Hardware Error Detection Using AN-Codes

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Ute Schiffel

geboren am 08. 07. 1980 in Sebnitz

Gutachter:	Prof. Christof Fetzer, PhD, Technische Universität Dresden
	Prof. Dr. Wolfgang Ehrenberger, Hochschule Fulda
Datum der Verteidigung:	20. Mai 2011

Dresden, den 10.06.2011

Abstract

Due to the continuously decreasing feature sizes and the increasing complexity of integrated circuits, commercial off-the-shelf (COTS) hardware is becoming less and less reliable. However, dedicated reliable hardware is expensive and usually slower than commodity hardware. Thus, economic pressure will most likely result in the usage of unreliable COTS hardware in safety-critical systems.

The usage of unreliable, COTS hardware in safety-critical systems results in the need for software-implemented solutions for handling execution errors caused by this unreliable hardware. In this thesis, we provide techniques for detecting hardware errors that disturb the execution of a program. The detection provided facilitates handling of these errors, for example, by retry or graceful degradation.

We realize the error detection by transforming unsafe programs that are not guaranteed to detect execution errors into safe programs that detect execution errors with a high probability. Therefore, we use arithmetic AN-, ANB-, ANBD-, and ANBDmem-codes. These codes detect errors that modify data during storage or transport *and* errors that disturb computations as well. Furthermore, the error detection provided is independent of the hardware used.

We present the following novel encoding approaches:

- *Software Encoded Processing (SEP)* that transforms an unsafe binary into a safe execution at runtime by applying an ANB-code, and
- *Compiler Encoded Processing (CEP)* that applies encoding at compile time and provides different levels of safety by using different arithmetic codes.

In contrast to existing encoding solutions, SEP and CEP allow to encode applications whose data and control flow is not completely predictable at compile time.

For encoding, SEP and CEP use our set of encoded operations also presented in this thesis. To the best of our knowledge, we are the first ones that present the encoding of a complete RISC instruction set including boolean and bitwise logical operations, casts, unaligned loads and stores, shifts and arithmetic operations.

Our evaluations show that encoding with SEP and CEP significantly reduces the amount of erroneous output caused by hardware errors. Furthermore, our evaluations show that, in contrast to replication-based approaches for detecting errors, arithmetic encoding facilitates the detection of permanent hardware errors. This increased reliability does not come for free. However, unexpectedly the runtime costs for the different arithmetic codes supported by CEP compared to redundancy increase only *linearly*, while the gained safety increases *exponentially*.

Für Arthur 

Es weht der Wind ein Blatt vom Baum,
von vielen Blättern eines.
Das eine Blatt, man merkt es kaum,
denn eines ist ja keines.
Doch dieses eine Blatt allein,
war Teil von unsrem Leben.
Drum wird uns dieses Blatt allein,
für immer, immer fehlen.

Hermann Hesse

Acknowledgments

Over the last years many people helped me to complete this thesis. Now, it is time to thank them for their support.

My advisor Christof Fetzer always believed in encoding – even when I did not. He was always open for discussions and an endless source of ideas. His constant request: *"You could publish at conference XYZ."* ensured a steady progress of my work. Thank you.

My colleagues at the chair for Systems Engineering at TU Dresden provided a friendly and enjoyable working environment. They were always open for discussing ideas, problems and gave lots of feedback on paper drafts and presentations. I especially thank, Martin Süßkraut whose ideas and suggestions considerably helped to improve the Encoding Compiler and this thesis, which he proof-read from the first to the very last page, André Schmitt who transformed my ideas for the ANB-encoding Compiler into a compiler pass during his diploma thesis, Thomas Knauth who implemented the list- and tree-based version management, Gert Pfeifer who had always time for me: either for just listening or for explaining some interesting P2P or DNS technique, Andrey Brito on whom I could count on to drop by on these long evenings before a paper deadline, Martin Nowack who proof-read the short version of this thesis, and Claudia Einer and Karina Wauer who helped me to survive in an otherwise women-free work environment.

Special thanks go to my husband Stephan. He endlessly discussed problems and possible solutions with me, he proof-read this thesis, and supported me wherever he could.

Und zu guter Letzt: Danke meine lieben Eltern, daß Ihr mich immer unterstützt und gefördert habt und auch heute noch für Stephan und mich da seid und alle unsere Vorhaben mit Ratschlägen und Hilfe begleitet.

Contents

1. Introduction

"Dependability is the ability to deliver service that can justifiably be trusted." [ALRL04] When building dependable systems, we have to consider what can go wrong in the system. Computing systems contain many sources for failures:

- Erroneous hardware might execute the software incorrectly.
- Software contains bugs and security vulnerabilities with high probability. These can lead to erroneous behavior when the software is executed.
- Users and operators of the system may make mistakes.

For building dependable systems, we have to deal with all these possible failures either by preventing them by removing their causes or by tolerating them. In a dependable computing system, we have to ensure that the software is executed correctly by the hardware, that the correct software is executed, that users and operators use the software correctly, and that the software itself is correct.

Focus of this thesis

This thesis focuses on ensuring that software is executed correctly. We present mechanisms that enable the user to detect if software was executed correctly or not. Being able to detect an incorrect execution, the user can react to the problem by, for example, using checkpointing and re-execution. However, tolerating the execution errors detected by our approaches is not part of this thesis. This thesis presents only mechanisms for detecting execution errors.

Nobody questions that software contains bugs. But is today's hardware unreliable? Many people assume that hardware executes software correctly. Most often, we think a software error is the culprit of a failing system. This assumption will change in the future. It is already wrong for highly dependable systems that require smaller failure rates. In the future, the share of failures that are caused by faulty hardware will increase. See Chapter 2 for a detailed discussion of hardware reliability.

While hardware reliability has improved dramatically over the last decades, the decreasing feature sizes[1] of integrated circuits and their growing design complexity lead to less reliable hardware in the future. According to Baumann [Bau05] the failure rate of hardware even with additional error mitigation mechanisms still ranges from 50–200 FIT[2]. Without the often expensive mitigation, the rate increases to over 50,000 FIT per chip.

A failure rate of 50,000 FIT means that a failure should be expected every 2 years for a chip running 24 hours a day, 7 days a week. For large scale server applications using hundreds of chips a failure rate of 50,000 FIT results in several

[1] Feature size means the size of the elements on a chip.
[2] FIT denotes *failure in time*. 1 FIT equals 1 failure in 10^9 device hours.

failing chips per week [Bau05]. Thus, this error rate bears an unacceptable economic risk for high-volume cost-critical systems. For nuclear plants even 200 FIT is much too high considering the tremendous consequences such a failure might have. 200 FIT means approximately one failure every 500 years for one continuously working chip. Currently Germany has 17 active nuclear plants. Even if each had only one safety-critical chip, that results in approximately one possibly fatal failure in one of these chips every 29 years. To summarize, the required failure rate for a single system depends on the number of such systems used and the criticality of these systems. For many systems, the currently provided hardware failure rates of commodity hardware are too high. Thus, we need additional means to ensure the correct execution of software by hardware.

Hardware-implemented reliability

Critical and, in particular, safety-critical systems in the past have been mainly built using custom hardware that provides better error detection and masking than common hardware. For example, custom reliable hardware is radiation hardened to prevent environment induced execution errors. Another approach is simple redundancy where two or more processors execute the same code and check each other.

Recent research developing safety-critical hardware tries to implement more sophisticated redundancy. These approaches seem very promising, especially with respect to runtime costs. However, to the best of our knowledge, none of the currently commercially available commodity systems use any of these new solutions that we shortly discuss in Section 10.2 of our *Related Work* Chapter.

Economics of reliability

In contrast to commodity hardware, the market for custom reliable hardware is comparatively small. Additionally, development costs for custom reliable hardware are high. Hence, custom reliable hardware is more expensive than commodity hardware. In contrast to that, the average selling price per computational device has decreased from 200$ to 5-10$ [Dec05]. In the future, economic pressure will necessitate the use of unreliable but cheap commodity hardware even in critical systems. This results in the need for mechanisms that facilitate to build reliable systems based on unreliable hardware. Thus, software-implemented mechanisms are required.

These mechanisms would also facilitate mixed-mode systems, which allow to consolidate hardware. *Mixed-mode systems* execute both safety-critical and non-critical applications on the same possibly unreliable commodity hardware. With the gained flexibility, hardware utilization can be improved ensuring optimal capacity utilization. Mixed-mode systems allow to use one powerful processor instead of several small ones, which are specifically adapted to the criticality of their tasks. The costs for this one commodity processor and its setup costs will be less than the costs of several custom processors and their installation. For these systems, the usage of custom reliable hardware would be especially uneconomic because running non-critical applications on expensive custom reliable hardware is a waste of resources.

For using commodity hardware in dependable systems, we have to handle their wide spectrum of possible failures. Commodity hardware exhibits not only fail-stop but also arbitrary value failures, which are more difficult to detect and

to mask. A system that exhibits arbitrary value failures might produce arbitrary erroneous output. This deviation from the expected behavior of a software is also known as *silent data corruption*. In contrast to crash failures (fail-stop), these failures are especially difficult to handle because they are not easy to detect.

Software-implemented reliability

Thus, techniques are required that facilitate building reliable systems using unreliable hardware. Therefore, it is required to extend the limited failure detection capabilities of commodity hardware with the help of software. This thesis presents several approaches that prevent silent data corruptions without the need for custom hardware. The presented approaches realize *failure virtualization* by turning harder to handle value failures into crash failures. Whenever an execution error is detected that might result in a silent data corruption, the execution is aborted. As Powell shows in [Pow95], under certain conditions more reliable systems can be build if the used protocols can assume a crash failure model instead of an arbitrary failure model. For the arbitrary failure model usually more redundancy is required. This increases the probability that any of the redundant parts fails. Additionally, the fail-stop model facilitates the usage of less complex algorithms to manage the redundancy that is used to tolerate hardware errors. This reduces the risk of bugs in these algorithms.

When implementing hardware error detection in software, one will need more CPU cycles to execute an application. In comparison to using custom reliable hardware, one can however use commodity hardware. Commodity hardware is typically faster and less expensive than custom reliable hardware because its development is faster progressing and less time and money consuming. The larger market of commodity hardware further decreases its cost. Furthermore, in many systems, only a few application components are critical and only these components need to be protected by additional error detection. These mixed-mode systems can be economically realized using commodity hardware and software-implemented hardware error detection. In summary, the flexibility gained by using a software-based approach allows

1. to reduce the hardware cost, and
2. to bound the performance impact of the software-based error detection by focusing on the critical application components.

Arithmetic codes

The error detection approaches presented in this thesis are based on *arithmetic codes*, which we introduce in Chapter 3. They facilitate software-implemented end-to-end hardware error detection, i. e., cover all components that might cause a data corruption. In particular, arithmetic codes cover

- errors that modify data stored in memory or
- data transported on a bus and
- errors that disturb computations implemented by logical circuits, e. g., address computations or computations done in an arithmetic logical unit.

The detection provided is end-to-end because after encoding data, errors disturbing its transportation, its storage, and its processing are detectable. For an ideal implementation of arithmetic codes no windows of vulnerability exist. This is much harder to achieve with hardware-implemented error detection mechanisms.

For example, if parities combined with redundant execution are used, data is vulnerable for unnoticeable modifications after its redundant computation and before its parity for storage is computed. Removing these windows of vulnerability is difficult.

An application that is protected from undetected execution errors by an arithmetic code is called *(arithmetically) encoded*. We will in this thesis present different ways to arithmetically encode programs written in C. The error detection capabilities of arithmetically encoded applications are largely decoupled from the error detection capabilities of the hardware used. They only depend on the chosen arithmetic code and its parameters. This eases hardware replacement in critical systems because less requirements are posed on the hardware used.

From the broad variety of known arithmetic codes (see Chapter 3), we chose the AN-codes because they are the most useful for software implementation. The other codes are either

- not suitable for software implementation because only a hardware implementation can provide their full detection capabilities, or
- they are less powerful because they do only support a restricted set of recognizable errors or of operations that conserve the code.

Our error detection implementations support different AN-codes with different error detection capabilities. Thereby different *safety levels* are provided to systems engineers. Our experiments show that AN-codes decrease the rate of silently corrupted output immensely compared to unprotected applications. Depending on the AN-code used, this reduction ranges from one to three orders of magnitude. Furthermore, the detection capabilities of AN-codes are also better than that of software-implemented redundancy – especially when permanent hardware errors are considered. Our experiments also show that a higher detection capability always comes at the cost of an increased runtime overhead. Increased safety as increased security does not come for free. This is true for different AN-codes compared to each other as well as for AN-codes compared to software-implemented redundancy. In Chapter 8 we will show the following: When choosing a detection mechanism with higher detection rate, the performance degradation is *linear*. However, the gain, i. e., reduction of the rate of undetected silent corruptions, grows *exponentially*. Thus, the different safety levels that we provide enable systems engineers to balance the error detection capabilities and the runtime overheads of the error detection.

Thesis structure

The thesis is structured in the following way: Chapter 2 discusses the evolution of reliability of hardware in the future and the impact that unreliable hardware has on computing systems. Furthermore, we introduce the failure model of hardware that we assumed in the development of the error detection mechanisms. Chapter 3 describes arithmetic codes in general and some specific codes. To encode an application it is required to use operations that are capable of handling encoded values. These – so-called *encoded operations* – preserve the code in an error-free execution. An erroneous execution with high probability results in invalid code words. Chapter 4 presents our encoded versions of the operations required to encode programs. Since the error detection capabilities of an encoded

program do not only depend on the arithmetic code used, but also on the code parameters, we discuss their selection in Chapter 5. Using the encoded operations described in Chapter 4 encoding an application can be done at different levels of abstraction and different points in the application's life cycle. Chapters 6, 7, and 8 present three different approaches to encode an application. While Chapter 6 introduces the Vital Coded Processor that was first described by Forin in [For89], Chapter 7 describes our newly developed *Software Encoded Processing (SEP)* and Chapter 8 presents our *Compiler Encoded Processing (CEP)*. For evaluating hardware error detection mechanisms error injection is a common tool. Since none of the available error injectors fulfilled our needs, we developed our own injection tools. Chapter 9 describes the reasons for doing so and the tools' design and implementation. The thesis is concluded with a review of related work in Chapter 10 and a conclusion in Chapter 11.

Contributions

Our contributions to the research community are:

The set of encoded instructions: For encoding whole applications, we need a code-conserving version for all instructions that we encounter in an application. This encoded version of an operation takes encoded input values and produces encoded output values if no error disturbs the execution. In Chapter 4 we describe how we encode instructions that will be encountered during encoding applications. To reduce the manual work of encoding, we only encoded a small number of arithmetic operations by hand. These *encoded base operations* are described in Section 4.2.1. For more complex operations such as type casting or shifting, we developed our own encodable C-implementations – the *replacement operations*. These are described in Section 4.2.2. All occurrences of such operations are replaced with their encodable version before the actual encoding is done. Thus, they are encoded automatically afterwards. The remaining sections of Chapter 4 describe the encoding of instructions implementing control flow and memory access. To the best of our knowledge, we are the first ones presenting a complete set of encoded instructions that facilitates complete encoding of applications implemented in ANSI C.

Support of dynamic memory for ANB- and ANBD-encoding Forin introduced in [For89] AN-encoding with signatures (ANB-codes) and timestamps (ANBD-codes). His implementation – the Vital Coded Processor – is introduced in Chapter 6. One of its main drawbacks is that it does not support dynamically accessed memory for which the access pattern is not known at compile time. In Section 4.6 we present our approach to support dynamically accessed memory using our newly developed *dynamic signatures*.

Software Encoded Processing (SEP) Chapter 7 describes Software Encoded Processing that protects binaries during execution from undetected silent data corruption through hardware errors. Therefore, we developed an encoded interpreter that executes the binaries using solely encoded operations. Error injections confirmed the error detection capabilities of this approach.

Compiler Encoded Processing (CEP) The encoded execution of binaries using an encoded interpreter is expensive in terms of runtime overhead. It can be expected that for safety-critical applications the source code is available since the source code of critical components is anyhow required to be able to fix bugs or modify their behavior. Thus, we developed the Encoding Compiler that we present in Chapter 8. In contrast to SEP, the Encoding Compiler supports different safety-levels by using different AN-codes. Furthermore, the runtime overhead induced by the resulting encoded applications is far less than for applications executed using SEP.

Comparison of duplicated execution and different AN-codes For comparing the different AN-codes to duplicated execution we implemented a simple software-based double modular redundancy and a signature-based control flow checking. We compared the error detection capabilities and runtime overhead of the different AN-codes, double modular redundancy, and double modular redundancy with additional control flow checking. Our experiments have shown that safety can be adapted to the required needs by choosing an appropriate safety-level – implemented for example by one of the AN-codes. However, the higher the obtained error detection capabilities are, the higher are also the runtime costs in terms of execution time. For details see Section 8.5.

Symptom-based error injector with debugging support For evaluating our error detection mechanisms, we used symptom-based error injection. Since we were not able to obtain a sufficient error injector that implemented the error model described in Section 2.5, we implemented our own injectors: FITgrind and EIS. They are described in Chapter 9. Their injection can be applied to arbitrary applications. Hence, we can compare the error detection capabilities of different error detection mechanisms and also of applications that are not protected by additional mechanisms at all. The applications analyzed can be exposed to different symptoms of hardware failures. Furthermore, the error injector EIS allows to debug error detection mechanisms by providing traces of the data flow that was influenced by an injected error. This helps to find weaknesses in our error detection mechanisms and was also not available in any other error injection tool.

2. Reliability of Hardware

After introducing some basic terminology, this chapter first motivates why hardware is expected to become less reliable in the future. Secondly, the chapter motivates why we should take care of unreliable hardware. Therefore, we represent studies and reports that demonstrate the impact of hardware errors on safety and security of systems and the economical impacts undetected errors had. Afterwards we introduce the symptom-based error model that we assume for all our further developments. This error model describes the symptoms unreliable hardware can cause at the level of software during the execution of an application. Hence, it is hardware independent and enables us to develop and test our hardware error detection independent of a specific hardware.

2.1. Terminology

For describing the reliability of systems – in this chapter mainly hardware systems – we use the terminology introduced in [ALRL04]. The most important terms are explained in the following.

Failure

A *failure* occurs when a system's behavior deviates from its expected correct behavior. For example, if for the system memory not the same word is read from an address as was written to that address before.

Error

An *error* is the deviation in a system's state from the expected correct version of the system's state. An error might lead to a failure if it gets *activated*. If an error does not result in a failure, it is said to be *masked*. In our memory example, a flipped bit is an error. If that bit is read, the error is activated. If that bit is overwritten with other valid data, the error is masked.

Fault

A *fault* is the cause for an error. A fault might be activated. In that case it leads to an error. In our memory example, the fault is, for example, the vulnerability of the memory to radiation induced bitflips.

Permanent faults are continuous in time. For hardware they are either caused by irreversible physical changes or by a faulty design. Permanent faults cause so-called *hard* or *permanent errors*. These errors are repeatable under the same conditions. For example, if a memory cell has a bit that is stuck at 1, every data item written into that cell will be modified accordingly.

The presence of *transient faults* is bounded in time. If an operation disturbed by a transient fault is repeated, the result will be most likely correct. An example for a transient fault is noise on the power supply.

An *intermittent fault* occurs, than vanishes, and than reappears again. Unstable or marginal hardware is prone to intermittent faults, for example, a transistor

that is going to break down – one time it is working correctly and the next time not.

Transient and intermittent faults cause *soft errors*. These are temporary malfunctions in a circuit [Lau06]. The *soft error rate (SER)* is the frequency of occurrence of soft errors.

2.2. Causes and Effects of Hardware Errors

According to Siewiorek et. al. [SCK04] hardware reliability has been increasing with every new generation. However, in the future the decreasing feature size of hardware will not lead to more reliable but to less reliable hardware [SG99]. Logical building blocks as well as memory will become more error-prone. While the decreasing size of the elements of processors and the growing design complexity of processors increase the performance achieved, they decrease reliability. Hence, the rate of hardware failures will increase.

In the following, we will first present the reasons for increasing unreliability of hardware and, second, discuss how unreliable hardware currently is and in the future will become.

2.2.1. Causes for Increasing Unreliability of Hardware

Effects of decreasing feature size

The decreasing feature sizes of hardware lead to higher variability in the electrical properties of hardware. Today's CPUs have a variation in operating frequency of about 30% which is dealt with by using die binning, i. e., testing the resulting chips to find their appropriate operating frequency. However, the variability will increase further with decreasing feature sizes because of the following reasons [Bor05]:

Higher variability in feature properties

dopant variation The threshold voltage of transistors is controlled by dopants inserted into the transistor channels. The smaller the transistors become, the less dopants are inserted. Thus, variations in the amount of dopants have a greater impact onto the electrical properties of the transistors. So, the uncontrollable variability of the production process leads to unpredictable properties of transistors.

subwavelength lithography Nowadays, the wavelength of the light used in lithography is bigger than the produced structures. That makes the structures unpredictable rough and uneven resulting in variations in the electrical properties of the produced transistors.

varying heat flux How much heat is produced highly depends on the functionality of a building block and thus varies across the die. Since the transistor's electrical properties are influenced by heat, transistors will have varying properties. Varying heat flux has a higher impact on smaller transistors than on larger ones.

