thus, the workload's behavior when injected. Additionally, the 64-bit variant of CHERI RISC-V has more unused, that is reserved, bits in its capability encoding. These bits will always produce a no-effect error, i.e., *OK* error (see Section 4.1).

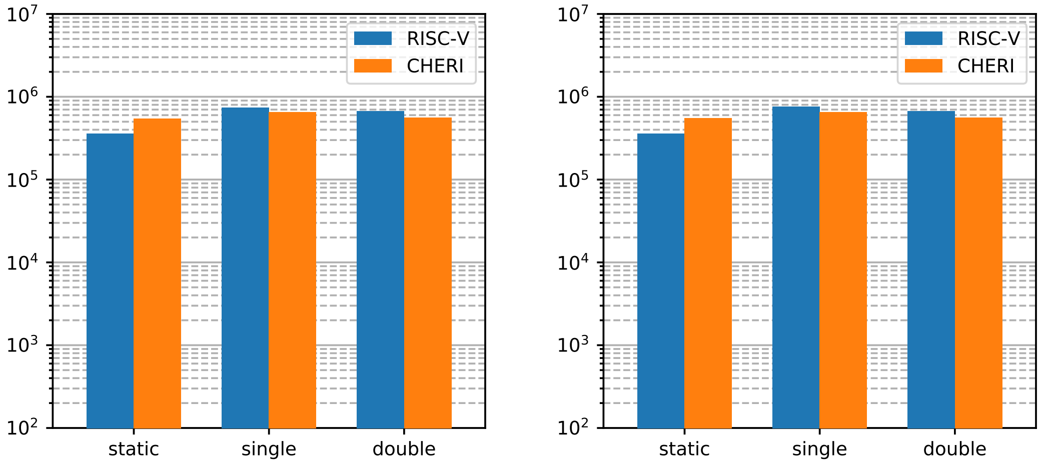
Figure 4.2a shows the number of faults that result in *SDC* errors for each architecture and each of the three implementation variants of the bubblesort benchmark. All variants are built with optimizations to avoid masking any effects of CHERI RISC-V (see Section 4.4.1.1), but without memory padding. To discuss the effect of the instruction length 64-bit variants of each architecture are evaluated.

In comparison to their 32-bit architecture variant, both CHERI RISC-V and RISC-V only show an increase of 31 percent in *SDC* errors averaged over all benchmark variants for its 64-bit architecture variant. Their relative *SDC* errors follow a pattern similar to the 32-bit variant: For the static variant, *SDC* errors are increased by 52 percent, while the single and double variants show decreased *SDC* errors (−11 percent and −16 percent respectively), when comparing CHERI RISC-V to RISC-V. To conclude, most of the additional bits in each data word do not lead to additional silent data corruption. Therefore, CHERI RISC-V reduces the frequency of unsignaled content failures, regardless of instruction length.

Similarly to the alignment changes with 64-bit architectures memory padding changes the memory layout of the benchmark. This can can affect the number of faults that lead to *SDC* errors on RISC-V. Figure 4.2b shows the number of faults that result in *SDC* errors for each architecture and each of the three implementation variants of the bubblesort benchmark with enabled memory padding. Again, all variants are built with optimizations to avoid masking any effects of CHERI RISC-V (see Section 4.4.1.1). 64-bit variants of each architecture are evaluated, allowing a better comparison to the unpadded variants presented in Figure 4.2a. However, all results discussed for the memory padding apply similarly to the 32-bit variants of each architecture.

As expected, RISC-V shows a slight overall increase in *SDC* errors (+1.14 percent), whereas CHERI RISC-V shows almost no overall increase (< 0.1 percent). The largest difference is detected for the single variant. Here, CHERI RISC-V shows a 14 percent decrease in *SDC* errors compared to RISC-V. Its unpadded variant, however, only showed a 11 percent decrease. Both the static and double variants, on the other hand, show no difference in the frequency of *SDC* errors, compared to their unpadded variants.

To summarize, the memory layout of a benchmark has a mostly neglible impact on frequency of unsignaled content failures. As expected, it increases the amount of *SDC* errors for the unprotected



(a) Number of *SDC* errors for each variant compiled *without* memory padding

(b) Number of *SDC* errors for each variant compiled *with* memory padding

Figure 4.2 – Number of faults that result in a *SDC* error for each bubblesort implementation variant and architecture. All variants are built with optimizations and are evaluated on *64-bit* architecture variants of CHERI RISC-V and RISC-V.

architectures. However, this increase is too small on average and only translates to improvements for the single variant. Therefore, in conclusion the memory layout has a small non-deterministic effect on silent data corruptions, and it should not be generally assumed that*SDC* errors are reduced for larger binaries in CHERI RISC-V.

4.4.1.3 Effects of parity-protected capabilities

In contrast to other CHERI-enabled architectures, CHERI RISC-V uses the compressed CHERI Concentrate encoding exclusively for its capabilities (see Figure 2.4b). A single-bit fault in the loaded capability, can either affect its flags, its tag, its object type, its bounds, or its base address. While faults in the bounds, flags or tags often lead to a trap, since their validity is checked before usage, faults which occur in the base address are much harder to detect. In the CHERI Concentrate representation, a capabilities' base and bounds are encoded *relative* to its base address and are still valid even when it changes. In other words, a fault that changes the base address of a capability, simply moves the pointed-to memory area while retaining valid bounds. In this case, the CHERI protection-model is unable to detect the capability corruption because its encoding is still valid. In consequence, its ability to detect faults is restricted. However, such faults could be detected, if each capability had a simple checksum mechanism, which could detect such single-bit changes to its fields.

Therefore, the CHERI RISC-V architecture was extended to add a per-capability parity bit. This extension is called CHERI-P in the following. When CHERI-P writes a capability to memory or into a register this calculates an even parity bit. It is `1` if there is an even number of ones in the memory representation of the capability and `0` otherwise. The parity bit is saved to an unused bit of the capability, before writing the modified version to memory. During a memory or register read which accesses a capability, this tag bit is then used to verifiy the integrity of the capability. If the integrity check fails, a trap is raised, which results in a *TRAP* error.

Figure 4.3 shows the number of faults that lead to *SDC* errors for each bubblesort implementation variant and the RISC-V and CHERI-P architecture. All benchmark variants are built with optimizations, without padding and for 32-bit variants of each benchmark.

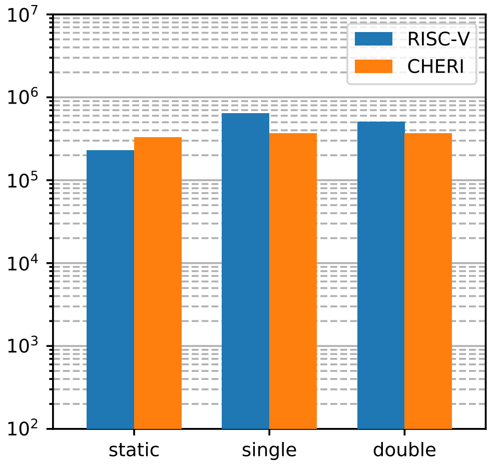


Figure 4.3 – Number of faults that result in a *SDC* error for each bubblesort implementation variant and the RISC-V and *parity-protected* CHERI-P architecture. All variants are built with optimizations and without padding and are evaluated on 32-bit architecture variants.

For the static variant, CHERI-P shows a decreased overhead of 45 percent in *SDC* errors in comparison to RISC-V and compared to optimized benchmark variant on non-parity protected CHERI RISC-V. Without parity-protection, the static variant shows a 75 percent increase.

For the single variant, CHERI-P decreases the number of *SDC* errors by 42 percent, which nearly quadruples the effect of the non-parity protected CHERI RISC-V (9 percent). With additional memory padding, this can be further improved to a 45 percent decrease in *SDC* errors.

The double benchmark also shows an additional decrease of CHERI-P (up to 28 percent) in *SDC* errors in comparison to RISC-V. The non-parity protected CHERI RISC-V, on the other hand, only showed a 16 percent decrease. However, in comparison to the single variant, the improvement achieved by parity-protection is smaller.

In summary, adding parity-protection to CHERI RISC-V's capabilties significantly reduces the number of *SDC* errors in comparison to RISC-V. Especially for the single benchmark variant, this reduction nearly halves the number of unsignaled content failures in CHERI RISC-V. Therefore, CHERI's capabilities should be parity-protected, if used in a dependable system.