Increasing variability will make processor designs, at least as done today, more and more unpredictable. It will not only increase the amount of hardware building blocks that fail altogether, but increased variability will also lead to more building blocks that are marginally functional. These are susceptible to soft errors and have a higher probability to fail altogether during the hardware's lifetime [CCL+08].

Increasing soft error rate

Decreased feature sizes and increased system sizes also lead to increased soft error rates. The problem of soft errors caused by radiation is known since the 1950s of the previous century. For example, soft errors were observed at locations near nuclear bomb test sites. Since then soft errors are continuously researched and tried to make hardware less susceptible to them [ZCM+96]. Borkar expects the SER (soft error rate) to increase exponentially with every new technology generation [Bor05].

One main cause for soft errors is radiation that causes energetic particles to travel through processors. These particles are responsible for inducing soft errors by for example changing the state of a transistor. Soft errors can be caused by three different kinds of radiation [Bau05]:

Alpha particles are emitted by impurities in the packaging materials of chips that are undergoing radioactive decay. Examples are uranium and thorium impurities. While emitted alpha particles travel through the processor, they transfer part of their kinetic energy into the surrounding material. Thereby, they can induce state changes [Bau05].

Since the discovery of the effect of impurities, they were reduced immensely. However, it is not possible to exclude impurities that emit energetic particles completely. Thus, alpha-particle-induced soft errors have to be considered when building reliable systems [KHP04].

High-energy cosmic radiation reacts with the Earth's atmosphere. This reactions emit cascades of secondary particles – mostly high-energy neutrons – that can also induce soft errors. The rate of these soft errors depends on altitude and geographical location.

According to Baumann [Bau05] high-energy neutrons have a higher potential to cause soft errors than alpha particles. Furthermore, it is difficult to protect from neutrons by shielding. Only thick layers of concrete reduce their impact [Bau05].

Low-energy cosmic radiation results in low-energy neutrons – so-called thermal neutrons. Boron is used as a dopant to control the threshold voltage of transistors. Low-energy neutrons can cause soft errors when encountering a specific boron isotope.

According to Baumann [Bau05] these soft errors can be easily prevented because the concerned boron isotope can be avoided easily. However, Wilkinson et. al. [WBB+05] reported that many integrated circuits still contain the concerned boron isotope.

Note that low- and high-energy neutrons are not exclusively caused by cosmic radiation. For example, cancer-radiotherapy equipment also emits low-energy

neutrons.

Decreased feature sizes result in decreased supply voltages. Thus, less energy, that is, less radiation, is required for causing a state change. The measurements presented by Constantinescu [Con03] indeed show a higher SER for memories operated at a lower supply voltage. Karnik et. al. [KHP04] observed that the error rate grows faster than the supply voltage decreases: *"The SER increases by 2x when the voltage is reduced from 1.2V to 0.8V. Over the range of measurements, the SER increases by 18 percent for every 10 percent reduction in the supply voltage."* Thus, in the future not only hardware operated under extreme conditions will be susceptible to soft errors. Commodity hardware under normal conditions will also be prone to radiation induced soft errors.

Furthermore, the increased system size leads to increased error rates for systems. Smaller transistors are less likely hit by an energetic particle and indeed the error rates per bit of memory are decreasing [Con03, DHW09]. However, the number of transistors per system follows Moore's Law and grows exponentially over time. Thus, the increased number of transistors increases the probability that any of them is hit by an energetic particle. Constantinescu from Intel [Con03] has shown that the SER of SRAM and DRAM increases linearly with their size. Dixit et. al. [DHW09] (working with Sun Microsystems) also confirm higher SERs for microprocessors with more memory. Dixit et. al. also state that the decreased SER per bit in RAM is partly caused by a change in design and not only by the reduced feature size. The new design was more lithography friendly and the resulting memories thus less susceptible. Memories in logical circuits such as registers have a less regular and thus less lithography friendly structure. Indeed, starting with the 90nm generation Dixit et. al. observed a higher SER for memories used in logical circuits than for SRAM.

There are various other causes for soft errors whose impact is also increased by decreasing feature sizes because they always go hand in hand with decreasing supply voltages:

- noise on the power supply
- charge sharing
- crosstalk and crosstalk-induced delays
- electromagnetic interference
- electrostatic discharge

Changing error model

Technology changes such as the feature size reduction do not only affect error rates observed, but also the kind of induced errors. For a long time single bitflips were a well-established error model. But with decreasing feature size multiple bitflips become more probable [DHW09].

Higher impact of physical effects

Furthermore, the reduced feature size increases the impact of physical effects such as the Miller and the skin effect. Thereby SER is also increased. The *Miller effect* means that the simultaneous switching of both terminals of a capacitor will modify the effective capacitance between the terminals. Thus, it can significantly affect on-chip delays [SK99]. According to the *skin effect* signals mostly propagate along the surface of wires at high frequencies. Because of the skin effect the resistance of wires varies with the signal frequency [Wal00].

Increasing hard error rate

Smaller features are also less reliable because decreased feature sizes increase the effects of transistor aging. While today these effects are factored in during design, this will not be possible anymore in the future [Bor05]. The amount of transistors that are either permanently broken or exhibit a borderline behavior will increase.

Possible causes for hard errors whose impact is increased by decreasing feature size are [BSO05]:

- *Electromigration* is the transport of material caused by the gradual movement of the ions in a conductor. It eventually results in highly resistive interconnects or contacts and leads to open circuits.
- *Gate oxide breakdown* is caused by the progressive build-up of defects inside the gate oxide of a transistor. Eventually these defects line up and constitute a conductive path across the dielectric.
- As we will detail later smaller structures are more difficult to handle during manufacturing. Hence, during chip fabrication with smaller feature sizes more hard errors are introduced.

Design errors

A further type of hard errors are design errors. Due to the increasing complexity of processors, it becomes more and more difficult to ensure their correctness [WA01]. Design errors are even worse than hard errors caused by randomly failing circuits because they are not detectable with a redundant execution that uses the same circuit.

Avižienis and He [AH99] studied the erratas of several large manufactures. These erratas describe known design faults in processors. Avižienis and He conclude that processors contain surprisingly many faults and that even for processors that are already sold for years still new bugs are found. However, for a large part of those bugs the manufactures do not intend to fix them. Furthermore, obtaining information about the faults of a processor is not easy. Erratas are confusing and difficult to read and not all manufactures provide them. Thus, users of hardware that shall be reliable should use means to detect hardware errors – even permanent ones.

2.2.2. (Un)Reliability of Hardware

After describing the reasons for increasing soft and hard error rates, we will now present the state and expected future development of the reliability of the building blocks of computing systems: memories and logical circuits.

Sensitivity of memory

Two different kinds of random access memory (*RAM*) are used: DRAM and SRAM. *Dynamic RAM (DRAM)* stores each bit of data in a separate capacitor and, thus, has to be refreshed regularly. It is usually used as main memory of computers. *Static RAM (SRAM)* uses bistable latching circuitry to store each bit and is not required to be refreshed regularly. It is used, for example, for caches, registers, and buffers in hard disks, routers etc.

The SER per bit of DRAM has reduced by more than 1000 times over seven generations. However, the SER of DRAM-based systems has remained unchanged because of the increased memory usage [Bau05].

Schroeder et. al. [SPW09] observed the DRAM error rates in Google's server fleet for 2.5 years. The observed memory error rates were with *"25,000 to 70,000 errors per billion device hours per Mbit"* [SPW09] much higher than expected. Furthermore, the results indicate that memory errors are not dominated by soft but by hard errors. These results contradict statements of Baumann that *"DRAM is one of the more robust devices"* [Bau05] and that we have to worry more about SRAM's reliability instead of DRAM's.

According to Baumann [Bau05] SER per bit of SRAM was increasing for a long time with every new generation. After removing the boron isotope that is susceptible to thermal neutrons from SRAM production, single-bit SER of SRAM stopped to increase and might even decrease in the future. However, due to increased usage of SRAM in systems, the SER for whole SRAM-based systems still increases with each new generation. Note that Baumann drew these conclusions from data of SRAM that already used error correcting codes (ECC) for memory error detection and correction. ECC is not commonly used in commodity systems. It is the opinion of Calhoun et. al. [CCL+08] that the error rates of SRAM will not be manageable by conventional memory protection techniques in the future.

Memory is not only composed of state storing elements, but does also contain logical circuits for accessing memory cells. As Karnik et. al. [KHP04] state *"Unlike a memory, the core logic often is not designed as a regular array and does not lend itself to ECC protection."*. According to Baumann [Bau02] the SER of the logical parts of memories is higher than that of memory that is additionally protected. He states that the SER of memory logic is only 100 to 1000 times lower than for unprotected SRAM.

Sensitivity of logical circuits

For a long time memory SER determined system SER to a large extent. However, logical circuits contain apart from combinational circuits also state storing elements such as latches and flip-flops. For them similar error rates as for SRAM are expected [Bau05]. Protecting these storage elements from undetected data modifications is a complex and expensive task.

Combinational circuits such as NAND- or NOR-gates are more robust against soft errors due to various masking effects that prevent soft errors from propagating. The following kinds of masking occur [KHP04]:

- *Logical masking* occurs if the error changes a state that is not used. In that case, the error has no influence. For example, if one input of a NAND-gate is modified, that does not change the gate's output if the other input is 0. In that case the output of the gate will be 1 – independent of the second erroneous input line.
- *Temporal masking* occurs if the error changes a signal or computation that is not currently latched, that is, transfered into a storage element such as a flip-flop.
- *Electrical masking* occurs if the introduced electrical pulse fades while traveling through a circuit and evaporates before reaching a storage element.

However, reduced feature sizes make combinational circuit's design less predictable. Interactions between adjacent circuits cannot be excluded and faulty

or nearly faulty transistors do not behave as expected. Furthermore, reduced propagation delays and increased clock frequency decrease the effect of temporal masking. The reduced supply voltage reduces the effect of electrical masking because smaller pulses suffice to modify state. Thus, it becomes more likely that electrical pulses introduced in a combinational circuit directly or indirectly modify the state of a storage element. Furthermore, the increased amount of combinational circuits on a chip makes it also more probable that any of them experiences a soft error [SKK+02, Bau05].

Will hardware become more reliable again?

Many of the problems that render hardware unreliable are identified today. Will that influence hardware design and manufacturing in a way that hardware will become more reliable instead of less reliable? Will new computer architectures and technologies replace current ones?

As one solution or part of the solution manufacturing could be improved. However, according to Baumann [Bau05]: *"The majority of process solutions seldom reduce SER by more than five times so their use does not justify the expense of additional process complexity, yield loss, and substrate cost."* Some modified processes provide a reduction by the factor 250. That still is not enough for high-reliability applications and does not justify the increased production costs [Bau05].

Hardware producers are right now struggling with developing new design processes that are able to handle at least variability. Currently it is unsuccessfully tried to handle variability by decoupling of design and technology [CCL+08]:

- Nowadays chip designers are restricted by *design-for-manufacturing rules* that depend on the used technology, e. g., 90nm or 65nm. These rules prevent the designers from designing circuits that in manufacturing will lead to a large amount of unusable dies. However, the set of rules keeps increasing. New rules often contradict old ones and adhering to all rules leads to less efficient designs in terms of power, timing, and area. Furthermore, the rules do not solve the problem completely. Designs often have to be refined after producing the first circuits.
- Furthermore, *standard cell libraries* are used to guarantee manufacturability. For each cell of such a library it is guaranteed that it can be manufactures correctly, and it is assumed that cells can be placed next to each other without influencing each other. However, with decreasing feature size interaction between neighboring cells increase. Analyzing manufacturability of all possible combinations of standard cells is an intractable task due to the amount of possible combinations and interactions.

Calhoun et. al. [CCL+08] discuss how future hardware development processes have to integrate the new design parameters introduced by the higher variability. The resulting design processes are much more complex and possibly costly than todays.

In contrast, Borkar in [Bor06] argues that any new technology would have to reduce costs while providing increased integration capacity and improved performance. However, currently discussed approaches for increasing reliability of circuits will always increase design and/or manufacturing costs.

2.3. Impact of Hardware Errors

This section will demonstrate the impact hardware errors can have. The usage of computing systems keeps increasing. They nowadays influence all aspects of our live – including our safety and financial well-being. We trust computing systems to function correctly. Hence, their impact and the impact of their failure on our live is tremendous.

As already described in the *Introduction* in Chapter 1 the impact of hardware errors depends on several factors such as error rate per device, number of deployed devices, utilization, and consequences and costs of a failure. In large scale systems even relatively low error rates per device can be unacceptable because with a high number of deployed devices the probability that any of them fails is non-negligible and grows with the number of devices. Safety-critical systems, where a failure can have catastrophic consequences, also demand very low error rates even if the number of deployed devices is relatively low.

In this section we will present studies and reports that describe the risk that is introduced by hardware errors and the impact hardware failures had.

Impact on safety

Safety of a computing system is *"the absence of catastrophic consequences on the user(s) and the environment"* [ALRL04]. Hardware failures can lead to the loss of safety. The New York Times reports that 5% of the mistakes in radiation therapy are partly or all together caused by hardware errors [Bog10]. These errors are dangerous and can be fatal as is demonstrated for some cases in [Bog10].

Another example where safety of patients is endangered is presented by Wilkinson et. al. [WBB+05]. They show that cancer-radiotherapy equipment emits thermal neutrons that cause soft errors in surrounding computing systems. These computing systems, for example, control the used radiation dosage and could cause a dangerous overdosage. Soft errors caused by thermal neutrons could be prevented by removing the neutron-susceptible boron isotope used in processors. However, Wilkinson et. al. did also analyze processors used nearby to cancer-radiotherapy equipment. They found the neutron-susceptible boron isotope in each of the processors analyzed.

Impact on security

Security is defined as *"a composite of the attributes of confidentiality, integrity, and availability, requiring the concurrent existence of 1) availability for authorized actions only, 2) confidentiality, and 3) integrity"* [ALRL04]. As we will show in the following undetected hardware errors can endanger security considerably.

Several studies [XCKI01, CXIW02, CXK+04] injected soft errors into the text segment of the process image of security-critical programs. Xu et. al. [XCKI01] injected the errors into the login procedures of ftpd (the daemon program for Internet File Transfer Protocol) and sshd (the daemon program for the Secure Shell Protocol). They observed a probability for security compromises of 1.1% for ftpd and of 1.5% for sshd. Considering the widespread usage of these tools that is considerable. Chen et. al. [CXIW02, CXK+04] also injected soft errors into the text segment of the process image of applications. Their target were two different firewalls. The observed result is that 2% of the injected errors

caused security vulnerabilities. Furthermore, the authors simulated a network with 20 firewalls. They defined a security violation as more than five otherwise unallowed packets entering the system. Under these assumption, the results show that a firewall failure rate of 2% leads to approximately two machines experiencing security violations within one year.

The studies presented in [BDL97, BDH+98, BDL01] show how cryptosystems such as RSA, ElGamal, and DSA can be broken under the presence of hardware errors. Hardware errors enable an attacker to obtain the secrets used in the cryptosystems with much less effort. Many of the attacks described in [BDL97, BDH+98, BDL01] require the ability of the attacker to modify specific bits. But as described in [SA03] the content of single SRAM cells can even be modified using a simple flashlight.

In [GA03] it is shown how soft errors help an attacker to gain control over a Java Virtual Machine (JVM). The authors demonstrated that an attacker can with a probability of 70% circumvent the type-based security barrier of the JVM if a soft error in the memory happened. The attacker can either wait for a real one or if he has access to the system induce one by himself. If the type-based security barrier is circumvented, the attacker can completely take over the JVM and execute arbitrary code.

Obviously not only our safety is endangered by unreliable hardware but also our security. However, unreliable hardware also can have immense economic impact for enterprises and individuals as well.

Economic impact

The financial losses induced by hardware errors can be tremendous and they are already occurring. In 2008, a bitflip in a message caused several hours of downtime for Amazon's servers in the US and the EU [Ser08]. The Los Alamos National Laboratory's had to expensively track down failures caused by soft errors when beginning to use their new ASC Q supercomputer [MHH+05].

2.4. Conclusions from the State of Hardware Reliability

The presented data confirms our claim that mechanisms are required that enable us to build computing systems using unreliable hardware. We have to ensure that we can depend on systems that are build using unreliable hardware. Therefore, we must not only ensure that the executed software is dependable[1], but we also must ensure that the software is executed correctly with a probability that is high enough for the intended application. While the consequences of undetected hardware failures might be only annoying for mobile phones, they might be fatal in cars or public transport or they might cause financial disasters when destroying important data in the data base backend of a banking solution.

To handle hardware failures, we first have to detect them. Thus, mechanisms to detect hardware errors are required. Preferably, these mechanisms are flexible, easy to use, and adaptable to the safety requirements of their user. These features can be achieved by implementing the error detection in software.

[1] That is not part of this thesis.

Furthermore, software-implemented hardware error detection eases the replacement of hardware in safety-critical systems. These systems usually are certified to one of the many functional safety standards such as IEC 61508[2] or ISO 26262[3]. Using a certified software-implemented hardware error detection will ease the recertification of the safety-critical system with the new hardware.

2.5. Software-level Symptoms of Hardware Errors

Software-implemented hardware error detection mechanisms should be hardware independent. For example, no assumptions should be made with respect to the failure modes of the underlying hardware. Hardware independence eases the applicability of the error detection in new systems because no special hardware has to be used and the detection mechanism needs not to be adapted to the hardware used.

To evaluate hardware independent error detection mechanisms, we need a hardware independent error model. Thus, we developed the following error model that describes the symptoms hardware errors can cause in software at the assembler level. It is an extension of the hardware-independent error model presented by Forin in [For89] and comprises the following symptoms:

Exchanged operand A different but valid operand is used, that is, instead of the intended operand another operand is used.

Exchanged operator A different operator is used, for example, an addition is executed instead of a subtraction. The operands remain the same.

Faulty operation An operator such as addition or subtraction does not work as expected and produces incorrect results despite of correct input values. Every usage of the result produced is influenced by this error.

Lost update A store operation to a register or memory location is omitted. This can result in the usage of out-dated values later on.

Modified operand An operand used by an instruction is modified by a single or a multiple bitflip. In contrast to a faulty operation, this error only influences one read of a value.

Further errors can be represented by combinations of these symptoms. For example, the replacement of a complete instruction comprised of operator and operands with a different one can be emulated using the symptoms *exchange operator* and *exchange operand*. Control flow errors can be emulated by combinations of instruction replacements.

This error model is based on the assumption that every hardware error that is not masked influences the execution of a program in some way and that all possible influences can be emulated by these basic symptoms.

[2]The international standard for *Functional safety of electrical/electronic/programmable electronic safety-related systems.*

[3]The international standard for *Road vehicles – Functional safety.*

We use this error model to develop and assess the in this thesis presented error detection approaches. First, it guides us in the selection of an arithmetic code for our error detection mechanism. Second, we use it to evaluate the mechanisms implemented.

3. Arithmetic Codes

One long known technique to detect hardware errors at runtime is the application of arithmetic codes to detect execution errors. Encoding with an arithmetic code adds redundancy to all data words. The result is a larger domain of possible data words. Of these possible data words only a subset are valid code words. Figure 3.1 on page 20 demonstrates the concept of arithmetic codes.

For using an arithmetic code, we have to encode applications, i. e., they have to be enabled to process encoded data. When an application is encoded using an arithmetic code, it will solely process encoded data, that is, all inputs have to be encoded and all computations use and produce encoded data.

Thereby, arithmetic codes facilitate an end-to-end protection of computations. Errors disturbing storage, transport, and processing are detectable with the same (arithmetic) code. Furthermore, in contrast to redundant execution, arithmetic codes also detect permanent hardware errors.

The goal of this chapter is to introduce arithmetic codes and to motivate our choice of an arithmetic code to implement hardware error detection. Therefore, this chapter first introduces arithmetic codes in general. Second, we introduce several arithmetic codes and discuss their advantages and disadvantages. We will compare the different codes with respect to the operations that these codes support and how many of the symptoms described in Section 2.5 they can detect. We will not discuss with which probability they detect these symptoms. This probability depends on the choice of the code parameters and can for most of the codes be chosen arbitrarily high. By discussing these codes, we motivate our choice to use AN-codes and their extensions to implement our error detection mechanisms. We conclude the chapter by summarizing our comparison.

Arithmetic code: informal definition

Arithmetic codes[1] are a long known technique to detect hardware errors at runtime. Arithmetic codes add redundancy to processed data. Thus, a larger domain of possible data words is created. The domain of possible words contains the smaller subset of valid code words – the so-called *encoded data* items. Arithmetic codes are preserved by correct arithmetic operations, that is, a correctly executed operation taking valid code words as input produces a result that is also a valid code word. On the other hand, faulty arithmetic operations do not preserve the code with a high probability, that is, faulty operations most likely result in a non-valid code word [Avi71]. Arithmetic codes protect data also from unnoticed random modifications during storage or transport on a bus. These random modifications will result in an invalid code word with high probability. Thus, they are detectable by checking the code.

[1] Note that some codes for lossless data compression are also called arithmetic codes. These are not equivalent with the ones used throughout this thesis.

Note that the probability with which an arithmetic code in the end detects errors depends on the choice of the code parameters. For AN-codes we discuss the parameter selection in Chapter 5.

Figure 3.1 depicts the general concept of arithmetic codes. It demonstrates the following examples:

1. One correctly executed operation that takes two valid code words as input and produces a valid code word as output.
2. One correctly executed operation that uses one valid code word and one non-code word as input and, thus, produces an invalid output.
3. One operation whose execution is faulty and that produces an invalid result despite the two valid code words that were used as input.

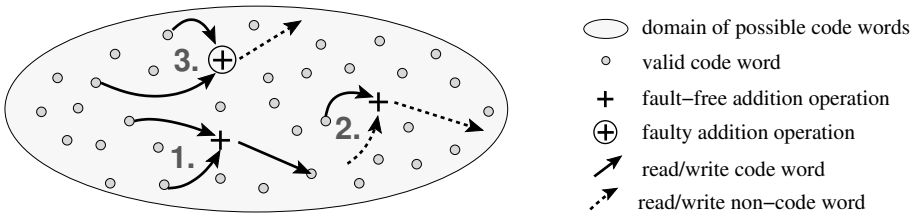


Figure 3.1.: Concept of arithmetic codes.

Formal definition

A formal definition for arithmetic codes was given by Avižienis in [Avi71]: "*Given the digital number representations, x, y, an arithmetic operation $*$, and an encoding $f : x \to x'$, we say that f is an arithmetic-error code with respect to $*$ if and only if there exists an algorithm $A*$ for coded operands to implement the operation $*$ such that $A*(x', y') \equiv (x * y)'$.*" This definition can be easily extended to single-operand and multi-operand operations.

In contrast to Avižienis, we use the following terminology and notations:

- x_c denotes the code word representation of x (denoted as $f(x) = x'$ above). We call x_c the *encoded data* or *encoded value*.
- The non-code word version of x_c, that is x, we denote as *unencoded* or *functional value*.
- The operation $*$ that processes the unencoded values we denote as *unencoded operation*.
- We call the operations $A*$ that preserves the arithmetic code *encoded operation* and use the notation $*_c$. Realizing $*_c$ we denote as *encoding* the operation $*$.

On the following pages we will introduce several arithmetic codes and compare them according to the criteria: supported operations, detected errors, and kind of implementation.

The set of supported operations of an arithmetic code is the set unencoded operations $*$ for which an encoded version $*_c$ exists. Rao [Rao74] demands that every arithmetic code should at least support addition. However, the more operations are directly supported by an arithmetic code, i. e., without the need to emulate the operation using other (encodable) operations, the more efficient

the implementation of an arithmetic code can be. We will examine which operations are directly supported by an arithmetic code. Note that none of the arithmetic codes known to us supports floating point operations. Thus, we will in the following only consider integer operations. For encoding floating point operations we will use an integer-based implementation (see Section 4.2.3).

Of course the error detection capabilities of an arithmetic code are also important. Ideally, the code supports detection of all of the errors defined in Section 2.5. Depending on the criticality of the intended application we can cut back on the provided safety, i. e., the amount and kind of detected errors. Thereby, we might gain a reduction in the runtime overhead generated by the code.

Last, we discuss how the presented codes can be implemented. Because of the costs and inflexibility of custom reliable hardware, we aim at a software-implemented solution. Thus, we prefer arithmetic codes that we can implement in software.

Classes of arithmetic codes

Furthermore, we will classify the codes presented according to one of the following possible classes:

- non-separate and non-systematic codes,
- separate and systematic codes, and
- non-separate but systematic codes.

Rao's definition of a *systematic* code is: *An arithmetic code, which has each codeword represented by, say n digits, is systematic if there exists a set of k digits ($k < n$) of the codeword representing the information* [i.e., the functional value] *and the remaining $n - k$ digits representing the check(s).* [Rao74]

If the encoded value is a tuple of the functional value and its check bits and for encoded operations these check bits are computed in parallel to the functional value, this code is *separate*. For a *non-separate* code functional value and check bits are processed together by the same operation. Separate codes are always systematic. Hence, the class separate and non-systematic is empty [Rao74].

Systematic codes have the advantage that the functional data can be used directly and no decoding is required but only code checking. Systematic but non-separate codes are especially useful because they do not require additional structures for processing the check bits and still the data can be directly used without an additional decoding step.

In the following we will give a short introduction into the main groups of arithmetic codes. Especially for AN-codes, we will present more details than absolutely necessary because some of the literature describing these codes is from the 1970s and often not available.

3.1. Berger Code

A code word in the Berger code consists of the functional value and a redundant part. The redundant part – the *check bits* – is the binary representation of the number of zeros contained in the functional value. For example, the check bits

of the 8 bit binary value 10110011 are 0011 because the value contains three zeros. There are 4 check bits because the functional value might contain up to 8 zeros. A code word is valid if the value of the check bits equals the number of zeros contained in the functional value.

The Berger code is a systematic and separate code. The functional value can be directly read from the code word. However, the redundancy has to be computed separately, i. e., we cannot just feed the functional and the redundant part together into an operation and expect the result to contain the correct functional value and check bits. Instead, the check bits have to be computed separately.

A further development of the Berger code is the Bose-Lin code presented in [BL85]. It provides the same error detection capabilities as the Berger code, but according to [DT99], the Bose-Lin code requires fewer check bits and the implementation of the code check is much simpler.

Supported operations

As the redundant part of the code words is computed separately, in principle any operation can be supported as long as the number of zeros contained in the result can be determined from the input values. In the worst case, this requires recomputing the checked operation and counting the zeros of the redundant result that was obtained. However, that is simple redundancy that provides no protection against design faults. Usually, in the hardware implementation of the Berger code, the redundant bits of an operation's result are computed by redundant hardware that is simpler than the checked operation. This kind of implementation is less susceptible to common cause errors.

To the best of our knowledge, encoding control flow statements with the Berger code is not possible. These would have to be protected by other means. Data flow is protected to some extend because the matching check bits have to be used in each computation.

Implementation

For most operations the hardware implementation of the Berger code is safer and more efficient than its software implementation. For example, if we want to compute the check bits of the result of an addition, we have to track all carries generated during the addition of the inputs. In hardware, this can be done very efficiently using an extended adder circuit. In software, this actually means a redundant and either less safe or less efficient recomputation of the addition.

Error detection capabilities

Of the error model described in Section 2.5 the Berger code can detect: faulty operations and modified operands. Depending on the actual implementation it can also detect exchanged operands and exchanged operators because the check bits used have to match the functional values used and the check bit computation for the result has to match the operator used. Thus, there is redundancy that can facilitate detection of exchanged operands or operators.

The Berger code cannot detect lost updates. Furthermore, the Berger code is known to detect only unidirectional errors, i. e., all flipped bits in one word flip in the same direction [LOBR09]. If, for example, one bit flips from 0 to 1 and another bit flips from 1 to 0, this is undetectable with a Berger code. This is a heavy limitation considering that multiple bitflips are becoming more probable with decreasing feature sizes [DHW09]. Note that also the Bose-Lin code can only detect multiple flipped bits if they do not neutralize each other.

Systems using this code

All applications of the Berger code that are known to us are indeed implemented in hardware. In [LTRN92] the Berger code is used in the design of a self-checking arithmetic logical unit (ALU). The authors present hardware-implemented check bit prediction algorithms for addition, two's complement subtraction, logic operations, the shift, and the rotate operation.

The authors of [LOBR09] also present a self-checking ALU that is realized using the Berger code. Additionally, this ALU does also provide correction of transient errors using the code to detect errors and a redundant execution to correct errors that occurred.

We will not use the Berger code for our software-implemented error detection for the following reasons:

- The Berger code is not sufficient for an implementation in software.
- It does not facilitate the detection of all error symptoms that we defined in Section 2.5 and might not detect multiple bitflips.

3.2. Residue Codes

Residue codes are systematic and separate codes. The code word for the functional value x is the tuple of x and its residue to a code specific constant A that is greater than 1. Thus, the encoded version of x in a residue code is

$$x_c = (x, x \mod A) = (x, x_A) \quad \text{with} \quad A > 1.$$

The code parameter A is used to adjust the detection capability of the code. The larger A is chosen, the less probable are undetected errors because the less functional values have the same residue.

A code word is valid if the check bits equal the modulus of A of the functional value, that is, if the following equation holds:

$$x \mod A = x_A.$$

Note that the check bits x_A exist redundantly to x. Thus, if using a residue code, we have to execute additional operations to compute the results's check bits for each operation. Hence, x and x_A of the tuple $x_c = (x, x_A)$ can be used to check the validity of x_c.

Further known residue codes are multiresidue codes and inverse residue codes. Multiresidue codes, i. e., residue codes that use multiple different residues, can be used to implement error correction as shown in [Rao70] and [Rao74, chapter 5]. For inverse residue codes the check bits are formed as $A - x_A$. According to [Avi71] these codes are better in detection of repeated-use faults, i. e., faults where a stuck bit is used several times. This kind of fault may occur in circuits that implement shift operations.

Supported Operations

Table 3.1 summarizes which operations are supported by residue codes and which not. The table depicts for the operations supported how the functional value and the check bits, i. e., the residue, are computed. Of course many operations could be implemented using other operations and programming constructs such as loops and branches. The table only presents solutions that apply basic arithmetic and logical operators to the input residues. In particular solutions that require a loop or branch are not presented.

encoded operation	**implementation**	
	functional value	**check bits**
arithmetic operations:		
$z_c = x_c +_c y_c$	$z = x + y$	$z_A = (x_A + y_A) \mod A$
$z_c = x_c -_c y_c$	$z = x - y$	$z_A = (x_A - y_A) \mod A$
$z_c = x_c *_c y_c$	$z = x * y$	$z_A = (x_A * y_A) \mod A$
$z_c = x_c /_c y_c$	$z = x/y$	*not directly encodable*
signed numbers: *supported*		
shift operations:		
$z_c = x_c <<_c y_c$	$z = x << y$	*not directly encodable*
$z_c = x_c >>_c y_c$	$z = x >> y$	*not directly encodable*
logical boolean operations:		
`or:` $z_c = x_c \,\|\|_c\, y_c$	$z = x \,\|\|\, y$	$z_A = x_A + y_A - x_A * y_A$
`and:` $z_c = x_c \,\&\&_c\, y_c$	$z = x \,\&\&\, y$	$z_A = x_A * y_A$
`not:` $z_c = !_c\, x_c$	$z = !\, x$	$z_A = 1 - x_A$
bitwise boolean operations:		
`or:` $z_c = x_c \,\|_c\, y_c$	$z = x \,\|\, y$	*not directly encodable*
`and:` $z_c = x_c \,\&_c\, y_c$	$z = x \,\&\, y$	*not directly encodable*
`not:` $z_c = \sim_c\, x_c$	$z = \sim\, x$	*not directly encodable*
comparisons: *not supported*		

Table 3.1.: Implementation of encoded operations for residue codes.

Note that we must not use the functional value z of the result for computing the check bits z_A. That would be nothing else than a redundant computation instead of an arithmetically encoded one. Of course that could be always used as a less safe fallback solution. This solution is less safe because redundancy is susceptible to permanently faulty hardware[2].

While the redundant computation of the check bits for addition, subtraction, and multiplication is easily done, we know no solution for the division. Of course the division can be emulated expensively using a loop that subtracts the divisor from the dividend until zero is reached.

The supported arithmetic operations addition, subtraction, and multiplication

[2]Our error injection results presented in Section 8.5 confirm this statement.

support positive and negative numbers as well. Thus, arithmetic with signed and unsigned numbers can be realized using a residue code.

The computation of the check bits for the left shift operation $x << y$ seems to be no problem. Because a left shift is equivalent to a multiplication with a power of two, the residue of the result is $(x_A * 2^y) \mod A$. However, for the computation of 2^y the functional value y is used directly instead of its residue y_A. Thus, any error leading to a modification of y will not be detectable. For encoding the left shift an encoded version of the computation of the power of two is required. This also has to be emulated, for example by multiplications with two in a loop. However, residue codes can only protect the multiplication, but neither the required comparison nor the loop itself.

The right shift operations are equivalent to a division with an appropriate power of two. However, since we do not know a way to compute the residue for the division operation directly, we also do not know a way to compute the residues for right shift operations. Additionally, the right shift also requires the encoded version of the power-of-two computation.

Boolean logical operations are easily implemented using the knowledge how to emulate these operations using arithmetic operations. In contrast to the ANSI-C standard, this implementation makes it necessary to restrict boolean values to 1 representing `true` and 0 for representing `false`. Note that the remainders x_A and y_A of the boolean values x and y are equal to x and y, that is, either equal to 0 or to 1. Thus, the implementations of the boolean operations for the residue computations are nothing else then a redundant, different implementation.

To summarize, to the best of our knowledge, directly encoding division, bitwise logical operations, comparisons, and control flow statements with a residue code is not possible. These either have to be protected by other means or have to be implemented using operations for which an encoded version exists.

Implementation

Residue codes can be implemented in software and hardware as well. Let us look at the following function `foo` implemented in pseudocode similar to C:

```
int foo (int x, int y, int z){
   int u=x+y;
   int v=u+z;
   return v;
}
```

If this function is protected by a residue code implemented in software, it would in parallel compute the residues of all computation results like the following function `foo_c` does:

```
(int, int) foo_c (int x, int y, int z,
                  int xA, int yA, int zA){
   int u=x+y;
   int uA=(xA+yA) % A;      // u % A
   int v=u+z
   int vA=(uA+zA) % A;      // v % A
   return (v, vA);
}
```

Error detection capabilities

Residue codes can detect the following errors: faulty operations and modified operands because these will result in non-matching residues. They can also

detect data and control flow errors such as exchanged operands and exchanged operators to some extent because the residues, i. e., check bits, are computed separately. Two such errors need to neutralize each other if an exchanged operand or operator shall be not detectable. If in the above example `u` is erroneously replaced by `x`, then either `x` has to have the same residue as `u` or `uA` also has to be replaced with `xA`. Otherwise, the error will result in a mismatch between `v` and its expected residue `vA` with a high probability. Finally, residue codes cannot detect lost updates.

Systems using this code

We know only of one example application of residue codes. That is the fault-tolerant STAR computer [AGM+71] that was developed at the Jet Propulsion Laboratory in the 1960s and used an inverse residue code for error detection. In that computer the code was implemented in hardware. In first versions of STAR an AN-code (see the next section) was used. However, that was replaced because the hardware implementation of a separate residue code was more efficient.

We will not use residue codes for our software-implemented error detection because of the following reasons:

- Residue codes do not facilitate the detection of all error symptoms that we defined in Section 2.5.
- They provide no directly encoded version of division. However, for encoding right shifts an encoded division is required. Especially for bitwise logical operations and unaligned loads and stores shift operations will be needed and should not be slowed down additionally by an emulated division operation, that is, a division that is implemented using other directly encoded operations.

3.3. AN-Codes

AN-codes are non-separate and non-systematic, that is, functional part and redundancy of a code word are processed together and the functional value cannot directly be read from the code word.

AN-encoding is done by multiplying the functional value with a constant A whose impact on the error detection rate we discuss in Chapter 5:

$$x_c = A * x \quad \text{with} \quad 1 < A.$$

Only multiples of A are valid code words and every operation processing AN-encoded data has to preserve this property. Code checking is done by computing the modulus with A. For a valid code word it is zero:

$$x_c \mod A = 0.$$

The functional value x is obtained by an integer division $x = x_c/A$.

Supported Operations

Table 3.2 summarizes which operations are supported by AN-codes and which not. For the supported operations, their implementation is shown. Note that

we depict the expected content of the encoded variables. If we replace x_c by Ax, that does not mean that at runtime a multiplication with A is executed. In contrast, it means that the content of the variable x_c contains Ax.

encoded operation	**implementation**
arithmetic operations:	
$z_c = x_c +_c y_c$	$z_c = Ax + Ay = A(x + y)$
$z_c = x_c -_c y_c$	$z_c = Ax - Ay = A(x - y)$
$z_c = x_c *_c y_c$	$z_c = (Ax * Ay)/A = A(x * y)$
$z_c = \lfloor x_c /_c y_c \rfloor_c$	$z_c = \lfloor (A * Ax)/\, Ay \rfloor = A \left\lfloor \frac{x}{y} \right\rfloor$
signed numbers: *supported*	
shift operations:	
$z_c = x_c <<_c y_c$	*not directly encodable*
$z_c = x_c >>_c y_c$	*not directly encodable*
logical boolean operations:	
`or:` $z_c = x_c \,\|\|_c\, y_c$	$z_c = x_c +_c y_c -_c x_c *_c y_c$
`and:` $z_c = x_c \,\&\&_c\, y_c$	$z_c = x_c *_c y_c$
`not:` $z_c = !_c\, x_c$	$z_c = 1_c -_c x_c = A -_c x_c$
bitwise boolean operations:	
`or:` $z_c = x_c \,\|_c\, y_c$	*not directly encodable*
`and:` $z_c = x_c \,\&_c\, y_c$	*not directly encodable*
`not:` $z_c = \sim_c\ x_c$	*not directly encodable*
comparisons: *AN-encoded numbers can be compared directly. However, obtaining a valid encoded result requires an unprotected and, thus, unsafe if-statement.*	

Table 3.2.: Implementation of encoded operations for AN-codes.

Note further that the presented implementations of the encoded operations are simplified versions that do only describe the general idea. We specifically ignore the characteristics of arithmetic operations as they are implemented in usual processors. Thus, here we do not consider such things as overflow and underflow behavior for addition, subtraction, and multiplication, or divisibility for the division. We do also ignore signedness as far as possible. We will discuss the exact implementations of AN-encoded operations that consider all these details in Chapter 4.

In contrast to residue codes, AN-codes support division additionally to addition, subtraction, and multiplication. It is important that in the encoded implementation of the division the dividend x_c is multiplied with A before it is divided by the divisor y_c. Otherwise, for a short time the functional value z would be available and could be modified unnoticedly. Likewise, the division with A that is required for the multiplication has to be executed after multiplying the two

AN-encoded operands and must not be applied to one of the operands before the multiplication is executed. The latter solution would lead to a window of vulnerability where the operand that was divided by A could be modified undetectable.

Like residue codes, AN-codes as well support signed numbers. A detailed discussion of problems that result from the finite nature of the common digital number representation in processors are discussed in Chapter 4.

Encoded versions of arithmetic and logic shift operations can be implemented using division and multiplication with powers of two because $a << k$ is equivalent to $a * 2^k$ and $a >> k$ is equivalent to $\lfloor \frac{a}{2^k} \rfloor$. In contrast to residue codes, the implementation of the power-of-two function can be encoded more completely for AN-codes because AN-codes do also support encoded comparisons. However, AN-codes still leave the control flow, that is, the loop required in the power-of-two computation, unprotected.

The implementation of AN-encoded logical operations is equal to residue-encoded logical operations. Likewise, no directly AN-encoded versions of bitwise logical operations are known.

Implementation

AN-codes are equally suitable for implementation in software and hardware. It is also possible to combine software and hardware implementation by realizing some encoded operations in hardware and some in software. For implementations of AN-codes that implement the code checking in hardware an efficient algorithm for computing modulo A exists if A is chosen to be of the form $A = 2^a - 1,\ for\ a \geq 2$. Since the goal of this thesis is a software-implemented solution, we do not present the algorithm here. You can find it in [Avi64] or [Rao74].

If an AN-code is implemented in software, this requires to transform the programs that shall be protected from undetected execution errors. We must ensure that encoded programs solely process encoded data and do preserve the code in an error-free execution. Furthermore, encoded programs must not decode data during computations. The AN-encoded version[3] of the previously introduced function `foo` is:

```
int_c foo_c(int_c xc, int_c yc, int_c zc){
   int_c uc=xc+yc;    // uc = A*(x+y)
   int_c vc=uc+zc;    // vc = A*(x+y+z)
   return vc;         // expected: vc mod A == 0
}
```

Error detection capabilities

AN-codes can detect the following errors: faulty operations and modified operands. Lets assume that in the above example `foo_c` one of the additions is faulty or `xc` is hit by a bitflip. This will be detected with high probability because it is unlikely that a random error of that kind will result in another multiple of A.

However, AN-codes might not detect exchanged operands, exchanged operators, and lost updates, that is, AN-codes might not detect data and control flow errors. For example, variable `yc` might be exchanged by another encoded variable `ac`

[3]The presented pseudo code is simplified and ignores the over- and underflow issues we describe in [WF07a] and Chapter 4.

that contains a correctly encoded value $A * a$. That might be the result of a bitflip that happens on the address bus. An AN-code will not detect this because one multiple of A is replaced with another. Thus, the result `uc` also still will be a multiple of A. On the other hand, a bitflip in the instruction unit of a CPU might cause an operator error, for example, a subtraction could be executed instead of an addition. However, this will also not lead to an invalid code word because subtraction and addition as well preserve the AN-code.

For the influence of the choice of A on the error detection capabilities see Chapter 5.

Systems using this code

Two applications of AN-codes – both implemented in software – are ED4I [OMM02] and TRUMP [CRA06]. ED4I duplicates data and instructions and these duplicates are AN-encoded. All results of duplicate instructions have to be multiples of A of the original results. In this way, most hardware errors are recognizable. However, whenever a program contains logical operations, the authors choose an A that is a power of two to make those operations encodable. Thereby they reduce the detection capabilities immensely because encoding is then equivalent to shifting to the left. The resulting code cannot detect bitflips in the higher order bits of data values. However, these bits contain the original functional value.