4.4.1.4 Summary

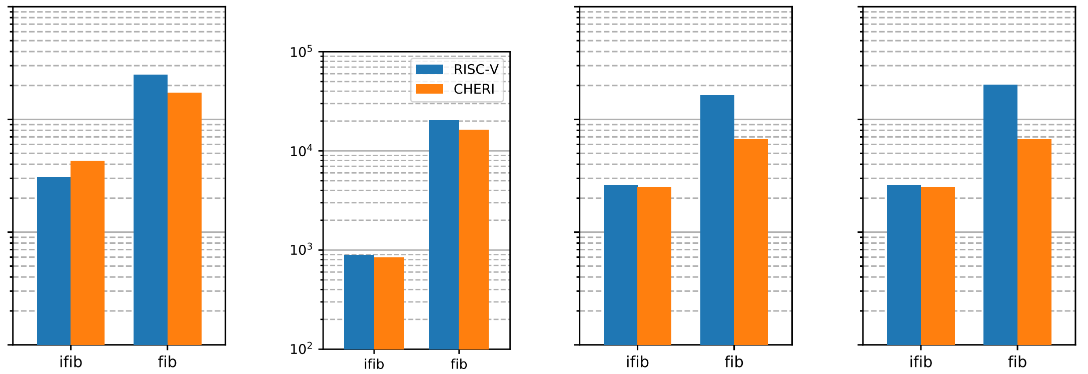
In summary, CHERI RISC-V reduces the frequency of unsignaled content failures in most cases. However, special consideration must be given to the compilation options of its workloads. If it is compiled without optimizations enabled, CHERI RISC-V's fault space inflation will often mask its reduction in silent data corruption. Workloads that use more indirection benefit more from the added protection of the CHERI-protection model. At most CHERI RISC-V reduces the frequency of *SDC* errors by 16 percent for the bubblesort benchmark variant. The instruction length of the CHERI architecture and the workloads memory layout, on the other hand, have a negligible influence on the soft-error resilience of CHERI RISC-V. Additionally, the influence of the compressed capability encoding on the vulnerabilty of the CHERI RISC-V architecture has been discussed. Protecting each capability with an additional parity can significantly improve the recorded decrease in unsignaled content failures for CHERI RISC-V. For the single variant, CHERI-P nearly halves the frequency of unsignaled content failures in comparison to the unprotected RISC-V. In conclusion, CHERI-P is even more resilient to soft-errors than CHERI RISC-V with regard to unsignaled content failures.

4.4.2 Hypothesis: Reduced Frequency of Late Timing Failures

As discussed in Section 3.1 CHERI RISC-V should offer better protection against late timing failures due to the added protection of control-flow related information in memory. Consequently, benchmarks which store a lot of control-flow information, such as loop-counters or return addresses in memory should experience fewer late timing failures.

Figure 4.4a shows the number of faults that result in a *TIMEOUT* error, for the two implementation variants of the fibonacci benchmark and both architectures. Similar to the bubblesort benchmark, memory and register faults are weighted according to the length of their equivalence interval and summed up. Again, a uniform distribution of faults across both register and memory is assumed. Both variants are compiled without optimizations or memory paddings, and are evaluated on 32-bit variants of each architecture.

Without optimizations, CHERI RISC-V shows a 40 percent increase in *TIMEOUT* errors for the iterative variants in comparison to RISC-V. This can be explained by the fact that only its loop-counter variable is control-flow related and could provide a benefit regarding *TIMEOUT* errors in CHERI RISC-V. Additionally, the non-optimized variants shows a enlarged fault space compared to the optimized variants. While the non-optimized benchmark's fault space contains 54,912



(a) Number of *TIMEOUT* errors for each variant, compiled *without* optimizations.

(b) Number of *TIMEOUT* errors for each variant, compiled *with* optimizations.

(c) Number of *TIMEOUT* errors for each variant, compiled with optimizations, and evaluated on 64-bit architectures.

(d) Number of *TIMEOUT* errors for each variant, compiled with optimizations and padding, and evaluated on 64-bit architectures.

Figure 4.4 – Number of faults that result in a *TIMEOUT* error for each fib implementation variant and various architecture variants. *ifib* refers to the iterative implemention and *fib* to the recursive implementation.

unique fault points for the RISC-V architecture, the same benchmark has 299,079 fault points for CHERI RISC-V, constituting an increase by factor 5.45. Similar to *SDC* errors (see Section 4.4.1.1), this inflation might shadow any benefits provided by CHERI RISC-V.

The recursive variant, however, shows a 31 percent decrease in *TIMEOUT* errors when comparing CHERI RISC-V to RISC-V. In contrast to the iterative implementation, most of its memory content is control-flow related. Namely, its stack stores the return addresses for the recursively called function. Similar to the iterative implementation its fault space is increased by a factor of 4.19. The overall decrease in *TIMEOUT* errors leads to the conclusion that the benefit provided by the additional protection of the return addresses is significant and outweighs the fault space size increase.

In summary, the decrease in *TIMEOUT* errors is related to the amount of control-flow related information which a program keeps in memory. For recursion heavy programs CHERI RISC-V shows a reduced frequency of *TIMEOUT* errors and, in turn, late timing failures, even when its fault space is heavily inflated in comparison to RISC-V. In the following, the number of *TIMEOUT* errors are evaluated for different compilation variants and architecture variants. First, Section 4.4.2.1 discusses the effect of compiler-optimizations, which successfully mitigated the fault space inflation for *SDC* errors (see Section 4.4.1.1). Then, Section 4.4.2.2 discusses the effect of instruction length and memory padding on the number of late timing failures. Finally, Section 4.4.2.3 evaluates the CHERI-P architectural variant with regards to *TIMEOUT* errors, after which Section 4.4.2.4 summarizes the results.

4.4.2.1 Effect of optimization

Table 4.2 shows the size of the fault space for optimized (marked (opt.)) and unoptimized variants of the fibonacci benchmark. Additionally to their absolute sizes, I record the relative difference (named Diff.) between CHERI RISC-V and RISC-V. Similar to the bubblesort benchmark, CHERI RISC-V increases the size of the fault space by 382 percent on average for unoptimized builds. For these the

Variant	RISC-V	CHERI	Diff. [%]	RISC-V (opt.)	CHERI (opt.)	Diff. [%]
fib	2,303,488	9,644,670	+318.7	1,076,480	3,325,560	+208.9
fib-iter	54,912	299,079	+444.7	9,216	20,727	+124.9

Table 4.2 – Number of unique fault points for RISC-V and CHERI RISC-V for optimzed and non-optimized variants of the fibonacci benchmark.

inflation decreases to 167 percent on average. In other words, building a benchmark with enabled optimizatons halves its fault space inflation.

Going back to Figure 4.4b, both variants are evaluated with enabled optimizations. The iterative variant benefits from enabled optimizations and shows a 6 percent decrease in *TIMEOUT* errors. The recursive variant, however, shows a slightly lower decrease in *TIMEOUT* errors (20 percent) with optimizations than without optimizations, where it decreased *TIMEOUT* errors by 31 percent . This decrease can be attributed to large reduction in *TIMEOUT* errors for RISC-V (−18 percent) in comparison to the small reduction for CHERI RISC-V (−6 percent). This, in turn, might be due to the badly optimized default code generation for recursive code by the RISC-V compiler.

In contrast to Section 4.4.1.1, enabling optimizations does not result in less frequent *TIMEOUT* errors, and thus late timing failures in CHERI RISC-V for workloads, which keep most control-flow related information in memory. Nonetheless, the fault space inflation is halved for optimized builds. To better distinguish the effects of CHERIs additional protection, only optimized are evaluated variants further.

4.4.2.2 Effect of instruction length and memory padding

As discussed in Section 4.2.3 and Section 4.4.1.2, evaluating the same benchmark for different instruction length variants of CHERI RISC-V can change its runtime behavior and memory layout.

Figure 4.4c shows the number of faults, which result in *TIMEOUT* errors for all variants of the fibonacci benchmark and 64-bit variants of RISC-V and CHERI RISC-V. All variants are built with optimizations and without padding.

For the iterative variant, CHERI RISC-V again shows a slight decrease of 4 percent in *TIMEOUT* errors in comparison to RISC-V. The recursive variant, however, shows a large decrease in *TIMEOUT* errors for the 64-bit variant of CHERI RISC-V. Compared to 64-bit RISC-V, it decreases *TIMEOUT* errors by 59 percent , whereas this decrease was only 20 percent for its 32-bit variant. Overall, the increased instruction length leads to a 51 percent increase in *TIMEOUT* errors averaged over both benchmark variants and architectures. Again, this can be attributed to additional no-effect bits in the 128-bit capabilities used for CHERI RISC-V, and the additional alignment requirements for its capabilities.

To summarize, CHERI RISC-V shows less late timing failures when a lot of control-flow related information is saved in memory in both 32- and 64-bit architecture variants. However, the reduction is nearly tripled for the 64-bit variant.

As discussed in Section 4.2.3, increasing the pool of valid, but incorrect addresses increases the number of undetectable faults for unprotected architectures. This effect can only provide a benefit for CHERI RISC-V when the workload uses pointers. However, the return addresses, which are stored in memory, can be seen as a form of a control-flow pointer. Figure 4.4d shows the number of faults that lead to a *TIMEOUT* error, for both benchmark variants, built with optimizations and memory padding enabled. They are evaluated for 64-bit variants of each architecture.

As expected, evaluating the recursive variant with enabled memory padding increases the number of *TIMEOUT* errors recorded for RISC-V by 24 percent. This translates to a 67 percent decrease in recorded *TIMEOUT* errors when comparing CHERI RISC-V to RISC-V, which is higher than the decrease for the 64-bit variant without padding (59 percent as shown in Figure 4.4c). A similar result is found, when 32-bit variants of each architecture are evaluated. Here, CHERI RISC-V shows a 33 percent decrease in *TIMEOUT* errors for the padded variant, and 20 percent for the unpadded variant, in comparison to RISC-V.

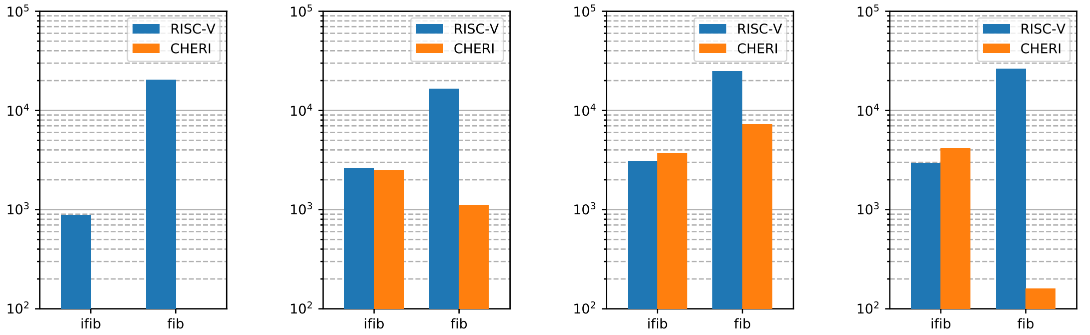
In summary the memory padding has an overall positive effect on the frequency of *TIMEOUT* errors and late timing failures. Consequently, CHERI RISC-V will exhibit less late timing failures for large binaries, or workloads which have a lot of static data.

4.4.2.3 Effects of parity

As discussed in Section 4.4.1.3, due to the capability encoding used by CHERI RISC-V certain capability corruptions cannot be detected. However, by adding a parity bit, which is verified any usage of the capability, they can be detected.

Figure 4.5a shows the number of faults that result in *TIMEOUT* errors for both benchmark variants evaluated on 32-bit variants of RISC-V and CHERI-P. Both variants show a complete absence of *TIMEOUT* errors when evaluated for the CHERI-P architecture. Consequently, all *TIMEOUT* errors that were recorded in the non-parity protected CHERI RISC-V architecture must be due faults in capabilities, which are undetectable due to the encoding.

Figure 4.5b, on the other hand, shows the number of faults that result in *TIMEOUT* errors for both variants evaluated on 64-bit variants of RISC-V and CHERI-P. Here, the recursive variant sees a smaller decrease of 93 percent in comparison to RISC-V. The iterative variant, sees a slight decrease of 4 percent, which is similar to the decrease observed for the non-parity protected CHERI RISC-V. The observation, that an increased instruction length worsen the benefits of CHERI RISC-V is in contrast to the result obtained in Section 4.4.2.2, where an increased instruction length leads to a decrease of the observed *TIMEOUT* errors.



(a) Number of *TIMEOUT* errors for each variant, built with optimizations, evaluated for *32-bit* architectures.

(b) Number of *TIMEOUT* errors for each variant, built with optimizations, evaluated for *64-bit* architectures.

(c) Number of *TIMEOUT* errors for each variant, built without optimizations, evaluated for *32-bit* architectures.

(d) Number of *TIMEOUT* errors for each variant, built without optimizations, evaluated for *64-bit* architectures.

Figure 4.5 – Number of faults that result in a *TIMEOUT* error for each fib implementation variant evaluated for RISC-V and CHERI-P. *ifib*, refers to the iterative implemention and *fib* to the recursive implementation. Both 32-bit and 64-bit architectures of each architecture are shown.

However, this observation can be attributed to optimization artifacts. Figure 4.5c, and Figure 4.5d show the number of *TIMEOUT* errors recorded for both variants of the fibonacci benchmark, compiled *without* optimizations, for 32-bit and 64-bit variants of RISC-V and CHERI-P. Without optimizations, *TIMEOUT* errors are decreased for an increased instruction length. While the recursive benchmarks shows a 71 percent decrease in *TIMEOUT* errors for 32-bit architectures, the same benchmark shows a 99 percent decrease for 64-bit architectures.

In summary, CHERI-P shows an additional improvement over CHERI RISC-V with regard to its frequency of late timing failures. However, its effect is highly dependent on the optimization level and instruction length for which the workload is evaluated.

Nonetheless, a 32-bit instruction length CHERI-P can eliminate *TIMEOUT* errors completely, if the workload is built with optimizations.

4.4.2.4 Summary

To summarize, CHERI RISC-V exhibits less late timing failures for workloads, which make heavy use of recursion, i.e., workloads that save a lot of control-flow related information in memory. For these, CHERI RISC-V reduced the frequency of late timing failures by 31 percent , even with a significantly enlarged fault space for unoptimized builds. Similar to unsignaled content failures, the fault space inflation skews the obtained results. Enabling compiler optimizations, however, does not further reduce the frequency of late timinig failures in CHERI RISC-V. Instead, CHERI RISC-V shows a smaller decrease in *TIMEOUT* errors of 20 percent in comparison to RISC-V. Considering the effects of instruction length, CHERI RISC-V fares better in its 64-bit architectural variant. Compared to the 32-bit variant, it shows a 59 percent reduction in late timing failures. Enabling memory padding further improves this to a 67 percent reduction. Therefore, the frequency of *TIMEOUT* errors is closely related to the memory layout of the workload and the instruction length of the architecture. Finally, parity-protected capabilities significantly reduce the frequency of late timing failures. The 32-bit variant of CHERI-P even eleminates *TIMEOUT* errors, if the benchmark is built with optimizations. The effect is more nuanced for the 64-bit variant of CHERI-P, however, *TIMEOUT* errors are still significantly decreased by 93 percent. In conclusion, CHERI RISC-V effectively reduces the frequency of late timing failures, which its workloads experience, and is, therefore, more resilient to soft-errors than RISC-V.

4.4.3 Hypothesis: Improved Detection of Existing Failure Modes

Another aspect of a system that is resilient to soft-errors, is its ability to detect erroneous conditions during its execution. A system that allows early and consistent detection of faults, is more resilient than a system that only detects a fault when it caused an unrecoverable error. Consequently, consistent and early fault detection is a cornerstone of a fault-tolerant and, therefore, resilient systems. To evaluate the fault detection capabilities of a system, the number of faults that cause a *TRAP* error are recorded. *TRAP* errors indicate that the system *detected* that an erroneous condition occurred and stopped execution. Therefore, a system that exhibits more *TRAP* errors better or, in other words, more consistently detects faults. Section 4.4.3.1 presents the recorded frequency of *TRAP* errors for each benchmark and architectural variant.