Chang et. al. in TRUMP also used an AN-code but only for operations that can easily handle encoded values such as addition and subtraction. Furthermore, the encoding is only applied to registers and not to memory. In the end that leaves supposedly only small parts of applications which are AN-encoded. As should be expected their fault injection experiments show a non-negligible amount of undetected failures for most of the tested applications.

The literature presents several special AN-codes that, for example, promise to provide error correction or to be systematic. In the following sections we will introduce such special AN-codes and discuss their properties. Note that this thesis aims at error detection and not at correction. However, if correction could be easily supported by a code, we would prefer this code.

3.3.1. Error Correcting AN-Codes

AN-codes can be designed in a way that they do not only support error detection but also error correction. Several such AN-codes can be found in the literature: [Mas64, Chi64, Man67, Avi65, Rao70, RG71], and [Rao74, chapter 4]. The general idea is to choose A in a way that every correctable error pattern produces a distinctive modulus for A. An error pattern is composed of the original code word and the number, the position, and the direction of the bits flipped by the error.

These error correcting AN-codes can be used to correct transmission errors and errors occurring during storage of the encoded value. Thus, it is possible to correct modified operands. However, it is not possible to correct errors occurring during the execution of operations. Thus, the symptoms faulty operation, exchanged operand, exchanged operator, and lost update are not correctable.

However, all these codes seem to depend highly on the hardware architecture used. When these codes were developed in the 1960s, they were intended to be implemented directly in hardware. For example, several papers note that one's complement is required. However, today's processors usually use the two's complement. Furthermore, none of the papers presents an efficient algorithm to infer the original code word from the error syndrome. In the worst case that would require a tabulated approach mapping invalid code words to valid code words and, thus, be impossible to use.

3.3.2. Systematic AN-Codes

For every AN-code with $2^{m-1} < A < 2^m$ a systematic non-separate code can be found [Mas64]. This systematic AN-code is a subcode of the original AN-code, that is, not every code word present in the AN-code is present in its systematic variant.

For obtaining the systematic AN-code, the functional value x is encoded using the following rule

$$x_c = 2^m * x + (-2^m * x \mod A).$$

The multiplication $2^m * x$ shifts the functional value x to the left. The addition of $-2^m * x \mod A$ adds the check bits.

A code word is valid if the following value

$$(-2^m * x \mod A)$$

equals the check bits. The value for x used in this computation can be directly read from the code word because the code is systematic and x is redundantly contained in the check bits.

Because x_c is formed by adding the remainder of the division $\frac{2^m * x}{A}$ to $2^m * x$, it is a multiple of A. Thus, x_c is a code word of an AN-code. The code is systematic because the higher order bits contain x and the m least significant bits contain the check bits. Table 3.3 presents two example codes with $A = 7$ and $A = 3$ respectively.

According to Rao [Rao74, page 174], this code is not closed with respect to addition, that is, adding two valid code words might result in a non-code word. The problem is twofold. First, just adding the check bits is not enough. They actively have to be adapted to fulfill the code requirements again. Second, the carries of the check bits added propagate from the check bits to the information bits, which contain the functional value.

For an example, look into Table 3.3 at the code using $A = 3$. The addition of 001 10 (1) and 010 01 (2) results in 011 11. However, 011 11 is no valid code word. The result should be 011 00 (3). Even worse an addition of two valid code words might result in another code word that does not represent the correct

x	$\mathbf{A = 7}$		$\mathbf{A = 3}$	
	$\mathbf{0 \leq x < 16}$	$\mathbf{x_c = 2^3 * x + (-2^3 * x) \mod 7}$	$\mathbf{0 \leq x < 8}$	$\mathbf{x_c = 2^2 * x + (-2^2 * x) \mod 3}$
0	0000	0000 000	000	000 00
1	0001	0001 110	001	001 10
2	0010	0010 101	010	010 01
3	0011	0011 100	011	011 00
4	0100	0100 011	100	100 10
5	0101	0101 010	101	101 01
6	0110	0110 001	110	110 00
7	0111	0111 000	111	111 10
8	1000	1000 110		
9	1001	1001 101		
10	1010	1010 100		
11	1011	1011 011		
12	1100	1100 010		
13	1101	1101 001		
14	1110	1110 000		
15	1111	1111 110		

Table 3.3.: Examples for systematic AN-codes for different As. The first column contains the functional value in decimal representation. The remaining columns use binary representation.

result. For example, the addition of 0010 101 (2) and 0011 100 (3) in the code formed with $A = 7$ results in 0110 001 (6), which is a valid code word but not the correct result.

Thus, systematic AN-codes are unsuitable for practical use because for every addition these effects have to be corrected expensively. For multiplications even more corrections are required.

It remains to be said that Forin [For89] claims to use this kind of code in combination with signatures. But he lacks to provide further information on implementations of encoded operations. When we tried, every addition required extensive corrections to reconstitute the code property.

3.3.3. $|gAN|_M$ Code

Because of the problems of systematic AN-codes, Rao [Rao74] developed a systematic non-separate AN-code – the $|gAN|_M$ code – that is closed with respect to addition. In a $|gAN|_M$ code not the least significant bits contain the check bits but the most significant ones while the least significant bits contain the functional value.

In contrast to the systematic AN-codes of the previous section, $|gAN|_M$ codes are closed with respect to addition, subtraction, and multiplication.

If the code shall support functional values x in the range $0 \leq x < k$, Rao defines the one element, i. e., the element that represents the 1, of the code as

$$1_c = gA = C_1 * k + 1. \tag{3.1}$$

Rao, furthermore, requires that A and k are relatively prime, i. e., have no other common divisor than 1. Obviously, 1_c is a multiple of A.

Any other element of the code is defined as a multiple of the one element:

$$x_c = (x * gA) \mod (A * k) = (C_1 * k * x + x) \mod (A * k). \tag{3.2}$$

Using Equation 3.1 we know that $C_1 * k + 1 \mod A$ has to be zero. Thus, using the chosen values for k and A, we can determine C_1. For example, for $A = 5$ and $k = 8$ Equation 3.1 is fulfilled for $C_1 = 3$. Thus, in that case, the remaining code parameters are $g = 5$, $g * A = 25$, and $A * k = 40$. Table 3.4 depicts the resulting complete $|gAN|_M$ code for $k = 8$ and $A = 5$.

$\mathbf{0 \leq x < 8}$		$\mathbf{x_c = (25x) \mod 40 = \lvert 25x \rvert_{40}}$	
decimal	binary	decimal	binary
0	000	0	000 000
1	001	25	011 001
2	010	10	001 010
3	011	35	100 011
4	100	20	010 100
5	101	5	000 101
6	110	30	011 110
7	111	15	001 111

Table 3.4.: $|gAN|_M$ code for $A = 5$ and $k = 8$. Note that the 3 least significant bits contain the functional value x.

The encoded addition of two code words x_c and y_c is implemented as the normal addition modulo $A * k$: $(x_c + y_c) \mod A * k$. The encoded subtraction is defined as $(x_c - y_c) \mod A * k$ and the multiplication as $(x_c * y_c) \mod A * k$. However, in contrast to the normal AN-code, no encoded division operation is known for the $|gAN|_M$ codes. Furthermore, supporting signed, that is, also negative numbers, requires a special hardware implementation as it was described in [OÖ89]. This specific hardware processes sign information in parallel to the actual $|gAN|_M$-encoded values.

Note that the code was called $|gAN|_M$ by Rao because he defined $M = A * k$ and used N to denote the functional values. Furthermore, $|gAN|_M$ denotes the modulus of gAN with M. Thus, the code name $|gAN|_M$ is nothing else than the encoding function that we described in Equation 3.2.

Olivier and Özgüner [OÖ89] showed how a $|g3N|_M$ code combined with specific hardware can be used to protect computations in systolic arrays from unde-

tectable errors. However, they did only use addition and multiplication. The presented systolic array did not require a division.

3.3.4. Conclusions for AN-Codes

We will not use any of the special AN-codes presented because they provide either less capabilities than common AN-codes or they still require fundamental research before being usable. The latter one is the case for error correcting AN-codes. So far, it was only shown that they exist and under which conditions. However, their implementation – especially the implementation of error correction – was not described. Systematic AN-codes, on the other hand, are difficult to realize because many corrections are required additionally to the executed operation to produce valid code words. Last, $|gAN|_M$ codes seemed promissing. Yet, in contrast to common AN-codes they do not support signed numbers and the division operation.

But we will look into extensions of common AN-codes further because:

- AN-codes are the only arithmetic codes that we know of that support a directly encoded division operation in addition to subtraction, addition, and multiplication.
- They do not facilitate the detection of all error symptoms that we defined in Section 2.5. However, they can be extended to detect these symptoms also (see the following sections 3.4 and 3.5).

3.4. ANB-Codes

To solve the problem of undetectable exchanged operators and operands, Forin in [For89] introduced *static signatures* (which he referred to as *signatures* or *"B"s*). In the resulting *ANB-code*, the encoding of a variable `x` is defined as

$$x_c = A * x + B_x \quad \text{with} \quad 0 < B_x < A.$$

If two encoded values are combined for example by adding them, the result's signature depends on the signatures of the input values. This *expected signature* for the result is precomputed when the signatures are assigned to the input values of a program, that is, at encoding time.

To check the code of x_c, x_c's modulus with A is computed. The result has to be equal to the assigned or precomputed expected signature B_x of x_c. The functional value x is obtained by an integer division $x = x_c/A$.

Supported operations

ANB-codes support nearly the same operations as AN-codes. However, currently, we know no easy solution to encode a division with an ANB-code. Thus, we either have to emulate it using subtractions executed in a loop or to use the less safe AN-encoded variant.

ANB-encoding operations often requires additional corrections. These corrections ensure that the encoded operation produces a valid code word with a signature

that only depends on the signatures of the input values and not on their functional values. For example, the multiplication requires extensive corrections because $x_c * y_c$ does not result in the intended $A * x * y + B_x * B_y$ but in $A^2 * x * y + A * x * B_y + A * y * B_x + B_x * B_y$.

Implementation

ANB-encoding can be realized completely in software and also partly in hardware. A complete realization in hardware of course would be possible. However, assigning signatures and precomputing expected signatures is easier to realize in software. Yet, encoded programs could benefit of hardware implementations of encoded operations. These, could reduce the runtime overhead induced by the encoding dramatically.

Like with AN-codes, we have to ensure that ANB-encoded applications solely process encoded data items and preserve the code. The ANB-encoded version of our small example looks as follows:

```
int_c foo_c(int_c xc, int_c yc, int_c zc) {
  int_c uc = xc + yc;    // uc = A*x+Bx + A+y+By = A(x+y)+Bx+By
  int_c vc = uc + zc;    // vc = A(x+y+z)+Bx+By+Bz
  return vc;             // expected: vc mod A == Bx+By+Bz
}
```

When encoding the program represented by `foo`, we assign static signatures to the input variables `x`, `y`, and `z`. Knowing the program, we can precompute the result's expected signature $B_v = B_x + B_y + B_z$. B_v also has to be smaller than A and larger than zero. This can be ensured by correcting the signatures during program execution. The corrections required are also precomputed at encoding time for static signatures. We add them to the concerned encoded value at runtime in each execution.

Note that for implementing dynamically allocated memory, we introduce dynamic signatures in Chapter 7 and [WF07b]. These are assigned and precomputed at runtime instead of at compile time.

Error detection capabilities

If an error would now exchange the variable `yc` (that represents the encoded value y_c) with another encoded variable $u_c = A*u+B_u$, the result's computed signature $v_c \, mod \, A$ would be $(B_x + \mathbf{B_u} + B_z)$ instead of the expected $(B_x + \mathbf{B_y} + B_z)$.

If the addition were to be replaced erroneously by a subtraction, the resulting computed signature would be $(B_x - B_y + B_z)$ instead of $(B_x + B_y + B_z)$.

Thus, an ANB-code can detect the following errors: faulty operations, modified operands, exchanged operands, and exchanged operators. Because an ANB-code can detect exchanged operands and operators, we say it detects data and control flow errors.

However, now consider that there is a bitflip on the address bus when storing variable `yc`. Thus, we have a lost update on `yc` because `yc` is stored in a wrong memory location. When reading `yc` the next time, the old version of `yc` is read – which is correctly ANB-encoded but outdated. This example shows that ANB-codes might not detect lost updates.

Systems using this code

We know of no approach that uses solely ANB-encoding apart from our encoding compiler that we will present in Chapter 8. This encoding compiler can apply different codes – including ANB-codes – to C programs.

3.5. ANBD-Codes

To detect the use of outdated operands, i. e., lost updates, Forin introduced a timestamp D that counts variable updates [For89]. In the resulting *ANBD-code*, the encoded version of `x` is $x_c = A * x + B_x + D$. For checking the validity of code words, the expected signature *and* the expected D have to be known. In contrast to the signature, D is usually computed dynamically, that is, during the execution of an ANBD-encoded application. How D is actually computed and checked depends largely on how the ANBD-encoding is realized. Thus, we will explain it later when we are describing different ANBD-encoding solutions in the chapters 6, 7, and 8. However, all approaches have in common that the code checker also must have access to the expected D to facilitate checking the validity of code words.

Supported operations

ANBD-codes support the same operations as ANB-codes. However, the correctional actions required to ensure a reasonable signature for the result are more complicated because they have to consider D additionally to the signatures.

Implementation

The implementation of ANBD-codes is similar to ANB-codes. It can be done completely in software. However, hardware support for especially expensive operations in terms of runtime would be desirable to reduce the overall runtime costs.

Error detection capabilities

The ANBD-code finally can detect all errors defined in our symptom-based error model in Section 2.5, i. e., it detects faulty operations, modified operands, exchanged operands, exchanged operators, and lost updates.

Systems using this code

The Vital Coded Processor (VCP) by Forin [For89] to the best of our knowledge was the first system to apply an ANBD-code. VCP ANBD-encodes an application on source code level. As we pointed out in [WM08a], VCP requires knowledge of the complete data and control flow of the encoded program to precompute the signatures of all output variables for code checking. This prohibits the usage of dynamically allocated memory and dynamic control flow.

Furthermore, encoding loops and nested control flow structures at source code level is cumbersome and not described by Forin. He only presents the encoding of additions and if-statements and gives a general idea for encoding loops. However, he does not present encoding of other operations and nested control-flow structures. The level of automation of Forin's encoding also remains unclear, and he presents neither an evaluation of the error detection capabilities of VCP nor any runtime measurements.

Our Software Encoded Processing (SEP) (see Chapter 7) and Compiler Encoded Processing (CEP) (see Chapter 8) use also ANBD-encoding. We developed both solutions with the aim to remove the restrictions posed by VCP and make ANBD-encoding more generally usable.

3.6. Comparison of the Codes

Table 3.5 summarizes the results of our comparison of different arithmetic codes. Note that for AN-codes only the common form and none of the special codes are considered. The latter ones pose non-acceptable restrictions or are not yet well enough researched to be an option.

	Berger	**residue**	**AN**	**ANB**	**ANBD**
detectable errors					
faulty operation	✓	✓	✓	✓	✓
modified operand	✓	✓	✓	✓	✓
exchanged operand	◯	◯	×	✓	✓
exchanged operator	◯	◯	×	✓	✓
lost update	×	×	×	×	✓
supported operations					
addition	✓	✓	✓	✓	✓
subtraction	✓	✓	✓	✓	✓
multiplication	✓	✓	✓	✓	✓
division	✓	×	✓	×	×
shift left	✓	×	×	×	×
logical right shift	✓	×	×	×	×
arithmetic right shift	✓	×	×	×	×
logical or	✓	✓	✓	✓	✓
logical and	✓	✓	✓	✓	✓
logical not	✓	✓	✓	✓	✓
bitwise or	✓	×	×	×	×
bitwise and	✓	×	×	×	×
bitwise not	✓	×	×	×	×
comparisons	✓	×	×	✓	✓
data and control flow	×	×	×	✓	✓
signed arithmetic	✓	✓	✓	✓	✓
implementation					
hardware	✓	✓	✓	✓	✓
software	×	✓	✓	✓	✓

✓ possible
× not possible
◯ possible, but susceptible to errors

Table 3.5.: Summary of properties of the arithmetic presented in this chapter.

To summarize, we will not use Berger codes because they can only be implemented efficiently and safely in hardware. We will also not use residue codes because

they do not directly support an encoded division. However, a division is required for implementing right shifts that in turn are required to realize bitwise logical operations and unaligned memory accesses as we will explain in Chapter 4. Furthermore, both, Berger and residue codes, are not able to detect lost updates. Additionally, their support for detecting replaced operands and operators is solely based on the redundant choice of the operator/operand and its matching variant in the check bit computation. Thus, two matching errors that result in the replacement of operator/operand in the computation of the functional values and in the computation of the check bits will go undetected.

ANBD-codes are the only of the presented codes that support our complete error model. But they do not support a directly encoded division. We will instead use the AN-encoded division. For higher safety, an encodable software implementation of division is required.

The existing implementation of ANBD-codes – the Vital Coded Processor by Forin [For89] – is not sufficient for today's demands because it requires special hardware and is only applicable to a restricted set of programs that for example are not allowed to use dynamically allocated memory or dynamic control flow. Furthermore, VCP's presentation in [For89] lacks detail and evaluation. Apart from an encoded addition and an encoded branch statement no other encoded operations are described. Forin does not discuss implementation of other operations or issues caused by the usually restricted size of functional and encoded values as well.

Hence, the objective of this thesis is to present solutions that make ANBD-codes usable in today's systems. This includes development of a set of encoded and encodable operations including support for dynamically allocated memory (see Chapter 4) and the application of these operations to programs (see chapters 7 and 8).

When implementing ANBD-codes, implementations for ANB- and AN-codes are generated as a by-product without much additional effort. This enables us to compare the error detection capabilities and overheads of these three codes.

4. Encoding an Instruction Set

Implementing an AN-, an ANB- or an ANBD-code in software, requires to adapt the programs that shall be enabled to detect errors disturbing their execution. These programs have to process encoded data items instead of unencoded ones, and the processing has to preserve the code in the error-free case. Thus, we have to modify the data types used to store data and the operations executed to process data. This process we name *encoding* the program.

This chapter presents the encoded versions of basic building blocks of applications. We published parts of this work in [WF07a, WF07b, SSF09]. To the best of our knowledge, we are the first who describe encoded operations in such detail. All encoding approaches can reuse the encoded building blocks presented in this chapter. Especially our own approaches presented in Chapters 7 and 8 do so.

For encoding an application, the following is required and, thus, presented in this chapter:

Implementation of encoding and decoding:
We need to encode inputs to our programs if they are not yet encoded. Inputs can be constants stored in the program itself or inputs given at runtime by the user. Furthermore, we must decode outputs that are sent to other entities that are not encoded. The code of these outputs has to be checked to prevent erroneous output. In Chapter 3, we already described the mathematical formulas describing these three operations (encoding, decoding, and code checking). In the following, we will describe the problems we encountered in mapping these formulas to computer-implemented arithmetic.

Encoded operations:
We need encoded versions of all operations used by the application that we encode. The encoded operations are comprised, for example, of arithmetic and logical operations. In contrast to their unencoded (native) counterparts, encoded operations process encoded data and ensure that in an error-free execution correctly encoded results are produced.

Encoded constants:
We have to encode all constants and initialization values used.

Encoded calls to external libraries:
We have to handle calls to external libraries.

Encoded data and control flow:
For ANB- and ANBD-codes we, furthermore, must encode data and control flow, that is, we have to check that instructions are executed in the correct order with the right operands and that all conditional jumps are executed correctly.

Encoded dynamically accessed memory:

If a program accesses memory, for example, in LLVM bitcode using load and store instructions, we do not know the access pattern previously, that is, we do not know which addresses are accessed when at runtime.

An example is an access to an array for which the accessed element depends on user input. Each element of the array should have a different signature to facilitate the detection of a faulty read or write to the array. However, if we want to check if, for example, a read operations accessed the correct value, we have to know the signature expected for this value. This signature we cannot determine statically before running the program because the user will determine at runtime which element is read and, thus, which signature the value is expected to have. For such cases, the expected signature used for code checking is determined dynamically at runtime.

Another example is dynamically allocated memory. Here, we cannot even assign signatures statically at compile time because we might only know at runtime how much memory is allocated. Thus, the signatures are even assigned dynamically.

Thus, we introduce the concept of *dynamic signatures* for encoding dynamically allocated or accessed memory in this chapter.

Versioning:

For ANBD-codes we have to implement versioning of repeatedly written data items additionally. The version D used in ANBD-codes facilitates the detection of lost updates. However, for checking the code of an ANBD-encoded data item, we have to be able to determine the version expected for this data item. We will propose several approaches to solve this problem.

This selection of building blocks is based on our experiences made during encoding the DLX (see Chapter 7) and the LLVM (see Chapter 8) instruction sets. DLX is an academic RISC[1] instruction set developed by Hennessy and Patterson [PH90]. The LLVM compiler framework [LA04] defines the LLVM bitcode that is a static single assignment assembler-like language. DLX and LLVM as well have the advantage that, in comparison to any native assembler, both have a manageable amount of operations for which we have to provide encoded versions. However, similar concepts are used in other programming and assembler languages. Hence, the concepts presented in this chapter can be applied to other instruction sets as well.

4.1. Implementation of Encoding and Decoding

This section describes the implementation of operations for encoding and decoding data. These operations are necessary for encoding input received by a program and constants used in the program and for decoding output produced by the program. Note that this section does not describe encoded operations

[1] RISC stands for reduced instruction set computer. It describes a CPU design strategy. RISC is based on the following assumption: A simpler instruction set can result in faster program execution if because of the simplification the single instructions become faster.

such as an encoded addition. These will be described in Section 4.2.

Note further that for understanding this section, knowledge of the representation of numbers within computing systems is required. The reader should be familiar with the two's complement and should know how type-casts from signed to unsigned types and vice versa work.

In the following we will assume that functional, i. e., unencoded, values have a size of n bits, while encoded values require m bits. For n and m the following relations hold $m > n$ and $n > 0$. Thus, $m > 0$ also holds. The pseudocode examples presented here and later in Section 4.2 assume $n = 32$ and $m = 64$. They use the appropriate signed and unsigned integer types `uint32_t`, `int32_t`, `uint64_t`, and `int64_t` as defined in the `stdint.h` header file of the C standard library.

Note that despite presenting only implementations for $n = 32$, we provide all encoding/decoding operations and encoded operations for integer types of the widths 8, 16, and 32. These implementations are very similar: Only constants required for some corrections depend on the width of the data type of the functional value. To ease implementation, the encoded versions of the different functional value types are all stored using a 64-bit integer type.

4.1.1. Provided Functions

Listing 4.1 contains the declarations of the functions `encode` and `decode` for an ANB-code. These functions we need

- for encoding input and
- for determining the functional value of a code word and at the same time checking if this code word is a valid code word, that is, contains the signature expected.

The interfaces for AN- and ANBD-encoding/decoding are defined quite similar. For AN-encoding/decoding, no signature is used, and ANBD-encoding/decoding additionally requires and applies the version information D.

```
1  /* Encode the given value using B as signature and the code parameter A.
2   * @param v functional value
3   * @param B signature for encoding
4   * @param A code parameter A
5   * @return ANB-encoded version of v: vc=A*v+B
6   */
7  uint64_t encode(uint32_t v, uint32_t B, uint32_t A);
8
9  /* Decode vc and check vc for validity.
10  * If vc is no valid code word, the application is aborted.
11  * @param vc ANB-encoded value
12  * @param expectedB the signature vc is expected to have
13  * @param A code parameter A
14  * @return functional value v of vc
15  */
16 uint32_t decode(uint64_t vc, uint32_t expectedB, uint32_t A);
```

Listing 4.1: Declaration of encoding and decoding functions for an ANB-code.

Next, we will describe two different possibilities to implement the interface described in Listing 4.1.

4.1.2. Encoding

Two different possibilities exist to implement the actual encoding of data: *signed* and *unsigned encoding*. There are encoded operations that require one kind of encoding and cannot be implemented using the other. For example, the signed division requires signed encoded values, while the unsigned division requires unsigned encoded values. Furthermore, the implementation of the encoded base operations depends on the chosen kind of encoding. For these reasons, we introduce these encoding kinds before describing the actual encoded operations.

When implementing the encoding of a number a, that is, the computation of $a_c = A * a + B_a$, we have to decide if the operations used are executed signed or unsigned. The first we call *signed encoding*, the second *unsigned encoding*. In C, the type of the processed variables determines if an operation is executed signed or unsigned. For addition and multiplication operations, signedness does not matter. They are the same for both, signed and unsigned integers.

However, before the actual encoding is executed, we have to cast the encoded variable a to the larger domain of encoded numbers, i. e., from n bits to m bits. This cast is either done with a sign extension or without. If we cast to an unsigned type, no sign extension is made, that is, independent of the content of the variable before casting, the newly introduced higher-order bits contain zeros. If we cast from a smaller signed type to a larger signed type, a sign extension is made, that is, the added higher-order bits equal the most significant bit of the casted value. For example a signed cast of 1010 from four to eight bits would result in 11111010, while an unsigned cast would result in 00001010.

Signed encoding

Listing 4.2 contains the signed encoding function's implementation, and Figure 4.1 depicts the encoding transformation applied to numbers which are encoded with sign extension, i. e., using a signed cast operation. We explicitly marked where positive and negative numbers are located under the condition that the corresponding bit patterns are interpreted as signed numbers using the two's complement. The values of the unsigned versions of the depicted values increases from left to right – starting from 0 and going up to $2^n - 1$ and $2^m - 1$ respectively. Note that due to space reasons the representation does not depict the correct size relations.

```
1 uint64_t encode(uint32_t v, uint32_t B, uint32_t A){
2     int64_t r_an = ((int64_t)(int32_t) A * (int64_t)(int32_t) v;
3     int64_t r_anb = r_an + (int64_t)(int32_t)B);
4     return (uint64_t) result;
5 }
```

Listing 4.2: Signed ANB-encoding.

In the first row of Figure 4.1 the unencoded functional numbers are depicted. The next row represents these numbers after type casting them to the type of the

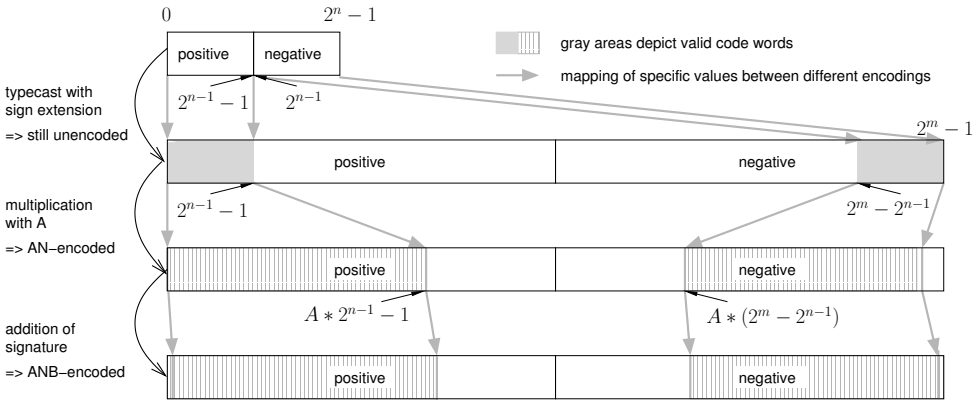


Figure 4.1.: Encoding transformations for an AN- and an ANB-code using signed operations.

encoded numbers that requires more bits for its representation. In Listing 4.2, this cast is implemented by `(int64_t)(int32_t) v`. We first cast from the unsigned 32-bit integer to the signed 32-bit integer type, and then from the smaller signed to the larger signed type. By the first step, we ensure that the sign extension is made in the next casting step. The resulting sign extension maps values in the upper half of the functional values ($[2^{n-1}, 2^n - 1]$) to the upper half of the new domain ($[2^{m-1}, 2^m - 1]$).

The next row depicts the multiplication with A, which just spreads the numbers across the available space. Finally, the fourth row depicts the addition of the signature. This increases the value by at most $A - 1$.

As can be seen in Figure 4.1, signed encoding results in the following code property: An encoded number has the same sign as its unencoded version. Thus, negative and positive functional values still have different signs after being encoded. This property is required for the implementation of the encoded signed division (see Section 4.2.1).

Unsigned Encoding

However , the implementation of the encoded unsigned division requires unsigned encoded values. The reason is that the interval of signed encoded numbers is not continuous, but split up in the middle as can be seen in Figure 4.1. However, the unsigned division requires a continuous interval from the smallest to the largest value.

For the transformations executed during unsigned encoding and their results look at Figure 4.2. Listing 4.3 contains the implementation of the unsigned encoding.

Unsigned encoding uses an unsigned cast without sign extension. Hence, it does not split up the domain of code words in the middle. Thus, unsigned encoded numbers are the precondition for the unsigned division operation.

Note that many encoded operations that we will introduce in Section 4.2 slightly differ for differently encoded values. For example, the implementation of an

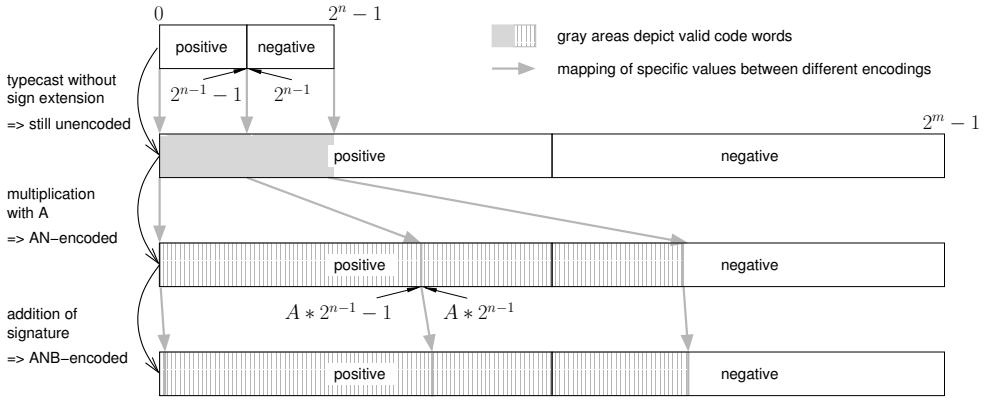


Figure 4.2.: Encoding transformations for an AN- and an ANB-code using unsigned operations.

```
1 uint64_t encode(uint32_t v, uint32_t B, uint32_t A){
2     return (uint64_t)A * (uint64_t)v + (uint64_t)B ;
3 }
```

Listing 4.3: Unsigned ANB-encoding.

encoded addition of signed encoded values is slightly different from the implementation for unsigned encoded values. However, due to space reasons we will introduce only the versions for unsigned encoded values.

For Software Encoded Processing that we describe in Chapter 7, we implemented both encodings and the appropriate encoded operations. We noticed no measurable differences in the generated slowdown. Thus, for Compiler Encoded Processing that we describe in Chapter 8, we implemented so far only the unsigned encoding.

4.1.3. Conversion: Signed Encoded ⇆ Unsigned Encoded

For some operations of computer implemented arithmetic signedness matters, that is, different versions exist for the unsigned and the signed interpretation of numbers. This is the case for divisions and comparisons. For additions, subtractions, and multiplication signedness does not matter, that is, for signed and unsigned numbers the same implementation of that operation is used by the processor to execute this operation.

The information if a signed or unsigned division or comparison is required, is either contained in the variables' types or directly encoded in the operation. For example, in C the signedness is defined by the variable types. Whereas, in LLVM signedness information is encoded into the called operation, for example, either `div` or `udiv` is called for the signed and unsigned division respectively.

For the operations for which signedness matters, we also have to provide an encoded version for the signed and the unsigned version. As we will demonstrate

later when presenting the encoded division operation, we need signed encoded values for implementing the encoded signed division and unsigned encoded values for implementing the encoded unsigned division. Thus, we need to be able to translate signed into unsigned encoded numbers and vice versa.

signed → unsigned

To transform a signed encoded into an unsigned encoded value, we have to check if the encoded value represents a negative number. If the code word is negative, we have to subtract the AN-encoded version of the sign bits (denoted by $signBits_c$). These encoded sign bits $signBits_c$ were added when first casting the functional value to the larger signed data type with sign extension and afterwards multiplying this value with A. If the code word is positive, nothing has to be done because signed and unsigned encoding of positive functional values result in the same code words because for neither case a sign extension is made.

Signed encoded values have the same sign as the functional value represented by them if both are interpreted using the two's complement. Thus, for determining if $signBits_c$ has to be subtracted, we can just check if the encoded value is smaller than zero using a signed comparison operation.

Encoded sign bits

The AN-encoded sign bits only depend on A and the data type sizes n and m. Otherwise, they are constant: $signBits_c = (2^m - 2^n) * A \mod 2^m$. The value $2^m - 2^n$ represents the unencoded sign bits, i. e., the upper $m - n$ bits of the value are set and the remaining bits are zero. These are the bits added to a negative functional value that is casted to a larger type with sign extension. Multiplying this value with A AN-encodes it, which is required because we want to subtract it directly from the encoded value if required.

unsigned → signed

When transforming an unsigned encoded value into a signed encoded value, we also have to check if the encoded value represents a functional value that is negative (if it is interpreted in two's complement). Since unsigned encoded numbers do not have the same sign as their represented functional values, this requires a comparison with the encoded version of the first negative functional value. In two's complement, the first number that is interpreted as a negative has the unsigned value 2^{n-1}. It represents -2^{n-1}. Its encoded version is $firstNegative_c = A * 2^{n-1}$. If the unsigned encoded value (without signature B) is larger than $firstNegative_c$, we have to add the encoded sign bits $signBits_c$ to transform it into its signed encoded representation. If the unsigned encoded value is smaller than $firstNegative_c$, nothing needs to be done because a positive functional value is represented.

Encoding of these transformations

These two transformations, *signed* → *unsigned* and *unsigned* → *signed*, have to be completely encoded. Thus, for an ANB-code we add a signature to each of the two constants $signBits_c$ and $firstNegative_c$, which are already AN-encoded. Furthermore, the mentioned comparisons are executed using the encoded comparison operations that we introduce later, and the required if-statement is encoded using the mechanism introduced in the next section.

We implemented the encoded transformation from unsigned to signed encoded values in the functions `unsignedToSigned_<code>` where `<code>` depending on the encoding equals `an`, `anb`, or `anbd`. The functions `signedToUnsigned_<code>`

implement the encoded transformation from signed to unsigned encoded values. We use these functions for implementing the encoded signed division.

4.1.4. Decoding

While encoding is applied statically to constants and dynamically to input data, decoding and code checking is used to produce unencoded outputs.

The decoding and code checking operations for signed and unsigned encoded values are different. The reason is that the modulo operations used to determine the signature contained and the division used to obtain the functional value depend on the signedness. Thus, for signed encoded values, signed division and modulo operations have to be used in decoding and code checking. For unsigned encoded values, unsigned division and modulo operations have to be used in decoding and code checking.

4.2. Encoded Operations

As explained before, for encoding a program, we have to replace all data values with their encoded versions. Thus, we have to provide *encoded versions* of all operations used in a program. These encoded versions of operations take encoded operands and produce valid encoded results without decoding the operands for the computation. If the latter restriction is violated, the error detection capabilities of the code are reduced. The reason is that with decoding a window of vulnerability is opened within which data may be modified undetectably.

In the following, we present the encoded versions of arithmetic and logical operations. We call these the *encoded operations*. The sections following this one, present concepts for encoding control and data flow.

Encoded base operations

We identified two different ways to realize encoded operations. For arithmetic, boolean logical, and comparison operations we use operations that we encoded by hand – the *encoded base operations*. We published these operations in [WF07a] and in this thesis we describe them in Section 4.2.1.

Encodable replacement operations

To reduce the error-prone manual work for encoding operations, we only encode a small number of operations by hand. For all remaining operations such as type casting, shifts, bitwise logical operations, or unaligned memory accesses, we developed our own encodable C-implementations – the *replacement operations*. These replacement operations we published in [SSF09] and in this thesis we describe them in Section 4.2.2.

The replacement operations are implemented in C and use only operations for which hand-encoded versions exist. Thus, the replacement operations can be automatically encoded using the hand encoded base operations. Therefore, all occurrences of operations for which no hand-encoded version exists must be replaced with their encodable replacement version before the actual encoding, that is, the replacement of variables and operations with their encoded versions,

is done. This replacement can be automated as well. Afterwards, the program together with the replacement operations can be encoded automatically.

Floating point operations

Our tools presented in Chapter 7 and 8 can only encode applications that solely process integers. Thus, floating point variables and operations have to be replaced with an encodable software implementation. Currently, that has to be done by hand.

4.2.1. Encoded Base Operations

This section describes the hand-encoded base operations that we provide for the following operations:

- addition,
- subtraction,
- multiplication,
- signed and unsigned division,
- signed and unsigned comparisons, and
- the boolean logical operations and, or, and xor.

Before describing the hand-encoded base operations, we introduce the basic requirements that encoded operations must fulfill and which we, thus, have to consider during hand-encoding the base operations. Furthermore, for encoding some of the base operations we need to be able to encode if-statements. Hence, we will introduce these also before finally describing the specification of the encoded base operations and their implementation.

Note further that our descriptions in the following consider only the ANB-encoded versions. The AN-encoded version can be obtained by removing instructions that correct signatures. For the ANBD-encoded variant, the version D has to be considered additionally which is handled similar to the signatures. We implemented all three encodings. However, we use only the AN- and ANB-encoded operations in our encoding approaches.

Requirements Posed on Encoded Base Operations

The following requirements that we pose on encoded operations ensure that the code is implemented in a safe way, i. e., without reducing the error detection capabilities of the code implemented.

No decoding

First of all, it has to be ensured that during an encoded operation the processed data is not decoded completely or partially because this would open a window of vulnerability where undetectable errors could happen. Complete decoding happens when a code word is divided by A. If the signature is removed without adequate substitution, this is a partial decoding. In this case the encoding is reduced to an AN-code that, for example, cannot detect the exchange of operands.

Atomicity

The term atomicity originates from the database domain. If a transaction is atomic, it is either executed completely or not at all. In the context of encoding,

we use atomicity in the sense that an encoded base operation is either executed completely or that we are at least able to notice if parts of an encoded operation were not executed. Thus, we require that an incomplete execution of an encoded operation produces results that are invalid code words. Thereby, we ensure the detection of control flow errors that disturb the execution of an encoded operation.

Unique signature

Each encoded operation has to result in a unique signature, that is, no two encoded operations produce the same output signature when presented with the same input signatures. This ensures detection of operator errors. Otherwise, two operators that produce the same signature for the same input signatures could be exchanged unnoticeably.

Encoding of If-statements

When hand-encoding arithmetic and comparison operations, we need to encode if-statements, for example, to ensure overflow behavior that conforms to the ANSI C-standard. Thus, we explain the encoding of an if-statement with the help of an example whose unencoded version is:

```
1 if( x >= 0 ){
2    y = z + x;
3 }else{
4    y = x - y;
5 }
```

We have to encode this if-statement in a way that facilitates detection of errors

- in the computations executed in the if- and the else-branch,
- in the computation of the condition (`x>=0`), and
- in the branch itself, i. e., if the taken branch does not match the result of the condition evaluation.

Forin in [For89] presented the following encoded version of the if-statement that fulfills these requirements. The comments show the variable contents for the error-free case:

```
1  sigCond = sigGEZ(xc);    // if(x<0) sigCond = sigNeg
2                           // else    sigCond = sigPos
3  if( xc >= 0 ){
4     yc = zc+xc;         // yc=A*(z+x)+Bz+Bx
5  }else{
6     yc = xc-yc;         // yc=A*(x-y)+Bx-By
7     yc += (Bz+Bx);      // yc=A*(x-y)+Bx-By+Bz+Bx
8     yc += -(Bx-By);     // yc=A*(x-y)+Bz+Bx
9     yc += -sigNeg+sigPos;
10                        // yc=A*(x-y)+Bz+Bx-sigNeg+sigPos
11 }
12 yc += sigCond; // By=Bz+Bx+sigPos
```

`sigGEZ(xc)` computes the signature for the greater-equal-comparison of x with zero. It evaluates to two different values: one value if the functional value x represented by x_c is greater than or equal to zero (`sigPos`) and a different value if it is less than zero (`sigNeg`). We explain its implementation later.

We define that after executing the if-statement the signature of y_c (the variable modified by the if-statement) has to be $B_z + B_x + sigPos$, independently of the chosen branch. The signature of y_c could be defined differently. Important is that after the execution of the if-statement y_c can only have that signature if everything was correctly executed and that any of the described errors would lead to y_c having a different signature.

With our intended signature for variables modified by the if-statement, computations of any variable in the else-branch must be completed with:

- the addition of the signature that is generated for this variable in the if-branch,
- the subtraction of the signature that is generated for this variable in the else-branch, and
- the subtraction of the expected value for `sigCond` for the else-branch (`sigNeg`) and addition of the one we defined the result to have (`sigPos`).

Note that in the Vital Coded Processor (see Chapter 6) and Compiler Encoded Processing (see Chapter 8) all the values applied to y_c in the lines 7 to 9 are constants that are known at encoding time. Thus, they are precomputed and applied with one addition. This is important because it prevents partial decoding that might happen otherwise if signatures are removed for example in line 8. Furthermore, this ensures that either all of these correction or none are applied. However, if all these corrections are missing, the result produced (y_c in our example) will not be a valid code word.

While computations such as addition, subtraction, and also condition computation are protected as usual by a predetermined signature, the branching behavior is explicitly checked by assuming a specific value for `sigCond` in the different branches. The addition of the actual `sigCond` in line 12 implements the check if the correct branch was taken and ensures that the signature of the condition is part of y_c's signature. The latter checks the correctness of the computation of the condition.

If any of the computations (in the branches or of the condition) or any of the used operands was erroneous, the signature of y_c will be destroyed because y_c's signature depends on any of these computations. If a branch is chosen which does not match x's size, that is, the expected comparison result, this would also result in a wrong signature of y_c, because the addition of `sigCond` in line 12 would not match the value expected for `sigCond` in the branches.