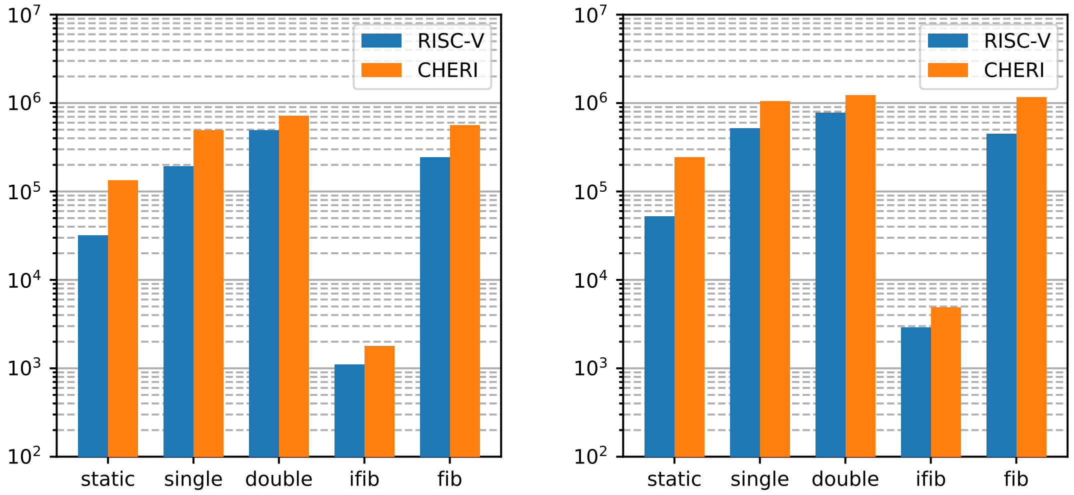
Additionally, the fault detection delay, that is the time between the faults inception and the resulting *TRAP* error, is recorded. Systems which show a lower fault detection delay are more resilient to soft-errors, since they can correct observed faults earlier. This is especially important for real-time systems, which often have fixed execution deadlines for their workloads. These deadlines must be met even if they encounter a fault during their execution. Section 4.4.3.1 discusses the observed detection latency for each benchmark and architectural variant. Finally, Section 4.4.3.3 summarizes the results and draws a conclusion about the enhanced detection capabilities of CHERI RISC-V.

4.4.3.1 *TRAP* errors

Figure 4.6 shows the number of faults, which result in a *TRAP* error for each benchmark, evaluated on 32-bit and 64-bit variants of RISC-V and CHERI RISC-V. All benchmarks are compiled with optimizations and without padding. In constrast to Section 4.4.1.1 and Section 4.4.2.1 unoptimized variants of each benchmark are not evaluated due to the previously shown fault space inflation.

For 32-bit architectures, CHERI RISC-V increases the number of *TRAP* errors by 143 percent on average in comparison to RISC-V. The static bubblesort variant shows the largest increase of 322.35 percent. Additionally, CHERI RISC-V shows a smaller benefit, i.e., increase of *TRAP* errors, for workloads which use more indirection, or store more control-flow information on the stack. This can be attributed this to the way CHERI RISC-V accesses stack variables. While RISC-V addresses variables saved on the stack directly through the current frame pointer, CHERI RISC-V introduces an additional level of indirection. To guarantee its protection model for stack objects, it must always indirectly address them through capabilities, which are also stored on the stack, and are loaded through the current frame pointer. Therefore, CHERI RISC-V uses more indirection by default, even if the workload is mostly stack-bound and does not use much indirection itself. This additional indirection leads to the observed large increase for workloads, which operate with little to no indirection, such as the static bubblesort variant (322 percent increase) and the iterative fibonnaci implementation (63 percent increase).

For 64-bit architectures a similar pattern is observed. CHERI RISC-V increases the number of *TRAP* errors by 151 percent averaged over all benchmarks. Again, stack-bound workloads show the largest increase in *TRAP* errors for CHERI RISC-V. In comparison to the 32-bit architectures, faults more often result in a *TRAP* error for CHERI RISC-V in 64-bit architectures. However, the difference



(a) Number of *TRAP* errors for each benchmark, evaluated on *32-bit* variants of RISC-V and CHERI RISC-V.

(b) Number of *TRAP* errors for each benchmark, evaluated on *64-bit* variants of RISC-V and CHERI RISC-V.

Figure 4.6 – Number of faults that result in a *TRAP* error for each benchmark evaluated for RISC-V and CHERI RISC-V. All benchmarks are compiled with optimizations and without padding.

is small (+8 percent), and can be attributed to the changed memory layout of the workload. To summarize, the instruction length of CHERI RISC-V only has a small effect on its fault detection capabilities.

Next, the effect of padding on the frequency of *TRAP* errors is evaluated. Similar to the changed instruction length, memory padding affects the memory layout of the benchmark. Figure 4.7 shows the number of faults that result in a *TRAP* error for each benchmark and both architectures. All benchmarks are compiled with optimizations and padding and are evaluated for 64-bit variants of each architecture. For the padded benchmarks CHERI RISC-V shows a 155 percent average increase in *TRAP* errors in comparison to RISC-V. Consequently, CHERI RISC-V shows only 4 percent more *TRAP* errors in comparison to the unpadded variant. Most of this increase can be attributed to the static and single variant of the bubblesort benchmark, where *TRAP* errors are increased by 376 percent and 113 percent respectively. The double bubblesort variant and the recursive fibonnacci implementation, conversely, show no increase in the number of *TRAP* errors recorded.

In conclusion, there is no clear pattern in the type of benchmark for which padding increases the number of detected faults. Nonetheless, it never worsen the fault detection for any of the evaluated benchmarks. Therefore, it can be assumed that the number of faults detected by CHERI RISC-V is not negatively affected by the memory layout or size of the workload's binary.

Finally, Figure 4.8 shows the number of faults that result in *TRAP* errors for each benchmark evaluated on 64-bit variants of RISC-V and the parity-protected *CHERI-P*. All benchmarks are built with enabled optimizations and disabled memory padding. *TRAP* errors are increased by 626 percent on average in comparison to RISC-V. This constitutes a 475 percent increase over non-parity CHERI RISC-V. However, a large increase is expected since any fault, which occurs in a capability, is now detected as a *TRAP* error. For the uniformly-distributed single-bit fault model that

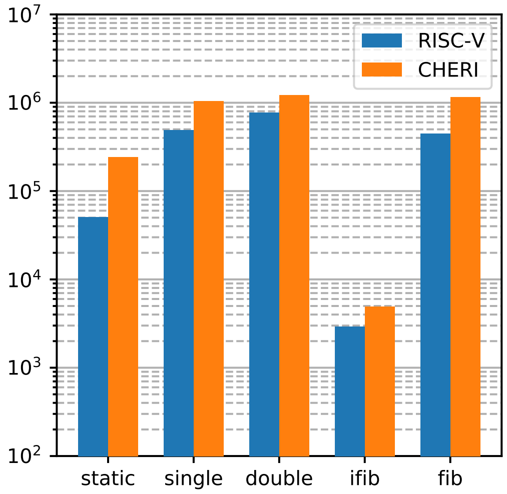


Figure 4.7 – Number of faults that result in a *TRAP* error for each benchmark evaluated for RISC-V and CHERI RISC-V. All benchmark are compiled with optimizations and *with* padding, and are evaluated on 64-bit architectures.

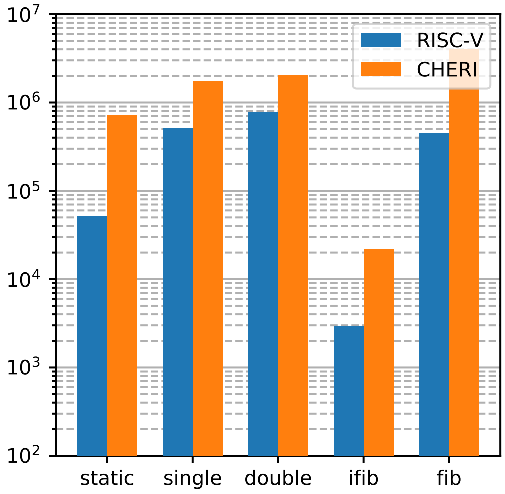


Figure 4.8 – Number of faults that result in a *TRAP* error for each benchmark evaluated for RISC-V and CHERI-P. All benchmarks are compiled with optimizations and without padding, and are evaluated on 64-bit architectures.

is assumed for thesis, this encompasses all possible capability faults. Assuming that most memory and register data in CHERI RISC-V is a capability, the large increase can be explained.

To summarize, CHERI-P increases the number of detected faults significantly in comparison to the non-parity protected CHERI RISC-V. Therefore, most of its faults must happen in its memory or register capabilities.

4.4.3.2 Detection Latency

A system that shows improved detection of existing failure modes is more resilient to soft-errors. In addition to an increased frequency of *TRAP* errors, a system can also show improved detection capabilities through a lower fault detection latency. The fault detection latency is the amount of time that passes after the inception of a fault, i.e., its injection point in FI experiments, and its detection by the system. In this thesis a is considered to be fault detected by the system if it shows a signaled halt failure, or in other words, stops with a *TRAP* error.

Figure 4.9 shows the fault detection latency averaged over all benchmarks, which are grouped by their variants (see Section 4.2.3). Going from left to right, *noopt* refers to all benchmarks compiled without optimizations, and without padding, while *opt* refers to all benchmarks compiled with optimizations, but without padding. Next, *padding* accumulates all benchmark variants, which are built with optimizations and padding. Finally, *parity* averages the detection latency of all benchmarks variants, built with optimizations and without padding. However, in contrast to *noopt*, *opt* and *padding*, which compare RISC-V and CHERI RISC-V, *parity* compares RISC-V and the parity-protected CHERI-P. Both 32-bit (Figure 4.9a) and 64-bit (Figure 4.9b) variants of the respective architectures are evaluated.