Forin [For89] presented the following approach for realizing `sigGEZ`: Signed numbers are represented using the two's complement as is done in most systems. When a negative value neg and a positive value pos are encoded using signed operations and the same signature B, the two encoded values $neg_c = A * neg + B$ and $pos_c = A * pos + B$ will not have the same signature if they are interpreted as unsigned values: $(pos_c \mod A) = B \neq (neg_c \mod A) = ((2^m + B) \mod A)$. Remember m is the width of the binary representation of encoded values. Thus, `sigGEZ` is defined as follows for $x_c = A * x + B_x$:

$$sigGEZ(x_c) = \begin{cases} B_x & if \quad x \geq 0 \\ (2^m + B_x) \mod A & if \quad x < 0 \end{cases}$$

For more information about encoded comparison operations see page 66.

Specification of Encoded Arithmetic Operations

Before finally discussing our hand-encoded base operations, we have to determine their intended semantics. Arithmetic operations implemented by processors are quite different from mathematical arithmetic using infinite integers. In processors the size of integers is restricted by the chosen data type. If the data type is n bits wide, the computations implemented by a processor take place in the congruence classes modulo 2^n: $\mathbb{Z}/2^n\mathbb{Z}$. If, for example, the addition of two values a and b would result in a value greater than $2^n - 1$, the addition's result is $(a + b) \mod 2^n$, that is, the addition of a and b is overflown, i. e., wrapped around.

Also higher level programming languages implement this wrapping around. For example, the ANSI C99 standard [ISO99] that defines the programming language C requires this wrapping around in the case of an over- or underflow for unsigned integer types. Note that signed operations require correct modulo arithmetic anyway, that is, the processor operations must wrap around correctly for over- and underflowing computations to implement signed arithmetic. For example, the addition of a negative and a positive number, if interpreted as unsigned values, is nothing else than an overflow that wrapped around correctly. Remember that the implementations of addition, subtraction, and multiplication are equal for signed and unsigned values. They differ only in which flags the processor sets to indicate which over- or underflows happened.

The encoding solutions presented by Forin in [For89] do not consider over- or underflows. Instead the programmer has to ensure that these will not occur. If an overflow happens, it might either lead to invalid code words or to valid code words that contain wrong functional values. The outcome depends on the choice of A. How arithmetic with signed values that requires correct overflows can be encoded is not discussed at all by Forin.

However, we want to execute arbitrary programs without restricting the programmer. Thus, we have to ensure that all operations work as the programmer expects them to work. Hence, it is required that the functional values represented by the encoded values are overflowing and underflowing as they would do without encoding. Table 4.1 summarizes the conditions that we have to fulfill for the different arithmetic operations that are subject to over- or underflows.

In the following section, we will show that computations with encoded values require additional corrective actions to be taken to ensure the correct overflow behavior in the domain of the functional values. The reason is that the algebraic structures of unencoded and encoded numbers are not isomorphic. If the two structures of encoded and unencoded numbers shall be isomorphic, A would have to be equal to 2^{m-n}. However, choosing A to be a power of two would result in

operation	requires		justification
	OF	**UF**	
add (signed)	×		addition of numbers with different signs
add (unsigned)	×		according to ANSI C99 standard
sub (signed)		×	subtraction of numbers with different signs
sub (unsigned)		×	according to ANSI C99 standard
mult (signed)	×		multiplication of numbers with different signs
mult (unsigned)	×		according to ANSI C99 standard

Table 4.1.: Requirements on operations with respect to correct realization of overflows (OF) and underflows (UF).

a minimal Hamming distance of only one between valid code words – rendering the code useless against a wide variety of bitflips. Since encoded and functional values are non-isomorphic, encoded numbers normally will not overflow where the functional values do. This means that we have to check for each operation if there would have been an underflow or overflow within the functional values. If so, we have to correct the obtained encoded result accordingly.

Signature Precomputation and Correction

For all the encoded operations, restrictions with respect to the signatures of the input parameters have to be fulfilled. For example, for a division $\frac{x}{y}$, the signature B_y should be unequal to zero because otherwise the result's signature $\frac{B_x}{B_y}$ would be undefined because a division by zero would occur in that case. Furthermore, we ensure that the signature of the result is greater than zero and smaller than A.

Signature correction functions

Together with our encoded base operations, we provide a *signature correction function* for each encoded operation. These facilitate choosing the signatures of the input parameters appropriately at encoding time or adapting the signatures of encoded values at runtime. Of course, the first method is the preferred one because it induces no runtime costs. Signature correction functions take a randomly chosen signature for each parameter of an operation as input, and provide an adaptation value for each signature. This adaptation value is to be added to the original signature. Thereby, this signature is changed to an appropriate signature for the operation. The described adaptation can also be applied to encoded values at runtime to change the signature of these values.

The correction functions are implemented in a way that they try to ensure that the randomness of the given signatures is reduced as few as possible. However, it cannot be completely prevented that the restrictions imposed lead to some signatures being more probable than others. For example, for the naive encoded addition where the result's signature is the sum of the signatures of the parameters, the input signatures chosen tend to be values smaller than $\frac{A}{2}$. Future implementations of the encoded operations could avoid this issue by correcting signatures for the results instead for the parameters.

Signature precomputation functions

Furthermore, we provide a *signature precomputation function* for each encoded operation. Given the signatures of the input values, this function computes the expected signature of the return value. Signature correction functions are used, for example, by the encoding compiler presented in Chapter 8 for determining the signatures of intermediate results at encoding time.

In the following, for the sake of simplicity we do not present signature correction and signature precomputation functions. All pseudocode examples presented contain comments at the return statement containing the formula for computing the signature expected for the result produced. The signature precomputation function of an operation just implements this formula. The signatures of the input values are chosen in a way that ensures that the result's signature is greater than zero and smaller than A.

Addition

Overflow

Figure 4.3 demonstrates the addition of two unsigned encoded values: $x_c = A * x + B_x$ and $y_c = A * y + B_y$. In the chosen example, the addition of the two functional values x and y overflows in the domain of the functional values. Thus, the result of the addition of the functional values is $x + y - 2^n$ because an addition (modulo the domain size 2^n) overflows at most once. The expected result of the addition of the encoded values, thus, should be $sum_{exp} = A * (x + y - 2^n) + B_x + B_y$. Instead the result of the addition of the encoded values x_c and y_c is $sum = A * x + B_x + A * y + B_y = A * (x + y) + B_x + B_y$. The reason is that the domain of encoded values is large enough that the sum of the encoded values does not overflow. That is due to the choice of A and the data type used.

Even if we choose A in a way that ensures that the domain of the encoded values overflows when the functional values do[2], the result would be unequal to sum_{exp}. The reason is that the overflow of the encoded values corresponds to modulo 2^m on the the encoded value instead of modulo 2^n on the functional value. These two different overflows would be equivalent for $A = 2^{m-n}$. However, as explained before A should not be a power of two due to safety reasons. Thus, both variants of an overflow in an addition of two functional values – with and without overflow in the encoded values – require additional (but different) corrective actions.

Note that sum_{exp} and sum are both multiples of A and carry the expected signature $B_x + B_y$, that is, both are valid code words. We call sum an *incorrect valid code* word because it contains a valid code word with an incorrect functional value. To obtain the correct code word a correction by `OVERFLOW_CORRECTION` $= sum - sum_{exp} = A * 2^n$ is required.

[2]Appendix A details the requirements for choosing an A that ensures that overflows happen for unencoded functional values and for encoded values at the same places. In that case, if an operation overflows in its execution with functional values, it will also overflow in its execution with encoded values.

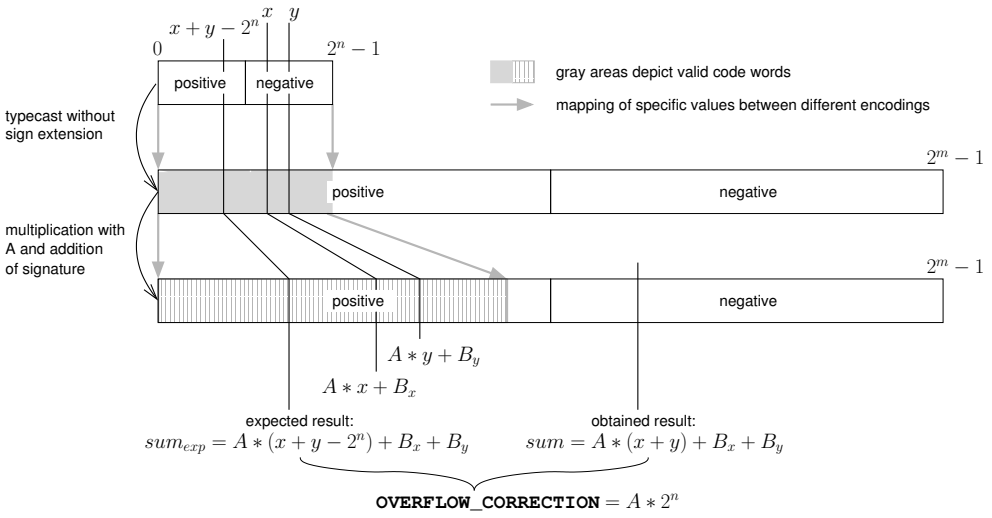


Figure 4.3.: Demonstration of the effects of encoding on the overflow behavior of an overflowing addition $x + y$ using unsigned encoded values.

The incorrect valid code word *sum* contains an incorrect functional value in the m-bit domain of the encoded values. If we divide *sum* by A and do not cast its type back to the n-bit-sized type of the functional values, we obtain $x + y$ instead of $x + y - 2^n$. The latter one can be obtained by casting to the n-bit-sized type of the functional values which is nothing else than a modulo computation with 2^n. Hence, incorrect valid code words can be decoded and their code can be checked successfully.

Why correct incorrect valid code words?

Thus, a legitimate question is, why we should correct these incorrect valid code words? Using such results that should have been overflown is no problem in additions, subtractions, and multiplications. The reason is that these operations and the modulo-2^n operation (that is, the cast to the n-bit-sized type) during decoding are commutative. However, this is not the case for divisions and comparisons. These are not commutative with the modulo operation. This will lead to different results for these operations for corrected words, e. g., sum_{exp}, and uncorrected results, e. g., *sum*. Furthermore, leaving the values of additions, subtractions, and multiplications uncorrected, could lead to unintended overflows within the encoded values.

Of course, we could just accept the greater domain of encodable numbers and do no corrections, that is, we would not restrict the functional values to values from 0 to $2^n - 1$. However, realization of arithmetic with signed numbers in the two's complement requires overflows to wrap around correctly.

Sign overflow

The described overflow problem occurs also for signed encoded values. However, the problem does not happen if an addition wraps around 2^n, but if an addition crosses 2^{n-1}, that is, if a so called sign overflow happens. A sign overflow means that one adds two signed positive numbers and the result is too large and flows over into the negative values. Figure 4.4 demonstrates this. Note

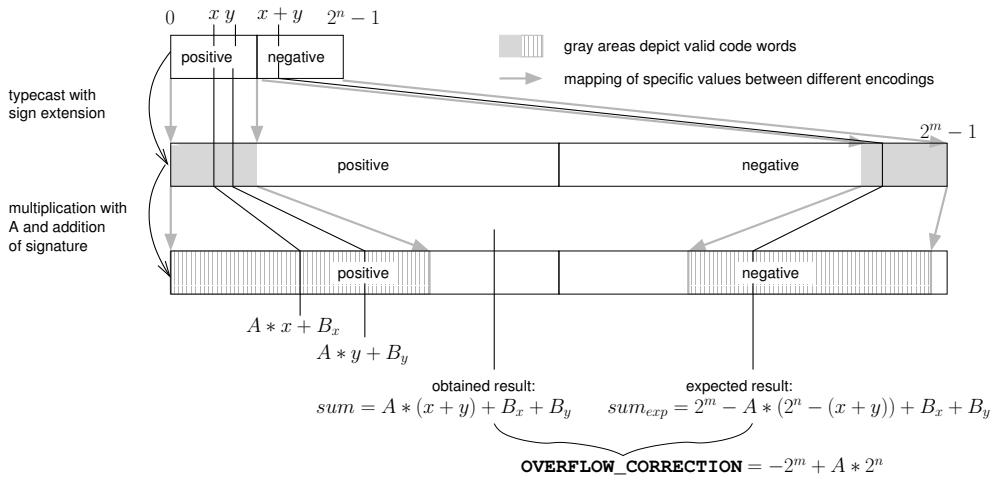


Figure 4.4.: Demonstration of the effects of encoding on the overflow behavior of an overflowing addition $x + y$ using signed encoded values.

that the expected encoded value of $x + y$ in the depicted example is $sum_{exp} = 2^m - A * (2^n - (x + y)) + B_x + By$ because of the sign extension made during encoding. Thus, the required correctional value is `OVERFLOW_CORRECTION` $= sum - sum_{exp} = -2^m + A * 2^n$. This reduces to $A * 2^n$ because the overflow correction is applied modulo 2^m.

Implementation

The pseudocode Listing 4.4 shows our implementation of an encoded addition using unsigned encoded values. Note that we denote x^y as `x^y` in all our pseudocode listings. We do not use `x^y` for indicating the `xor` operation that it signifies in C.

First, the encoded addition adds the unsigned encoded values (line 7) and afterwards it does an encoded overflow correction if required (lines 10 to 24). The overflow correction in Listing 4.4 uses an encoded if-statement to ensure that control flow errors influencing the correction are detectable. Thus, it does not impair the safety that we remove the signatures from *result* in line 15 because this comparison is checked by the encoding of the if-statement that either applies the overflow correction or not.

Note that in all our pseudocode listings we assume that A is a constant global value that requires at most 31 bits. This ensures that `OVERFLOW_CORRECTION` does not require more than 64 bits for storage.

The implementation of the encoded addition using signed encoded values looks similar. However, its check if an overflow happened is more complicated because it has to be checked if a sign overflow of two positive (in two's complement) or two negative functional values should have happened.

Note that in our Compiler Encoded Processing that we will present in Chapter 8 `sigPos` and `sigNeg` are constants at runtime. Their values only depend on the signatures of the two encoded input operands. These signatures in Compiler

```
1  const uint32_t B_OC = ...; // signature of overflow correction
2  const uint64_t OVERFLOW_CORRECTION = A*2^32 + B_OC;
3  const uint64_t SMALLEST_INTEGER_NOT_POSSIBLE_ENC = A*2^32;
4
5  uint64_t add_anb(uint64_t xc, uint32_t Bx, uint64_t yc, uint32_t By){
6     // the actual addition
7     uint64_t result = xc + yc;
8
9     // expected signatures of if-condition check
10    uint32_t sigPos = Bx + By;
11    uint32_t sigNeg = (2^64 % A + sigPos) % A;
12
13    // application of the overflow correction
14    // if an overflow happened
15    if(result - Bx - By >= SMALLEST_INTEGER_NOT_POSSIBLE_ENC){
16       result = result - OVERFLOW_CORRECTION;
17    }else{
18       result = result - B_OC - sigNeg + sigPos;
19    }
20
21    // encoded check of the previous if-statement's control flow
22    uint64_t diff = result - SMALLEST_INTEGER_NOT_POSSIBLE_ENC;
23    uint32_t sigCond = diff % A;
24    result += sigCond;
25
26    return result;     // = A*((x+y) % 2^n) + Bx + By + sigPos - B_OC
27                       // = A*((x+y) % 2^n) + 2*(Bx + By) - B_OC
28 }
```

Listing 4.4: Encoded addition using unsigned encoded values and doing an encoded overflow correction.

Encoded Processing are assigned at encoding, i. e., compile, time. In Software Encoded Processing (see Chapter 7) these signatures are only known at runtime. Thus, for Compiler Encoded Processing, the value $-B\_OC-sigNeg+sigPos$ applied in line 18 is constant that is applied with one addition, while it is not for Software Encoded Processing. For Compiler Encoded Processing, this ensures that the corrections are either applied completely or not at all. If they are not applied, an invalid code word is produced as a result and, thereby, the loss of the addition of the correction becomes detectable.

To come back to our requirements for encoded operations that we defined in Section 4.2.1, note that none of the encoded values used in `add_anb` is completely or partially decoded.

Furthermore, in Compiler Encoded Processing, if any of the instructions forming `add_anb` is not executed, but should have been executed, the produced result will be an invalid code word. For Software Encoded Processing, we can not guarantee this because $-B\_OC-sigNeg+sigPos$ is computed at runtime and parts of its application might be left out. This results in some undetectable error scenarios. However, these always require several matching errors such as

- control flow erroneously goes to the else branch where no overflow correction is done and
- the addition of `sigPos` and the removal of `sigNeg` are lost.

Because of the encoded overflow correction, the signature of the result is $2 *$

$(B_x + B_y) - B_{OC}$. No other encoded base operation will produce this signature for the same input signatures B_x and B_y.

Note that the sign overflow, that is, an addition that crosses 2^{n-1} in the functional values, is possible without correction if unsigned encoded values are added. On the other hand an overflow, that is, an addition that wraps around 2^n in the functional values, is possible without correction if signed encoded values are added. Thus, if with a dataflow analysis or by using additional information provided through the programmer the signedness of the added operands can be determined, overflow corrections could be partly removed. However, this might also lead to additional conversions between signed and unsigned encoded values. Furthermore, currently, neither the DLX nor the LLVM instruction set provide us the required information. But in the future, with additional data flow analysis, this can be used for further optimizations. Future work

Subtraction

For the subtraction similar issues as for the addition exist. For unsigned encoded values, underflowing subtractions, that is, subtractions $x - y$ with $y > x$, do not result in the expect result. The reason as with the addition is that the functional values and the code words are no isomorphic algebraic structures. These they could only be if A would be a power of two, which is an unsafe choice for A.

Figure 4.5 demonstrates what happens if two unsigned encoded values are subtracted whose functional versions cause an underflow if they are subtracted. The result of $x - y$ with $y > x$ in the domain of n-bit values is $2^n - (y - x)$ because $x - y$ wraps around at 2^n by $y - x$, i. e., the value by which y is larger than x. Thus, the expected result of the encoded version of this subtraction is $sub_{exp} = A * (2^n - (y - x)) + B_x - B_y$. Underflow

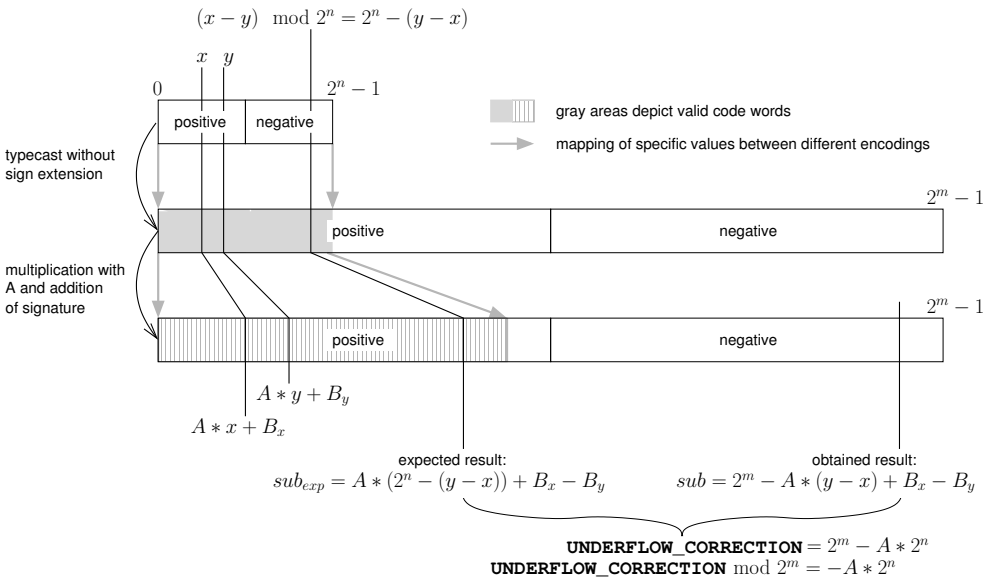


Figure 4.5.: Demonstration of the effects of encoding on the underflow behavior of an underflowing subtraction $x - y$ using unsigned encoded values.

However, the obtained result of $x_c - y_c$ wraps around 2^m resulting in the subtraction result $sub = 2^m - (A * y - A * x) + B_x - B_y = 2^m - A * (y - x) + B_x - B_y$. Hence, an underflown subtraction result has to be corrected using `UNDERFLOW_CORRECTION`$= sub - sub_{exp} = 2^m - A * 2^n$. This correction is applied to the encoded values and thus is used modulo 2^m. Thus, `UNDERFLOW_CORRECTION` $\mod 2^m = -A * 2^n$ will be subtracted for correcting underflown results of the unsigned encoded version of the subtraction.

Note that the signature correction function for the subtraction of unsigned encoded values ensures that $B_x < B_y$. This prevents underflows of the encoded values for the case that $x = y$ holds.

Sign underflow

For a subtraction with signed encoded numbers, the underflow is not the problem. However, subtractions that cross 2^{n-1}, that is, $x - y$ with $x \geq 2^{n-1}$ and $y < 2^{n-1}$ require additional corrections. Since this is similar to the sign overflow of an addition we call it a *sign underflow*. Subtractions with a sign underflow require corrective actions as Figure 4.6 demonstrates. The encoded version of y is $A * y + B_y$ because y is smaller than 2^{n-1}. However, the encoded version of x is $2^m - A * (2^n - x) + B_x$ because x is larger than 2^{n-1} and, thus, a sign extension is made during encoding of x. The expected result of this subtraction would be $sub_{exp} = A * (x - y) + B_x - B_y$ because the functional values do not underflow. However, the result obtained by subtracting $x_c - y_c$ is $sub = 2^m - A * (2^n - x) + B_x - (A * y + B_y) = 2^m - A * 2^n + A * (x - y) + B_x - B_y$. Hence, the subtraction of `UNDERFLOW_CORRECTION`$= sub - sub_{exp} = 2^m - A * 2^n$ is required for the encoded version of the subtraction of signed encoded values crossing 2^{n-1}.

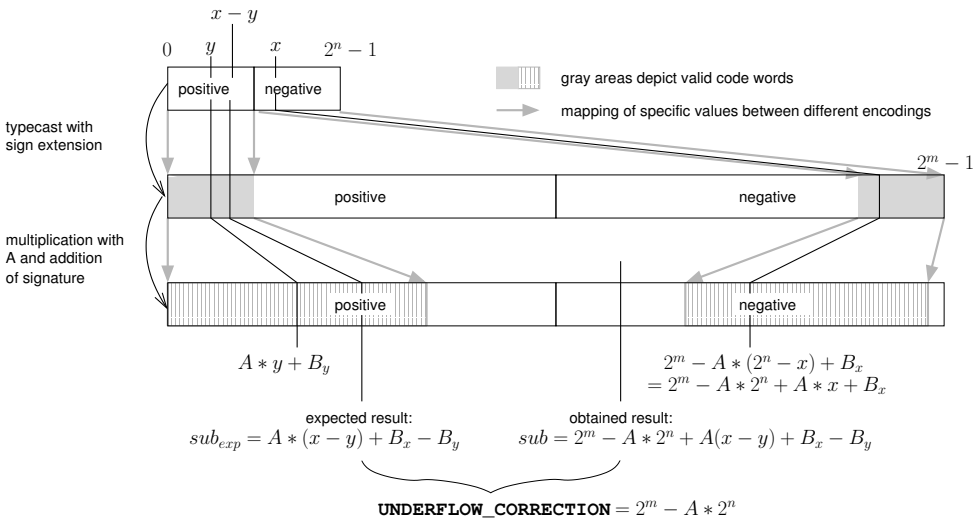


Figure 4.6.: Demonstration of the effects of encoding on the underflow behavior of an underflowing subtraction $x - y$ using signed encoded values.

Implementation

The pseudocode Listing 4.5 shows our implementation of an encoded subtraction that subtracts unsigned encoded values (line 6) and does an encoded underflow

```
1  const uint32_t B_UC = ...; // signature of underflow correction
2  const uint64_t UNDERFLOW_CORRECTION = 2^32 * A + B_UC;
3
4  uint64_t sub_anb(uint64_t xc, uint32_t Bx, uint64_t yc, uint32_t By){
5     // the actual subtraction
6     uint64_t result = xc - yc;
7
8     // expected signatures of if-condition check
9     uint32_t sigPos = Bx - By;
10    uint32_t sigNeg = (2^64 + sigPos) % A; // Note that our implementation
11                                           // ensures that 2^64 + sigPos
12                                           // does not overflow
13
14    // application of the underflow correction
15    // if an underflow happened (else branch)
16    if(xc > yc){
17       result = result + B_UC;
18    }else{
19       result = result + UNDERFLOW_CORRECTION - sigNeg + sigPos;
20    }
21
22    // encoded check of the previous if-statement's control flow
23    uint32_t sigCond = result % A;
24    result = result + sigCond;
25
26    return result;     // = A*((x-y) % 2^n) + Bx - By + sigPos + B_UC
27                       // = A*((x-y) % 2^n) + 2*(Bx - By) + B_UC
28 }
```

Listing 4.5: Encoded subtraction using unsigned encoded values and doing an underflow correction.

correction if required (lines 9 to 24). The underflow correction in Listing 4.5 as the overflow correction in Listing 4.4 uses an encoded if-statement to ensure that control flow errors influencing the correction are detectable. Again the corrections applied in line 19 in Listing 4.5 are summarized into one constant at encoding time for Compiler Encoded Processing, but not for Software Encoded Processing. This summarization into one constant ensures that all corrections are applied or if not, the result is an invalid code word. As previously described for the encoded addition, for Software Encoded Processing, due to the missing summarization error scenarios exist that might lead to undetectable left out underflow corrections.

As the addition, the subtraction fulfills our requirements that we defined in Section 4.2.1:

- There is no complete or partial decoding.
- If `sub_anb` is not completely executed, the result will not be a valid code word with the expected signature with high probability.
- The subtraction produces the signature $2 * (Bx - By) + B_{UC}$, which is not produced by any other encoded operation for the same input signatures B_x and B_y.

Multiplication

Value correction

For implementing an encoded multiplication, we have to solve several problems: When multiplying two encoded numbers x_c and y_c, several corrective actions are required to obtain a valid encoded result with a signature that only depends on the signatures of x_c and y_c and not on the functional values multiplied.

Overflow correction

Furthermore, as with addition and subtraction, we also require that the multiplication overflows correctly in the domain of functional values, i. e., wraps around 2^n. Therefore, further corrective actions are required. For the multiplication, the correctional value used depends on the input parameters because a multiplication may overflow, i. e., wrap around, several times and not only once as the addition does at most.

128-bit integer processing

Last, the intermediate results generated within the multiplication may require a data type that is twice as large as the type used for encoded values. In our case, where encoded values are 64-bit integers, that means that we have to store and process 128-bit values.

Currently, our encoding approaches run on x86-64-based systems with SSE4 extensions[3]. SSE4 on 64-bit systems provides us with 128-bit operations and, thus, solved the last problem described in the paragraph above. For platforms that do not provide a hardware implementation of 128-bit operations, we provide a software implementation of the operations required. However, these are much slower than the hardware implementation. In our following pseudocode listings we use the type definition `uint128_t` to declare 128-bit values.

Listing 4.6 demonstrates our first implementation of the encoded multiplication. The listing focuses on obtaining a valid code word as result. Hence, it does not implement the overflow correction. We present it because it seems to be an easy solution and we want to point out the problems of this first implementation.

```
1  // preconditions:
2  // x*By+y*Bx < A and Bx*By < A
3
4  uint64_t mult_anb(uint64_t xc, uint32_t Bx, uint64_t yc, uint32_t By){
5     uint128_t result=(uint128_t)xc*(uint128_t)yc;// result = A^2*x*y
6                                                  //          + A*x*By
7                                                  //          + A*y*Bx
8                                                  //          + Bx*By
9     uint128_t tmp1=(result/A % A) * A;           // tmp1 = A*x*By + A*y*Bx
10    uint64_t tmp2=result % A;                    // tmp2 = Bx*By
11    result=(result-tmp1)/A;                      // result = A*x*y
12    result+=tmp2;                                // result = A*x*y+Bx*By
13    return (uint64_t)result;
14 }
```

Listing 4.6: First version of an encoded multiplication using unsigned encoded values but without overflow correction and restricted functional values.

[3] SSE4 stands for Streaming SIMD extensions (version 4) and is an extension of the x86 instruction set.

This solution poses the following restrictions onto the encodable functional values and usable signatures:

- $x * B_y + y * B_x < A$
- $B_x * B_y < A$

Especially the first restriction is problematic because it constrains the functional values. In contrast, we choose the signatures, i. e., B's, at encoding time ourselves. Thus, restrictions on these can be implemented. Note that despite restricting the signatures being possible, restricting the signatures might reduce the safety because it reduces the amount of possible different signatures. However, the functional values should not at all be restricted because this restricts the user of an application instead of the encoding process. A further issue in Listing 4.6 is that before the last correction (in line 11) the result is available as partially decoded multiple of A: $A * x * y$. Thus, it is possible to exchange this value unnoticeable with another multiple of A. Obviously, these restrictions make this implementation practically unusable. Thus, we developed a second improved version that we use in our current encoding solutions and present in Listing 4.7

Value correction

In Listing 4.7, first, the actual multiplications and all corrections for obtaining a valid code word are executed in lines 14 to 22. Thereafter, `result` contains a correctly encoded multiplication result with the signature $B_x * B_y$. For implementing the corrections required, we need the functional values. These we can obtain by simple decoding without additional code checking (lines 9 and 10) because if any error modifies these values this will result in corrections that do not match the result of the multiplication in line 14. Also, all errors occurring in the corrections implemented from line 16 to line 22 lead to modifications that do not match the result of the multiplication in line 14. Thus, errors in the lines 9 and 10 and from line 16 to line 22 will result in an invalid code word in line 22 with high probability.

Overflow correction

Second, we have to handle overflows of the functional values that might happen. The functional values should wrap around 2^n where n is the size of the functional values in bits. We need this behavior, for example, for implementing encoded shift operations that are implemented as multiplication with powers of two as presented in Section 4.2.2. However, if the multiplication of the functional values overflows, the multiplication of the encoded values overflows differently as was already the case for the encoded addition. In contrast to the addition, a multiplication may wrap around several times depending on the functional values. Thus, we cannot use a predetermined correction, but have to compute the correction on execution time. Line 26 determines how often the multiplication has overflown. Therefore, we multiply the functional values in a larger domain (preventing the overflow) and divide the result obtained by 2^n. In the following an encoded if-statement is used to apply the correction if required.

Note that the computations of `quotient` in line 26 is unprotected. Errors modifying `quotient` might lead to undetectable errors because `quotient` is multiplied with a multiple of A in line 27. Thus, even if `quotient` is faulty, `result` will be a valid but possibly incorrect code word.

Due to space reasons, we do not present the encoded multiplication of signed

```
1  const uint64_t OVERFLOW_BOUND = 2^32;
2  const uint128_t OVERFLOW_BOUND_c = A * OVERFLOW_BOUND;
3  const uint128_t TWO_TO_128_MOD_A = 2^128 % A;
4
5  uint64_t mult_anb(uint64_t xc, uint32_t Bx, uint64_t yc, uint32_t By){
6
7      // Decoding the parameters for later corrections.
8      // ------------------------------------------------
9      uint32_t x = getFunctionalValue( xc );
10     uint32_t y = getFunctionalValue( yc );
11
12     // Actual multiplication with corrections.
13     // ------------------------------------------------
14     uint128_t result = (uint128_t)xc * (uint128_t)yc;
15     // result = A^2*x*y + A*x*By + A*y*Bx + Bx*By
16     result = result - (uint128_t)A *
17         ((uint128_t)x * (uint128_t)By + (uint128_t)y * (uint128_t)Bx);
18     // result = A^2*x*y + Bx*By
19     result = result + (uint128_t)(A-1) * (uint128_t)Bx * (uint128_t)By;
20     // result = A^2*x*y + A*Bx*By
21     result = result / A;
22     // result = A*x*y+Bx*By
23
24     // Encoded overflow correction if required.
25     // ------------------------------------------------
26     uint64_t quotient = ((uint64_t)x * (uint64_t)y) / OVERFLOW_BOUND;
27     uint128_t overflow_correction = OVERFLOW_BOUND_c * quotient;
28
29     uint128_t sigPos = Bx * By;
30     uint128_t sigNeg = (TWO_TO_128_MOD_A + sigPos) % (uint128_t)A;
31
32     if ( quotient >= 1 ){
33         result = result - overflow_correction;
34     }else{
35         result = result + sigPos - sigNeg;
36     }
37
38     // encoded check of the previous if-statement's control flow
39     uint128_t sigCond = (result - OVERFLOW_BOUND_c) % A;
40     result = result + sigCond;
41
42     return (uint64_t)result;    // result = A*x*y+Bx*By+sigPos
43                                 // result = A*x*y+2*Bx*By
44 }
```

Listing 4.7: Final version of our encoded multiplication using unsigned encoded values.

encoded values in detail. Its value correction is similar. The only difference is that signed integer types are used instead of unsigned ones. The overflow correction uses other boundaries for determining if an overflow happened. The reason is that the domain of signed encoded values is split up in the middle. This we explained before when describing the encoding operations and the encoded addition.

Future work

To the best of our knowledge, this encoded multiplication does no partial decoding, is atomic, and produces a unique signature. The only unprotected part that we know of is the computation of `quotient`. However, up to now the encoded base operations were only subject to careful reviews and to error injections in the context of our evaluations of the encoding schemes SEP and

CEP presented in chapters 7 and 8 respectively. While that might be sufficient for the relatively simple addition and subtraction, it clearly is not for the rather complex multiplication.

In the future, we propose that the safety of the encoded base operations is assessed more sophisticated. For example, an extensive error injection can be applied to them because they contain a more manageable amount of possible error injection points than complete applications. We also think that the encoded base operations lend themselves to the application of more formal methods such as model checking or symbolic error injection for assessing their correctness and efficacy with respect to error detection.

Division

If ANB- or ANBD-encoded values are directly divided, the obtained result is unusable. Only the direct division of AN-encoded values produces reasonable results. We could emulate ANB- and ANBD-encoded division operations completely in software without using the hardware implemented division operation of the processor using encoded versions of a while-loop, a comparison, a subtraction, and an addition. Pseudocode Listing 4.8 demonstrates the unencoded but encodable implementation of the software-implemented division. However, the processor's division implementation is surely faster than a software solution. Thus, we use an implementation of the encoded division that utilizes the processor's division implementation, but is partly only AN-encoded. The advantage of the decreased execution time of this solution comes with the disadvantage of potentially undetected exchanged operand errors.

```
1 /* Return x/y */
2 uint32_t div(uint32_t x, uint32_t y){
3    uint32_t result = 0;
4    while( x >= y ){
5       result++;
6       x = x - y;
7    }
8    return result;
9 }
```

Listing 4.8: Unencoded version of a software implemented unsigned division.

Note that the division $\frac{x_c}{y_c}$ of the AN-encoded values x_c and y_c results in a completely decoded functional value $\frac{A*x}{A*y} = \frac{x}{y}$ because A is contained in both – dividend and divisor. To prevent this, we have to multiply the divisor x_c with A before dividing it by y_c. Note that, thus, the encoded division requires in our case data types and operations that are able to handle at least integers with the size of 96 bits. Therefore, we will again use the SSE4 extensions that enable us to process integers as large as 128 bits.

Divisibility

Furthermore, we have to ensure that the implemented integer division truncates the result correctly. While the AN-encoded integer division $\left\lfloor \frac{x_c}{y_c} \right\rfloor = \left\lfloor \frac{A*x}{A*y} \right\rfloor = \left\lfloor \frac{x}{y} \right\rfloor$ results in a correctly truncated but decoded result, the correctly AN-encoded integer division $\left\lfloor \frac{x_c*A}{y_c} \right\rfloor = \left\lfloor \frac{A*x*A}{A*y} \right\rfloor = \left\lfloor \frac{x*A}{y} \right\rfloor$ leads to a result that is not a valid AN-code word in cases where y does not divide x without a remainder. Thus, we have to complete the division with a correction that ensures the divisibility

of dividend and divisor, that is, the division is remainder-free. We call this correction truncation correction and use the variable `truncCorrection` in the following to store its value.

The pseudocode in Listing 4.9 presents the implementation of our encoded unsigned division whose operands are unsigned encoded. Note that our signature correction function for the division ensures that B_y is never zero to prevent a division by zero.

```
1  uint64_t divu_anb(uint64_t xc, uint32_t Bx, uint64_t yc, uint32_t By){
2     // compute correction that ensures divisibility
3     uint32_t x = getFunctionalValue( xc );
4     uint32_t y = getFunctionalValue( yc );
5     uint64_t truncCorrection = x % y;
6
7     // remove signatures and apply divisibility correction
8     // after that step the operations are only AN-encoded
9     xc = xc - Bx - A * truncCorrection;
10    yc = yc - By;
11
12    // execute the unsigned division
13    uint64_t result = ((uint128_t)xc * (uint128_t)A) / yc;
14       // result = A * floor( x/y )
15
16    // apply a signature to the result
17    uint64_t Br = Bx/By;
18    result = result + Br;
19
20    return result; // result = A*floor(x/y) + Bx/By
21 }
```

Listing 4.9: Encoded unsigned division using unsigned encoded operands and producing and unsigned encoded result.

Lines 3 to 5 compute `truncCorrection`. This can be done unencodedly. Any error in the truncation correction most likely results in the following division of the encoded values being not remainder-free, that is, $(A*x-A*truncCorrection)*A$ (computed in line 9) will not be divisible by $A*y$ if $truncCorrection$ is erroneous. As stated before, if this division is not remainder-free, the result will not be a valid AN-code word.

Signedness

For a division signedness matters, that is, an unsigned division produces different results than a signed division of the same values. Values are the same if they have the same bit pattern, which could be interpreted unsigned or signed in two's complement. Assembler languages such as the DLX or the LLVM instruction set reflect this signedness issue by providing different operations for signed and unsigned division. This signedness issue applies also to the encoded versions of these operations: If we want to execute a signed division, we have to do a signed division of signed encoded values. If we want to execute an unsigned division, we have to do an unsigned division of unsigned encoded values.

Listing 4.10 presents our implementation of a signed division that takes unsigned encoded values as input. Thus, we have to transform the parameters to their signed encoded equivalents before dividing them. Therefore, we use for ANB-encoded values the encoded function `unsignedToSigned_anb`, which

we described in Section 4.1. The result obtained by the following AN-encoded division is signed encoded. If the result returned shall be unsigned encoded, we have to transform it back using the function `signedToUnsigned_anb`, which we also described in Section 4.1.

In contrast to the division itself, the signedness transformations are completely encodable for the AN-, ANB-, and ANBD-code. This includes that the signature of the values is changed, that is, the transformed value has a different signature than the input value and the signature produced depends on the input signature and the operation. For an increased comprehensibility, the pseudocode in Listing 4.10 ignores this signature modification and assumes that the sign transformations do not change the signature.

```
1  uint64_t div_anb(uint64_t xc, uint32_t Bx, uint64_t yc, uint32_t By){
2     // compute correction that ensures divisibility
3     int32_t x = getFunctionalValue( xc );
4     int32_t y = getFunctionalValue( yc );
5     int64_t truncCorrection = x % y;
6     if( truncCorrection  < 0){
7        truncCorrection += A;
8     }
9
10    // transform parameters from unsigned to signed encoding
11    int64_t xc_s = unsignedToSigned_anb( xc, Bx );
12    int64_t yc_s = unsignedToSigned_anb( yc, By );
13
14    // remove signatures and apply divisibility correction
15    // after that step the operations are only AN-encoded
16    xc = xc_s - (int64_t)A * truncCorrection - (int64_t)Bx;
17    yc = yc_s - (int64_t)By;
18
19    // execute the signed division
20    int128_t dividend = (int128_t)xc_s * A
21    int64_t result = dividend / yc_s;
22
23    // apply a signature to the result
24    int64_t Br = Bx/By;
25    result = result + Br;
26
27    return signedToUnsigned_anb( (uint64_t)result, Br );
28 }
```

Listing 4.10: Simplified encoded signed division using unsigned encoded operands and producing an unsigned encoded result.

Comparisons

For encoding applications, we need encoded versions of the following comparison operations:

- the equality and inequality comparison, which is equivalent for unsigned and signed values,
- signed less-equal, greater-equal, less, and greater comparisons, and
- unsigned less-equal, greater-equal, less, and greater comparisons.

Note that for comparisons (apart from equality and inequality) signedness matters. Thus, we have to provide encoded versions for signed and unsigned comparisons as well.

Equality and Inequality Comparison For AN-encoded equality and inequality comparisons the encoded values can be directly compared. Afterwards, an unchecked if-statement is used to adapt the result that is either 0 for `false` or 1 for `true` to a code word, that is, either to 0 or to A.

AN-encoded

The ANB- and ANBD-encoded versions of equality and inequality are realized by combining greater and less comparisons with a boolean `and`. For example, the equality comparison $x == y$ can be replaced with $(x >= y)$ and $(x <= y)$. This we implement as a replacement operation, that is, for ANB- and ANBD-encoding, equality and inequality comparisons are replaced with their encodable versions before the actual encoding is done.

ANB- or ANBD-encoded

Less, Greater, Less-equal, and Greater-equal Comparisons Forin in [For89] presented an encoded if-statement whose condition is a comparison with zero. This is so far the only known form of an encoded comparison. All our encoded comparisons presented in the following use the same principle as Forin's encoded if-statement. They use an encoded if-statement whose condition implements the intended comparison and in the branches assigns the appropriate encoded boolean value to the result. Therefore, we have to provide a way to compute a predictable signature for the branching condition that depends on the condition's result, that is, the signature of the branching condition has one value if the condition is true and another if it is false.

Remember that the encoded if-statement ensures that the result produced has only one predictable signature independent of the branch taken. The encoded boolean values returned by such an encoded comparison can be used in further processing, for example, by boolean operations such as `and` or as condition of an if-statement or a loop.

In the following, we first present unsigned comparisons, followed by signed comparisons.

Listing 4.11 presents our encoded implementation of an unsigned less-equal comparison operation that returns the encoded version of `true` if the functional value of x_c is smaller than or equal to the functional value of y_c. Otherwise, the encoded version of `false` is returned. The implementation presented takes unsigned encoded values as parameters and returns an unsigned encoded result.