Due to the usage of *def/use* pruning, special consideration must be taken to weigh the recorded fault latency correctly. For each recorded crash time, its average detection delay is calcualted with

$$t_{delay,avg} = t_{crash} + 1 - (t_{begin} + t_{end})/2 \tag{4.1}$$

where t_{begin} and t_{end} refer to begin and end time of the equivalence interval respectively. This average delay is then again averaged over all benchmark variants for in respective class, and plotted in Figure 4.9.

For 32-bit architectures CHERI RISC-V consistently shows an average fault detection latency that is lower than in RISC-V. Relatively speaking, this decrease in detection is the lowest for non-optimized benchmark variants, where CHERI RISC-V shows a 35 cycle average delay and RISC-V shows a 135 cycle average, constituing a 74 percent decrease. For optimized variants this relative difference is even larger at −95 percent. Compiling each benchmark with optimizations and padding, further enlarges the average detection latency of RISC-V (now 2,101 cycles), whereas CHERI RISC-V shows no difference in its detection latency. Given that the additional memory padding only affects errors, that occur due to faulty pointers which CHERI RISC-V protects by default, this result is reasonable. Again, this increases the relative difference in detection latency to −97 percent. Finally, adding a parity-protection to each capability decreases the average detection latency for CHERI RISC-V to an average of 1.2 cycles. This corresponds to 100 percent decrease in comparison to RISC-V. The median detection latency for CHERI-P is 1 cycles. Averaging the obtained result over all non-parity variant groups yields an average detection of 499 cycles for RISC-V and 37 cycles for CHERI RISC-V, which corresponds to a relative difference of −93 percent.

For the 64-bit variants of each archictecture, CHERI RISC-V also shows a significantly decreased average fault detection latency. In comparison to the 32-bit variants, however, the relative difference is much more consistent across the different benchmark variant groups. Averaged over all groups, which are evaluated for the non-parity protected CHERI RISC-V, it decreases the fault detection latency by 91 percent. With parity-protected capabilities, the average detection latency further decreased to 7 cycles, which constitutes a 96 percent decrease in comparison to RISC-V.

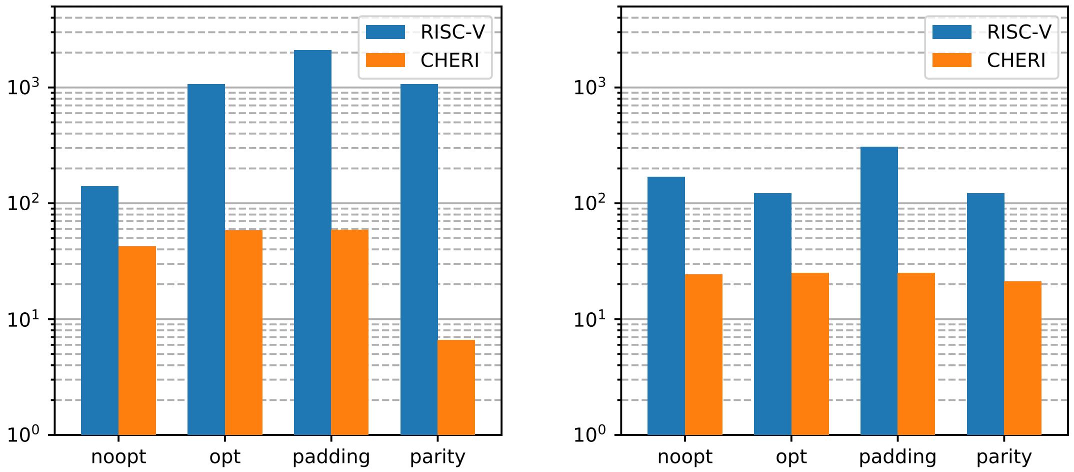
In summary, CHERI RISC-V significantly improves the average fault detection latency for all evaluated benchmarks. Regardless of the instruction length, CHERI RISC-V's fault detection latency is at least an order of magnitude lower than that of RISC-V.

4.4.3.3 Summary

At the beginning of this section, it was proposed that an architecture that improves detection of existing failure modes is more resilient to soft-errors. To improve detection, it must either exhibit more signaled halt failures, which can be corrected in fault-tolerant systems, or reduce the fault detection latency. CHERI RISC-V significantly increased signaled halt failures regardless of the evaluated instruction length, compilations options, and parity protection. Additionally, it improved detection latency by at least an order of magnitude, regardless of its evaluated instruction length, or benchmark variant. In conclusion CHERI RISC-V is more resilient to soft-errors than RISC-V when considering the improved detection of existing failure modes.

4.4.4 Summary

This section evaluated the soft-error resilience of both RISC-V and CHERI RISC-V. First the fault model was presented with which one data-heavy sorting benchmark, and one control-flow heavy recursion benchmark was evaluated. For both benchmark different implementation variants were discussed, which are especially designed to exhibit features, for which I predicted an improved soft-error resilience in Section 3.1. Addtionally, four compile-time variants of each benchmark were shown that are also evaluated to minimize the effect of the different code generation for both architectures. Each of the hypothesis made in Section 3.1 to evaluate, if and by how much the CHERI-protection model improved the soft-error resilience.



(a) Average latency for 32-bit architectures.

(b) Average latency for 64-bit architectures.

Figure 4.9 – Detection latency in cycles averaged over all benchmarks grouped by their variants. Both 32-bit and 64-bit architectures of CHERI RISC-V, CHERI-P and RISC-V are shown.

With regards to unsignaled content failures, CHERI RISC-V reduces their frequency in comparison to the unprotected RISC-V. However, special consideration must be given to the compilation options of each benchmark. If it is compiled without enabled optimizations, CHERI RISC-V's fault space inflation will often mask its reduction in the frequency of unsignaled content failures. At most, CHERI RISC-V showed a 16 percent decrease in unsignaled content failures. Additionally, the hypothesis that CHERI RISC-V fares better, that is more effectively prohibits these failures when the workload uses a lot of indirection, could be validated. Finally, an extended version of the CHERI RISC-V architecture was discussed that protects its capabilities' integrity with an additional parity bit. This CHERI-P architecture could reduce the frequency of unsignaled content failures by 45 in comparison to the unprotected RISC-V architecture.

A similar behavior was observed regarding to late timing failures. In comparison to the unsignaled content failures CHERI RISC-V significantly reduced their frequency. CHERI RISC-V reduced late timing failures the most for 64-bit architectures and workloads with padded memory. For the recursive variant of the fibonacci benchmark it reduced the number of late timing failures by a factor of 67 in comparison to RISC-V architecture. Evaluating the CHERI-P architecture showed a further decrease of late timing failures, and even eliminated them for certain benchmark and architecture configurations.

Finally, the improved error detection mechanism of CHERI RISC-V were discussed. It was observed that CHERI RISC-V significantly increased the number of signaled halt failures in comparison to RISC-V, often by more than a magnitude. CHERI RISC-V especially detects error more frequently if its capabilities are parity-protected. Additionally, the average detection latency of faults was derived for RISC-V, CHERI RISC-V and CHERI-P in different variants. In summary, CHERI RISC-V significantly decreases the fault detection delay, regardless of the evaluated benchmark or architecture variant.

Overall, all hypotheses presented in Section 3.1 could be validated. The CHERI protection model improves a systems error resilience to soft-errors.

CONCLUSION

5

This thesis evaluated the effect of capability-based memory protection on the soft-error resilience of a system. To evaluate the influence of memory protection two variants of the *same* architecture are evaluated, of which one has a memory protection system and the other has no protection. Specifically, RISC-V and CHERI RISC-V, which is an implementation of the CHERI protection model for RISC-V, are compared. To gauge their resilience to soft-errors multiple micro-benchmarks are evaluated on both architecture variants using the *fault forecasting* technique *fault injection*. For this, the FAIL* fault injection meta-framework is extended to allow fault injection experiments for the CHERI RISC-V architecture.

First, both architectures are compared in the frequency with which they exhibit unsignaled content and late timing failures in response to the injected faults. I find that CHERI RISC-V significantly reduces the number of unsignaled content failures. This reduction increases if more indirection is used by the evaluated benchmark. Similarly, CHERI RISC-V exhibits late timing failures less frequently than the unprotected RISC-V. Nonetheless, I observe that the obtained results are highly dependent on the instruction length of the evaluated architecture variant and the compilation options of the evaluated benchmark. Especially benchmarks compiled without compiler optimizations inflate the fault space for CHERI RISC-V and skew the obtained results. Additional research is required to evaluate the representativeness of the obtained results.

Secondly, both architectures are evaluated for the number of faults, which they detect, and their average detection latency. I discover that CHERI RISC-V generally detects more faults through signaled halt failures. Additionally, it also detects such faults much faster than the unprotected RISC-V.

Last, I observe that adding a simple parity-bit to each capability further improves the obtained results. CHERI RISC-V now exhibits unsignaled content failures and late timing failures less frequently and more faults can be detected.

In summary, I conclude that CHERI RISC-V shows increased soft-error resilience in comparison to RISC-V. Consequently, capability-based memory protection can be used as a protection technique against soft-errors. Although it increases the system attack surface, its additional protection is significant and surmounts the additional susceptibility.

Future work could evaluate more complex benchmarks to validate the obtained results for larger systems. Additionally, capability-based operating system, such as CheriOS, that use the compartmentalization mechanism of CHERI could be evaluated. Additional encapsulation of the evaluated software could be beneficial to the system's soft-error resilience. Furthermore, only uniformly-distributed single-bit faults were evaluated. CHERI's split memory architecture could be evaluated under the assumption that different types of memory have different inherent susceptibilities to faults. In this context multi-bit faults could be evaluated too. Finally, other memory protection schemes, such as classic MPU-based protection could be compared to the additional protection provided by CHERI. CHERI also works side-by-side with existing MPU and, therefore, a combination of both protections is also possible and might provide further benefits.

LIST OF ACRONYMS

BPSG	Borophosphosilicateglass
SOI	Silicon-on-insulator
MBU	Multi-bit upset
ECC	Error-correcting code
SMT	simultaneous multithreading
CFC	control-flow checking
COTS	commercial off-the-shelf
MMU	memory-managment unit
MPU	memory-protection unit
LLVM	low-level virtual machine
CHERI	Capability Hardware Enhanced RISC Instructions
RISC	reduced instruction set computer
PCC	program counter capability
DDC	default data capability
ISA	instruction set architecture
TCB	Trusted Computing Base
PLT	procedure linkage table
TLS	thread-local storage
FI	fault injection
EEA	execution environment abstraction
HTIF	host-target interface
CSR	control-and-status register

LIST OF FIGURES

2.1 A particle strikes a transistor which gate is floating or below the threshold voltage (a), creating a wave of minority and majority charge carriers in its wake. Subsequently, a depletion funnel is formed which collects most majority charge carriers into the nearest depletion region (b) and subsequently the nearest gate, causing a large current spike. Finally, most generated charge carriers have recombined, but a small diffusion current remains (c). 4
2.2 In the "chain of threats" faults activate errors, which propagate to failures, that themselves cause new faults. Errors are propagated across interfaces and can cause failures in the external system behavior or connected components. 7
2.3 Visualization of CHERIs protection properties. 17
2.4 Compressed (128 bit) and uncompressed (256 bit) capability representations. 20

3.1 Structure of the FAIL* plumbing layer, or in other words, its FI abstraction. Taken from [Sch16]. 27
3.2 Sequence diagram of a typical control-flow between an experiment procedure and the simulator which has been extended with an EEA layer. Taken from [Sch16]. 28
3.3 Structure of FAIL* assessment layer. It consists of four steps – tracing, import, pruning and injection –, which store their (intermediary) results in the internal database. Finally, during post-injection analysis, this database can be queried to extract results for the FI campaign. Taken from [Sch16]. 29
3.4 The class diagram for the virtual fault space implementation. 34
3.5 Example for equivalence intervals in a masked byte-addressed fault space. Each interval is indicated by a green color. 36
3.6 Sail register declarations with the corresponding generated C code for differently typed registers. `lbits` is a multi precision number type defined by the Sail standard library. 40

4.1 Number of faults that result in a *SDC* error for each bubblesort implementation variant and architecture. All variants are built without memory padding and evaluated on 32-bit architecture variants of CHERI RISC-V and RISC-V. 47
4.2 Number of faults that result in a *SDC* error for each bubblesort implementation variant and architecture. All variants are built with optimizations and are evaluated on *64-bit* architecture variants of CHERI RISC-V and RISC-V. 50
4.3 Number of faults that result in a *SDC* error for each bubblesort implementation variant and the RISC-V and *parity-protected* CHERI-P architecture. All variants are built with optimizations and without padding and are evaluated on 32-bit architecture variants. 51

4.4 Number of faults that result in a *TIMEOUT* error for each fib implementation variant and various architecture variants. *ifib* refers to the iterative implemention and *fib* to the recursive implementation. 53
4.5 Number of faults that result in a *TIMEOUT* error for each fib implementation variant evaluated for RISC-V and CHERI-P. *ifib*, refers to the iterative implemention and *fib* to the recursive implementation. Both 32-bit and 64-bit architectures of each architecture are shown. 55
4.6 Number of faults that result in a *TRAP* error for each benchmark evaluated for RISC-V and CHERI RISC-V. All benchmarks are compiled with optimizations and without padding. 57
4.7 Number of faults that result in a *TRAP* error for each benchmark evaluated for RISC-V and CHERI RISC-V. All benchmark are compiled with optimizations and *with* padding, and are evaluated on 64-bit architectures. 58
4.8 Number of faults that result in a *TRAP* error for each benchmark evaluated for RISC-V and CHERI-P. All benchmarks are compiled with optimizations and without padding, and are evaluated on 64-bit architectures. 58
4.9 Detection latency in cycles averaged over all benchmarks grouped by their variants. Both 32-bit and 64-bit architectures of CHERI RISC-V, CHERI-P and RISC-V are shown. 60

LIST OF TABLES

2.1 The CHERI capability data type. “∼” refers to a variable sized field. 18
2.2 Selection of permission bits in the MIPS implementation of the CHERI protection model. Adopted from [Wat+19b]. 19
2.3 Existing CHERI implementations, adapted from [WSWMN19] 22

4.1 Number of unique fault points for RISC-V and CHERI RISC-V for optimzed and non-optimized variants of the bubblesort benchmark. 49
4.2 Number of unique fault points for RISC-V and CHERI RISC-V for optimzed and non-optimized variants of the fibonacci benchmark. 54

LIST OF LISTINGS

LIST OF ALGORITHMS

3.1 FINDMATCHING 37
3.2 PURGEMATCHING 38

REFERENCES

[Acc+86] Mike Accetta et al. “Mach: A New Kernel Foundation for UNIX Development.” In: 1986, pp. 93–112.

[AJJ04] Folkesson J. Aidemark J. and Karlsson J. “Experimental Dependability Evaluation of the Artk68-FT Real-time Kernel.” In: *Proceedings of the 10th IEEE International Conference on Embeddded and Real-Time Computing Systems and Applications (RTCSA ’04)*. Gothenburg, Sweden, Aug. 2004.

[Arma] *ARM Architecture Reference Manual for ARMv8*. URL: `https://developer.arm.com/documentation/ddi0487/latest/` (visited on 10/01/2020).

[Armb] Alasdair Armstrong. “Sail architectural description language.” In: (). URL: `https://www.cl.cam.ac.uk/~pes20/sail/` (visited on 01/10/2020).

[Aus99] Todd M. Austin. “DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design.” In: *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 32. USA: IEEE Computer Society, 1999, 196–207. ISBN: 076950437X.

[Avi+04] 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.

[Avi85] A. Avizienis. “The N-Version Approach to Fault-Tolerant Software.” In: *IEEE Transactions on Software Engineering* SE-11.12 (1985), pp. 1491–1501.

[Avr] *AVR Instruction Set Manual*. 2020. URL: `http://ww1.microchip.com/downloads/en/DeviceDoc/AVR-Instruction-Set-Manual-DS40002198A.pdf`.

[Bar77] William B. Barker. *Longitudinal parity generator for use with a memory*. U.S. Patent 4 035 766. July 1977.

[Bau05] R. C. Baumann. “Radiation-induced soft errors in advanced semiconductor technologies.” In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 305–316.

[Bau+09] Christoph Baumann et al. “Formal Verification of a Microkernel Used in Dependable Software Systems.” In: *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*. SAFECOMP ’09. Hamburg, Germany: Springer-Verlag, 2009, 187–200. ISBN: 9783642044670. DOI: `10.1007/978-3-642-04468-7_16`. URL: `https://doi.org/10.1007/978-3-642-04468-7_16`.

[Bau+95] R. Baumann et al. “Boron as a primary source of radiation in high density DRAMs.” In: *1995 Symposium on VLSI Technology. Digest of Technical Papers*. 1995, pp. 81–82.

[BB84] R. D. Bannon and M. M. Bhansali. *Digital data storage error detecting and correcting system and method*. U.S. Patent 0 042 966. Apr. 1984.

[BF93] R. W. Butler and G. B. Finelli. "The infeasibility of quantifying the reliability of life-critical real-time software." In: *IEEE Transactions on Software Engineering* 19.1 (1993), pp. 3–12.

[BSH75] D. Binder, E. C. Smith, and A. B. Holman. "Satellite Anomalies from Galactic Cosmic Rays." In: *IEEE Transactions on Nuclear Science* 22.6 (1975), pp. 2675–2680.

[Bud20] Marcel Budoj. "Schotbruch: Automatisierte Ableitung von Injektionsplattformen für transiente Hardwarefehler aus formalen Prozessormodellen." PhD thesis. May 8, 2020. URL: `https://www.sra.uni-hannover.de/Theses/2019/budoj_20_ma.pdf`.

[Chea] *Cheri-OS micro kernel, Project Page*. URL: `https://github.com/CTSRD-CHERI/cherios`.

[Cheb] *cheribsd Project Page*. URL: `https://github.com/CTSRD-CHERI/cheribsd`.

[Che89] C.-L. Chen. *Double error correction - triple error detection code for a memory*. U.S. Patent 0 107 038. Aug. 1989.

[CWA00] S. Chatterjee, C. Weaver, and T. Austin. "Efficient checker processor design." In: *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*. 2000, pp. 87–97.

[CZ85] James A. Cairns and James F. Ziegler. *Coated ceramic substrates for mounting integrated circuits*. U.S. Patent 4 528 212A. July 1985.

[Dav+07] Francis M. David et al. "Improving Dependability by Revisiting Operating System Design." In: *Proceedings of the 3rd Workshop on on Hot Topics in System Dependability*. HotDep'07. Edinburgh, UK: USENIX Association, 2007, 1–es.

[Dav+19a] Francis M. David et al. "Curios: Improving reliability through operating system structure." English (US). In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*. Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008. 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008 ; Conference date: 08-12-2008 Through 10-12-2008. USENIX Association, Jan. 2019, pp. 59–72.

[Dav+19b] Brooks Davis et al. "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, 379–393. ISBN: 9781450362405. DOI: `10.1145/3297858.3304042`.

[Del97] Timothy J. Dell. "A white paper on the benefits of chipkill-correct ecc for pc server main memory." In: 1997.

[Dev+08] Joe Devietti et al. "Hardbound: Architectural Support for Spatial Safety of the C Programming Language." In: *SIGPLAN Not.* 43.3 (Mar. 2008), 103–114. ISSN: 0362-1340. DOI: `10.1145/1353536.1346295`. URL: `https://doi.org/10.1145/1353536.1346295`.

[DH12] Bjoern Doebel and Hermann Haertig. “Who Watches the Watchmen? Protecting Operating System Reliability Mechanisms.” In: *Eighth Workshop on Hot Topics in System Dependability (HotDep 12)*. Hollywood, CA: USENIX Association, Oct. 2012. URL: `https://www.usenix.org/conference/hotdep12/workshop-program/presentation/D{\"o}bel`.

[DW11] A. Dixit and A. Wood. “The impact of new technology on soft error rates.” In: *2011 International Reliability Physics Symposium*. 2011, 5B.4.1–5B.4.7.

[Fil+20] N. Wesley Filardo et al. “Cornucopia: Temporal Safety for CHERI Heaps.” In: *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 608–625. DOI: `10.1109/SP40000.2020.00098`. URL: `https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00098`.

[Gar66] H. L. Garner. “Error Codes for Arithmetic Operations.” In: *IEEE Transactions on Electronic Computers* EC-15.5 (1966), pp. 763–770.

[Gol+96] Ian Goldberg et al. “A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker.” In: *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*. SSYM’96. San Jose, California: USENIX Association, 1996, p. 1.

[GSZ09] B. Gill, N. Seifert, and V. Zia. “Comparison of alpha-particle and neutron-induced combinational and sequential logic error rates at the 32nm technology node.” In: *2009 IEEE International Reliability Physics Symposium*. 2009, pp. 199–205.

[HAR15] G. Hubert, L. Artola, and D. Regis. “Impact of scaling on the soft error sensitivity of bulk, FDSOI and FinFET technologies due to atmospheric radiation.” In: *Integration* 50 (2015), pp. 39 –47. ISSN: 0167-9260. DOI: `https://doi.org/10.1016/j.vlsi.2015.01.003`. URL: `http://www.sciencedirect.com/science/article/pii/S0167926015000048`.

[Hen+13] Jörg Henkel et al. “Reliable On-Chip Systems in the Nano-Era: Lessons Learnt and Future Trends.” In: *Proceedings of the 50th Annual Design Automation Conference*. DAC ’13. Austin, Texas: Association for Computing Machinery, 2013. ISBN: 9781450320719. DOI: `10.1145/2463209.2488857`. URL: `https://doi.org/10.1145/2463209.2488857`.

[Hof+14] Martin Hoffmann et al. “Effectiveness of Fault Detection Mechanisms in Static and Dynamic Operating System Designs.” In: *Proceedings of the 17th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC ’14)*. Ed. by IEEE Computer Society. Reno, NV, USA, 2014, pp. 230–237. DOI: `10.1109/ISORC.2014.26`. URL: `http://www4.cs.fau.de/Publications/2014/hoffmann_14_isorc.pdf`.

[Hof+15a] Martin Hoffmann et al. “dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel.” In: *Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium (RTAS ’15)*. Ed. by Richard West. Seatlle, WA, USA, 2015, pp. 259–270. DOI: `10.1109/RTAS.2015.7108449`. URL: `http://danceos.org/publications/RTAS-2015-Hoffmann.pdf`.

[Hof+15b] Martin Hoffmann et al. “dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel.” In: *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS ’15)*. Best Paper. Piscataway, NJ, USA: IEEE Press, Apr. 2015, pp. 259–270. DOI: `10.1109/RTAS.2015.7108449`.

[Hof16] Martin Hoffmann. “Konstruktive Zuverlässigkeit: Eine Methodik für zuverlässige Systemsoftware auf unzuverlässiger Hardware.” doctoralthesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2016.

[Inta] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation. 2019.

[Intb] *Intel Architecture Software Developer’s Manual*. Intel Corporation. Santa Clara, California, USA, 1999.

[JA88] J. . Jou and J. A. Abraham. “Fault-tolerant FFT networks.” In: *IEEE Transactions on Computers* 37.5 (1988), pp. 548–561.

[Jim+02] Trevor Jim et al. “Cyclone: A Safe Dialect of C.” In: *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*. ATEC ’02. USA: USENIX Association, 2002, 275–288. ISBN: 1880446006.

[JSK02] Robert L. Jardine James S. Klecka William F. Bruckert. *Error self-checking and recovery using lock-step processor pair architecture*. U.S. Patent 6 393 582 B1. May 2002.

[Kem80] Gary H. Kemmetmueller. *RAM error correction using two dimensional parity checking*. U.S. Patent 4 183 463. Jan. 1980.

[Kim77] D. Robert Kim. *Longitudinal parity generator for use with a memory*. U.S. Patent 4 016 409. Apr. 1977.

[Lay+98] P.J. Layton et al. “Single-Event Latchup Protection of Integrated Circuits.” In: (1998). URL: `https://www.techbriefs.com/component/content/article/tb/supplements/etb/briefs/1836` (visited on 10/28/2020).

[Li+16] Tuo Li et al. “Processor Design for Soft Errors: Challenges and State of the Art.” In: *ACM Comput. Surv.* 49.3 (Nov. 2016). ISSN: 0360-0300. DOI: `10.1145/2996357`. URL: `https://doi.org/10.1145/2996357`.

[LV62] R. E. Lyons and W. Vanderkulk. “The Use of Triple-Modular Redundancy to Improve Computer Reliability.” In: *IBM Journal of Research and Development* 6.2 (1962), pp. 200–209.

[Lyo00] Daniel Lyons. “Sun Screen.” In: (2000). URL: `https://web.archive.org/web/20100528111258/https://www.forbes.com/forbes/2000/1113/6613068a.html` (visited on 07/21/2020).

[Mah+10] N. N. Mahatme et al. “Analysis of soft error rates in combinational and sequential logic and implications of hardening for advanced technologies.” In: *2010 IEEE International Reliability Physics Symposium*. 2010, pp. 1031–1035.

[Mah+14] N. N. Mahatme et al. “Impact of technology scaling on the combinational logic soft error rate.” In: *2014 IEEE International Reliability Physics Symposium*. 2014, 5F.2.1–5F.2.6.

[MBS08] A. Meixner, M. E. Bauer, and D. J. Sorin. “Argus: Low-Cost, Comprehensive Error Detection in Simple Cores.” In: *IEEE Micro* 28.1 (2008), pp. 52–59.

[Mic+05] S. E. Michalak et al. “Predicting the number of fatal soft errors in Los Alamos national laboratory’s ASC Q supercomputer.” In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 329–335.

[Mip] *MIPS32 Architecture for Programmers Volume IV-a: The MIPS16 Application Specific Extension to the MIPS32 Architecture*. 2001.

[MJ81] Joseph T. Marino Jr. *DES Parity check system*. U.S. Patent 4 262 358. Apr. 1981.

[MTI97] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. “Fault injection techniques and tools.” In: *Computer* 30.4 (1997), pp. 75–82.

[Muk08] Shubu Mukherjee. *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 9780080558325.

[MW79] T. C. May and M. H. Woods. “Alpha-particle-induced soft errors in dynamic memories.” In: *IEEE Transactions on Electron Devices* 26.1 (1979), pp. 2–9.

[Nag+09] Santosh Nagarakatte et al. “SoftBound: Highly Compatible and Complete Spatial Memory Safety for c.” In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’09. Dublin, Ireland: Association for Computing Machinery, 2009, 245–258. ISBN: 9781605583921. DOI: `10.1145/1542476.1542504`. URL: `https://doi.org/10.1145/1542476.1542504`.

[Nar+18] B. Narasimham et al. “Scaling trends and bias dependence of the soft error rate of 16 nm and 7 nm FinFET SRAMs.” In: *2018 IEEE International Reliability Physics Symposium (IRPS)*. 2018, pp. 4C.1–1–4C.1–4.

[Nic10] Michael Nicolaidis. *Soft Errors in Modern Electronic Systems*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 1441969926.

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. “CCured: Type-Safe Retrofitting of Legacy Code.” In: *SIGPLAN Not.* 37.1 (Jan. 2002), 128–139. ISSN: 0362-1340. DOI: `10.1145/565816.503286`. URL: `https://doi.org/10.1145/565816.503286`.

[OG65] G. A. Oliver and E. L. Glaser. “System Design of a Computer for Time-Sharing Applications.” In: *Managing Requirements Knowledge, International Workshop on*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1965, p. 197. DOI: `10.1109/AFIPS.1965.95`. URL: `https://doi.ieeecomputersociety.org/10.1109/AFIPS.1965.95`.

[OSM02a] 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.

[OSM02b] 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.

[PF82] Patel and Fung. “Concurrent Error Detection in ALU’s by Recomputing with Shifted Operands.” In: *IEEE Transactions on Computers* C-31.7 (1982), pp. 589–595.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. “A case for Redundant Arrays of Inexpensive Disks (RAID).” In: *ACM SIGMOD Record* 17 (July 1988). DOI: `10.1145/50202.50214`.

[Pic82] J. C. Pickel. “Effect of CMOS Miniaturization on Cosmic-Ray-Induced Error Rate.” In: *IEEE Transactions on Nuclear Science* 29.6 (1982), pp. 2049–2054.

[RB90] A. L. N. Reddy and P. Banerjee. “Algorithm-based fault detection for signal processing applications.” In: *IEEE Transactions on Computers* 39.10 (1990), pp. 1304–1308.

[RB96] A. Roy-Chowdhury and P. Banerjee. “Algorithm-based fault location and recovery for matrix computations on multiprocessor systems.” In: *IEEE Transactions on Computers* 45.11 (1996), pp. 1239–1247.

[Rei+05] G. A. Reis et al. "SWIFT: software implemented fault tolerance." In: *International Symposium on Code Generation and Optimization*. 2005, pp. 243–254.

[RJS19] Abhishek Rhisheekesan, Reiley Jeyapaul, and Aviral Shrivastava. "Control Flow Checking or Not? (For Soft Errors)." In: *ACM Trans. Embed. Comput. Syst.* 18.1 (Feb. 2019). ISSN: 1539-9087. DOI: `10.1145/3301311`. URL: `https://doi.org/10.1145/3301311`.

[RM06] Amir Rajabzadeh and Seyed Ghassem Miremadi. "CFCET: A hardware-based control flow checking technique in COTS processors using execution tracing." In: *Microelectronics Reliability* 46.5 (2006), pp. 959 –972. ISSN: 0026-2714. DOI: `https://doi.org/10.1016/j.microrel.2005.07.108`. URL: `http://www.sciencedirect.com/science/article/pii/S0026271405002921`.

[Rot99] Eric Rotenberg. "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors." In: 1999, pp. 84–91.

[Rus] *Rust: A language empowering everyone to build reliable and efficient software*. URL: `https://www.rust-lang.org` (visited on 08/09/2020).

[San+08] P. N. Sanda et al. "Soft-error resilience of the IBM POWER6 processor." In: *IBM Journal of Research and Development* 52.3 (2008), pp. 275–284.

[SB13] Matthew S. Simpson and Rajeev K. Barua. "MemSafe: ensuring the spatial and temporal memory safety of C at runtime." In: *Software: Practice and Experience* 43.1 (2013), pp. 93–128. DOI: `10.1002/spe.2105`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2105`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2105`.

[Sch10] Ute Schiffel. "Hardware Error Detection Using AN-Codes." PhD thesis. TU Dresden, Dec. 2010. URL: `https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa-69872`.

[Sch+15] H. Schirmeier et al. "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance." In: *2015 11th European Dependable Computing Conference (EDCC)*. 2015, pp. 245–255.

[Sch16] Horst Schirmeier. "Efficient Fault-Injection-based Assessment of Software-Implemented Hardware Fault Tolerance." Dissertation. Technische Universität Dortmund, July 2016. DOI: `10.17877/DE290R-17222`.

[Sla11] C. Slayman. "Soft error trends and mitigation techniques in memory devices." In: *2011 Proceedings - Annual Reliability and Maintainability Symposium*. 2011, pp. 1–5.

[Sle+99] T. J. Slegel et al. "IBM's S/390 G5 microprocessor design." In: *IEEE Micro* 19.2 (1999), pp. 12–23.

[Sti12] Michael Stilkerich. "Memory Protection at Option - Application-Tailored Memory Safety in Safety-Critical Embedded Systems." doctoralthesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2012.

[Wah+93] Robert Wahbe et al. "Efficient Software-Based Fault Isolation." In: *In Proceedings of the 14th ACM Symposium on Operating Systems Principles*. 1993, pp. 203–216.

[Wat+19a] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 20191213*. Tech. rep. EECS Department, University of California, Berkeley, 2019. URL: `https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf`.

[Wat+19b] Robert N. M. Watson et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)*. Tech. rep. 927. Cambridge, United Kingdom: University of Cambridge, Computer Laboratory, 2019. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf` (visited on 08/09/2020).

[Woo+14] J. Woodruff et al. "The CHERI capability model: Revisiting RISC in an age of risk." In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, pp. 457–468.

[Woo+19] J. Woodruff et al. "CHERI Concentrate: Practical Compressed Capabilities." In: *IEEE Transactions on Computers* 68.10 (2019), pp. 1455–1469.

[Woo99] Alan Wood. "Data integrity concepts, features, and technology." In: *White paper, Tandem Division, Compaq Computer Corporation* (1999).

[WSWMN19] Robert N. M. Watson, Peter Sewell Simon W. Moore, and Peter G. Neumann. *An Introduction to CHERI*. Tech. rep. 941. Cambridge, United Kingdom: University of Cambridge, Computer Laboratory, 2019. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf` (visited on 08/09/2020).

[Xia+19] Hongyan Xia et al. "CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety." In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, 545–557. ISBN: 9781450369381. DOI: `10.1145/3352460.3358288`. URL: `https://doi.org/10.1145/3352460.3358288`.

[Ziv+19] Darko Zivanovic et al. "DRAM Errors in the Field: A Statistical Approach." In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, 69–84. ISBN: 9781450372060. DOI: `10.1145/3357526.3357558`. URL: `https://doi.org/10.1145/3357526.3357558`.

[ZL79] J. F. Ziegler and W. A. Lanford. "Effect of Cosmic Rays on Computer Memories." In: *Science* 206.4420 (1979), pp. 776–788. ISSN: 0036-8075. DOI: `10.1126/science.206.4420.776`. eprint: `https://science.sciencemag.org/content/206/4420/776.full.pdf`. URL: `https://science.sciencemag.org/content/206/4420/776`.