Unsigned comparisons

If two unsigned encoded values y_c and x_c are subtracted without underflow correction, the signature of the result may have two different but deterministic values: $(B_y - B_x)$ if the subtraction is not underflown, and $(((2^m + (B_y - B_x)) \mod A)$ if it is underflown. This we use to implement an encoded if-statement that checks the comparison that itself is executed unencoded. Note that the two signatures are only guaranteed to be different if A does not divide 2^m.

```
1  const uint64_t TWO_TO_64_MOD_A = 2^64 % A;
2
3  /* unsigned encoded unsigned less-equal of xc and yc*/
4  uint64_t setleu_anb (uint64_t xc, uint32_t Bx, uint64_t yc, uint32_t By){
5
6     // unencoded comparison that uses the unsigned AN-encoded values
7     uint64_t result = (xc - Bx) <= (yc - By);
8        // result is either
9        // 1 if x <= y or
10       // 0 otherwise
11
12    // encoded check of comparison
13    uint64_t diff = yc - xc;// if x > y,  diff = 2^m - (xc-yc) (underflow)
14                            //                = 2^64 - (xc-yc)
15                            // if x <= y, diff = yc-xc
16    uint64_t sigCond = diff % A;
17    uint64_t sigPos = By - Bx;
18    uint64_t sigNeg = (TWO_TO_64_MOD_A + sigPos) % A;
19
20    if( result ){
21       // expected: sigCond == sigPos
22       result += (A-1);
23    }else{
24       // expected: sigCond == sigNeg
25       result += (sigPos - sigNeg);
26    }
27    result += sigCond;
28
29    return result; // result = A + sigPos if x <= y
30                   // result = sigPos     otherwise
31 }
```

Listing 4.11: Encoded unsigned less-equal comparison operation.

In line 7 of Listing 4.11 the comparison is executed on the only AN-encoded values. Further decoding – apart from removing the signatures – is not required because unsigned encoding preserves the size relations of unsigned values. The remaining code of `setleu_anb` checks this comparison in an encoded fashion and ensures that the returned value is a valid code word in the error-free case. Therefore, the two compared values are subtracted from each other in line 13 without underflow correction. This subtraction underflows if $x_c > y_c$ which is only the case if $x > y$ because we are using unsigned encoded values and our signature correction function for this comparison ensures that $B_x < B_y$. The latter is required because otherwise the subtraction would also underflow if $x = y$ and $B_x > B_y$ hold. On the other hand, the subtraction does not underflow if $x \leq y$. As explained before, the underflown subtraction result will result in a different signature in line 16 than the non-underflown subtraction result. This feature is used by the following encoded if-statement (lines 20 to 27), which transforms `result` into a valid code word, to check the unencoded comparison of line 7. Finally, a code word is returned that contains either $A + sigPos$ to represent `true` or $sigPos$ to represent `false`.

As for our previous pseudocode examples, the signatures and correctional values (here B_x, B_y, $sigPos$, and $sigNeg$) are constants for Compiler Encoded Processing (see Chapter 8). Thus, they can be precomputed at compile time and the application of the corrections in line 25 of Listing 4.11 is either completely

executed or completely lost. The latter results in an invalid code-word as a result with high probability.

Note that after line 22 the result is only AN-encoded until line 27. This could be prevented by extending the signature by adding another constant `sigIf` in both the if- and the else-branch. The returned code word would than be either $A + sigPos + sigIf$ or $sigPos + sigIf$. Note that this signature should be different for each instance of such a comparison. Otherwise, the intermediate results of the signatures could be exchanged unnoticeable. If such an extension is really necessary, should be decided based on an extensive analysis of the encoded operations either by error injection using one of the tools presented in Chapter 9 or by analytical methods such as symbolic error injection or model checking. If these methods show that the described scenario is a severe vulnerability, actions should be taken and the additional signature should be added.

Other unsigned comparisons are implemented similarly as the presented less-equal comparison. For the greater-equal comparison, just the parameters are exchanged with each other. For the less and the greater comparison, we have to ensure that the subtraction also underflows if the functional values compared are equal.

Signed comparisons

Signed comparisons are more complicated than unsigned comparisons. Figure 4.7 in the first line demonstrates the structure of the domain of unsigned functional, i. e., unencoded, values. These are completely ordered and can be compared directly. On the other hand, if the same values are interpreted as signed values using the two's complement, as is done in the second line of Figure 4.7, they break down into two intervals – the positive and the negative values. Within these intervals the values are completely ordered. However, the value of the negative values if interpreted unsigned is larger than the that of the positive values. When encoding values, this relation is conserved no matter which kind of encoding is used. Thus, a direct comparison of unsigned encoded values will not result in a correct result if one of the values is positive and the other negative.

One solution would be to transform the unsigned encoded parameters into signed encoded values and to use signed comparisons to directly compare them. Furthermore, we would have to design an encoded check for this otherwise unencoded comparison. Another, simpler solution is: We shift the signed interpreted numbers so that we can compare them directly using the unsigned comparison operations.

We implemented both versions and our runtime measurements showed no measurable differences. Thus, we decided to use the simpler and easier to implement second solution because its implementation surely is less error-prone.

Figure 4.7 in the third row depicts this simple solution. We add to both values that are compared 2^{n-1}. Of course, this is done using the encoded addition operation and adding the encoded version of 2^{n-1}. By adding 2^{n-1}, we move the positive values to the upper part of the domain and the negative ones to the lower part. This effect is caused by the overflow behavior of the addition. After that addition, for example, 0 becomes 2^{n-1} and -1 that is represented by $2^n - 1$ becomes $2^{n-1} - 1$. The reason is that the addition wraps around when crossing

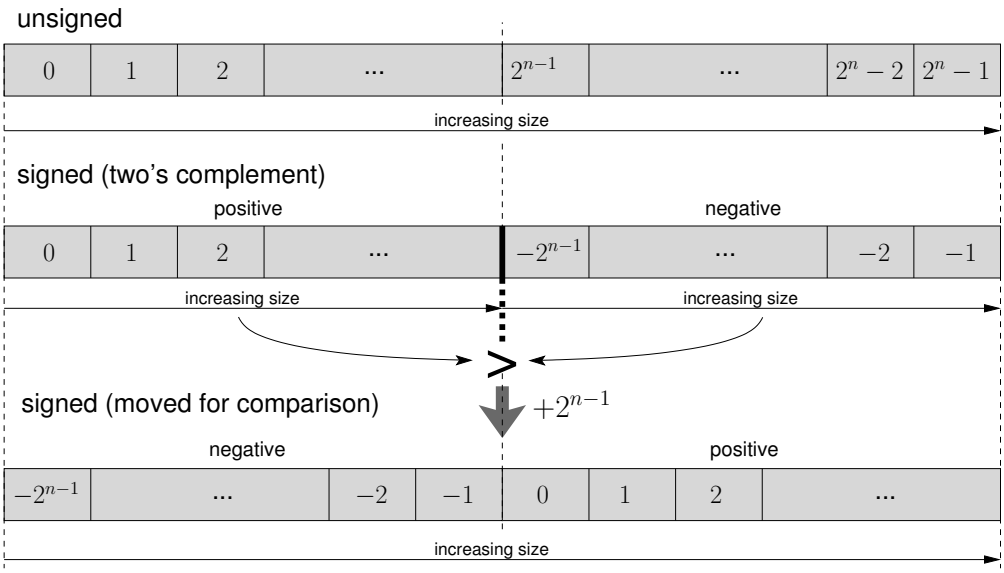


Figure 4.7.: Less-than relation for different number representations.

2^n. The resulting values can be compared using the encoded implementations of the unsigned comparison operations.

Listing 4.12 presents the pseudocode of the described solution. Note that this pseudocode is simplified with respect to the signature handling. It just assumes that no corrections of the signatures of the parameters are required, that means that all called encoded operations can handle the given signatures and have no size constraints for the signatures of their parameters. We normally use special signature precomputation and correction functions to ensure that the restrictions of encoded operations with respect to signatures are met. Calls to these functions we left out in the pseudocode presented.

Note, furthermore, that the signature precomputations for Compiler Encoded Processing (see Chapter 8) are done at encoding, i. e., compile, time. Also the computations in line 17 and 18 of Listing 4.12 result in constants that are computed at encoding time for Compiler Encoded Processing. Thus, their computation can be assumed to be error-free.

Boolean Logical Operations

For encoding the LLVM-intermediate representation or DLX-binaries we need three logical operations: `and`, `or`, and `xor`. Their boolean, i. e., one-bit, variants we implement as encoded base operations using arithmetic operations. We describe these implementations in the following. Their bitwise variants are implemented as replacement operations and we describe them in Section 4.2.2.

Boolean and

The precondition for our implementation of the boolean functions is that `true` is represented by the functional value 1 and `false` is represented by the functional

```
1  const uint64_t FIRST_NEGATIVE_C = 2^31 * A + B_FN;   // n = 32
2
3  /* signed less-than comparison */
4  uint64_t setlt_anb (uint64_t xc, uint32_t Bx, uint64_t yc, uint32_t yc){
5     // Shift both parameters by half of the domain size of the
6     // functional values along the ring formed by the functional values.
7     // Thereby, they are made directly comparable using the unsigned
8     // comparison function.
9     uint64_t xs_c = add_anb(xc, Bx, FIRST_NEGATIVE_C, B_FN);
10    uint64_t ys_c = add_anb(yc, By, FIRST_NEGATIVE_C, B_FN);
11
12    // signatures of the previous results:
13    //-----------------------------------
14    // Remember that the definition of the signature of the result of
15    // add_anb(ac, Ba, bc, Bb) is 2*(Ba+Bb)-B_OC
16    // These signatures are constant at runtime.
17    uint32_t Bxs = 2*(Bx+B_FN)-B_OC;
18    uint32_t Bys = 2*(By+B_FN)-B_OC;
19
20    // the actual comparison - now possible as an unsigned comparison
21    return setltu_anb(xs_c, Bxs, ys_c, Bys);
22 }
```

Listing 4.12: Encoded signed less-than operation with simplified signature handling.

value 0. In that case, the boolean `and` operation can be implemented using the multiplication. Because of the size restrictions of its operands, this multiplication will never overflow. This facilitates the usage of an implementation of the multiplication operation that does not correct overflows.

Boolean xor

The boolean `xor` operation is equivalent to a 1-bit addition, that is, an addition that overflows if the sum is greater than 1.

Boolean or

The implementation of `or` also uses an encoded addition. However, the addition used overflows differently. If the addition result obtained represents 2, it *"wraps"* to 1 instead of 0. A larger addition result is not possible because the input values are restricted to 0 and 1. Listing 4.13 contains the pseudocode of our encoded boolean `or`.

Runtime Overhead

The presented encoded base operations are slower than their unencoded native counterparts. We did measure the slowdown generated by AN- and two different ANB-encoded versions of the base operations. We did not evaluate our ANBD-encoded operations because currently none of our encoding approaches uses them.

For determining the slowdown of the encoded operations, we executed each operation to be measured for a specific amount of time – 15 seconds for the presented measurements – in an endless loop and counted the number of iterations executed until the execution was aborted by the timer signal. This we did for native and encoded versions. The obtained results were used to compute the slowdown of the different encoded versions compared to the native version. The

```
1  const uint64_t TWO_TO_64_MOD_A = 2^64 % A;
2  const uint64_t OVERFLOW_CORRECTION = A + B_OC;
3
4  uint64_t or_anb (uint64_t xc, uint32_t Bx, uint64_t yc, uint32_t By){
5     // might result in encoded representations of
6     // 0, 1, or 2
7     uint64_t result = xc + yc;
8
9     // correct result to a representation of 1 if it is 2
10    // using an encoded if statement
11    uint64_t sigPos = Bx + By;
12    uint64_t sigNeg = (uint64_t)(TWO_TO_64_MOD_A + sigPos) % A;
13    if( result >= 2*A ){ // correct if result represents 2
14       result = result - OVERFLOW_CORRECTION;
15    }else{
16       result += (sigPos - sigNeg - B_OC);
17    }
18    uint64_t diff = (uint64_t)(result - 2 * A);
19       // if result represented 2, no underflow happens
20       // otherwise,               underflow happens
21    uint64_t sigCond = diff % A;
22    result = result + sigCond;
23
24    return result;
25 }
```

Listing 4.13: Encoded boolean or operation.

slowdown is defined as slowdown $= \frac{\text{number of iterations for native operation}}{\text{number of iterations for encoded operation}}$. The measurements were executed on a machine that has two Intel Xeon processors with in total 8 cores and runs a 64-Bit Fedora 10. The measurements were not parallelized and we tried to reduce any other load on the machine during the experiments as much as possible.

Note that we measured two different ANB-encoded versions:

encoded overflow correction These versions use an encoded if-statement for applying (or not applying) the overflow correction in additions, subtractions, and multiplications. Note that these are the implementations that we presented in this chapter.

unencoded overflow correction In contrast to the versions with encoded overflow correction, these versions use an unencoded if-statements to implement the overflow correction. Otherwise, the implementations are equivalent to the ones presented in this chapter.

We introduced this less safe version of the ANB-encoded operations as a reaction to the relative high slowdowns observed for the ANB-encoded operations with encoded overflow correction.

The measured AN-encoded versions are similar to the ANB-encoded versions, but do not contain any signature handling. Thus, they are much simpler and are much faster.

Figure 4.8 depicts the measured slowdowns for the encoded operations that handle encoded values that represent 32-bit functional values and 1-bit boolean values respectively. The results for 8-bit and 16-bit operations look similar.

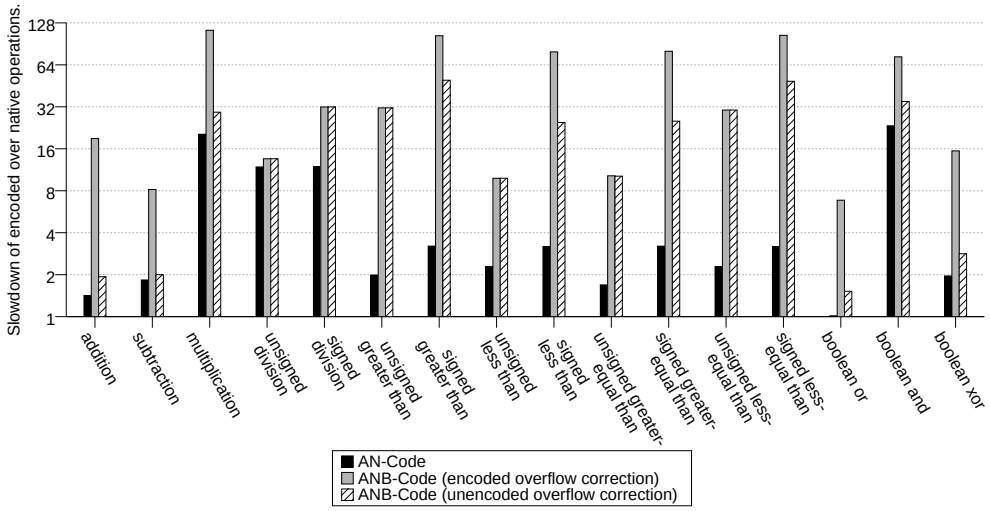


Figure 4.8.: Slowdowns of AN- and differently ANB-encoded operations compared to unencoded native operations.

Comparing different codes

As predicted the AN-encoded operations are much faster than both of the ANB-encoded versions for all operations. The ANB-encoded versions of additions, subtractions, and multiplications that use an encoded overflow correction are considerably slower than the ones using an unencoded overflow correction. The same is true for the signed comparisons and the boolean logical operations. The reason is that these operations either use an encoded addition or multiplication. There are no differences for divisions and unsigned comparisons because they do not use any overflow correction.

Comparing different operations

The encoded versions of additions, subtractions, `or` and `xor` are the operations with the least slowdowns. Encoded multiplications, divisions, and boolean `and` operations are expensive. The reason is the processing of 128-bit integers that is required for these operations. The ANB-encoded comparisons are expensive due to the required encoded if-statement. Their AN-encoded versions are not much slower than the addition and subtraction. For the division operation the differences between the three codes are less pronounced because the ANB-encoded versions of the division are only AN-encoded. Their higher slowdown just comes from the signature removal and in case of the signed division from the transformations between signed and unsigned encoded values.

Future work

So far, we did not separately evaluate the error detection capabilities of the encoded base operations. However, they are part of our error injection experiments that we applied to whole encoded applications. For the results see sections 7.4.1 and 8.5.3.

4.2.2. Encodable Replacement Operations

Encoding by hand is a tedious and error-prone task. Hence, we automated as much of the remaining encoding tasks as possible by providing a library of so-called *encodable replacement operations*. This library contains implementations of the following operations:

- shifts,
- modulo operations,
- casts,
- bitwise logical operations,
- unaligned memory accesses, and
- inequality and equality comparisons for ANB-encoding.

The replacement operations are written in such a way that they can be automatically encoded. They are currently only used for Compiler Encoded Processing. Before executing the actual encoding pass, our encoding compiler replaces all otherwise non-encodable operations with their appropriate encodable replacement operation. These replacement operations are described in the following. In Software Encoded Processing these operations either do not exist or remain unencoded and, thus, unprotected.

Shift Operations

Encoded versions of arithmetic and logic shift operations can be implemented using division and multiplication with powers of two because a `<<` k is equivalent to $a * 2^k$ and a `>>` k is equivalent to $\frac{a}{2^k}$. For obtaining the 2^k required, we use a tabulated *power-of-two* function named `powerOfTwo` with precomputed values. The function `powerOfTwo` takes k with $0 \leq k \leq 32$ as parameter and returns 2^k. It obtains 2^k from a table. The function `powerOfTwo` is also implemented natively, i. e., unencoded, and is automatically encoded. In contrast to the logic right shift, an arithmetic right-shift additionally requires a sign-extension to be made if the shifted value is negative. In that case all new high-order bits have to be set to one.

The pseudocode in Listing 4.14 represents the encodable variant of the 8-bit arithmetic right shift operations. The presented function `ashr8` shifts `val k` bits to the right. Logical right shift and left shift and shift operations for the different sizes of functional values use the same principles.

While we replace arithmetic right shifts with a call to a function containing its replacement implementation, we replace logical shifts directly in the source code with the required division or multiplication respectively.

Modulo Operations

Modulo operations we replace using a division, a multiplication, and a subtraction operation. See Listing 4.15 for the implementation. In the last line the variable

```
1  int8_t ashr8 (int8_t val, int8_t k){
2     const static uint8_t signExt[]={0,0x80,0xC0,0xE0,0xF0,
3                                     0xF8,0xFC,0xFE,0xFF};
4     if (val < 0){
5        // shift and set new high-order bits to one
6        uint8_t shifted = (uint8_t)val / (uint8_t)powerOfTwo((uint8_t)k);
7        return shifted + signExt[(uint8_t)sh];
8     }else{
9        // shift
10       return val / powerOfTwo((uint8_t)k);
11    }
12 }
```

Listing 4.14: Encodable arithmetic shift right operations for 8-bit functional values.

`mod` contains the modulo of the division $\frac{a}{b}$ This replacement is also directly generated within the code without adding a call to a dedicated function.

```
1 // The following code computes the modulo a%b in an encodable fashion.
2 // We assume a and b have the type uint32_t.
3
4 uint32_t q = a/b;
5 uint32_t p = b*q;
6 uint32_t mod = a-p;
```

Listing 4.15: Implementation of an encodable modulo operation.

Cast Operations

Cast operations that transform variables of one type into another type also have to be emulated using the encoded base operations because they might change the value stored in the variables. We make these changes that are implicitly described by the cast explicit. Thereby, we support their automatic encoding by the following encoding steps.

We have to handle the following kinds of integer cast operations:

downcasts that cast from a larger type to a smaller.

signed upcasts that cast from a smaller to a larger type with a sign extension, that is, the newly introduced higher-order bits have the same value as the highest-order bit of the source value.

unsigned upcasts that cast from a smaller to a larger type without a sign extension, that is, the newly introduced higher-order bits are always zero.

Furthermore, we support casts from integers to pointers and vice versa. However, since pointers are already represented as integers in encoded programs, nothing has to be done for handling these casts. Other casts are not required because other types are not directly supported.

Downcasts

For downcasts we use a modulo computation with the appropriate power of two to reduce the size of the contained value so that it fits into the type. The used power of two is hard-coded into our cast implementation. The modulo

operation we implement in an encodable way using division, multiplication, and subtraction. If, for example, the 32-bit integer a is downcast to 8 bit, our replacement operation `trunc_32_2_8` for this downcast computes the new value like this: $a \mod 2^8 = a - (256 * \lfloor \frac{a}{256} \rfloor)$.

Signed upcasts

Signed upcasts need to check if the casted value is negative, that is, we check if its highest-order bit is set. If that is the case, a sign extension has to be made by adding the appropriate sign bits. If the bit is not set, no changes to the value are required. Assume that we cast a negative 8-bit integer a to a signed 16-bit type. Therefore, we have to execute the encoded version of $a = ff00_{hex} + a$. If a is positive, no sign extension, that is, no addition of $ff00_{hex}$ is required. The constants required for adding the sign extension are also hard-coded into the implementations and encoded automatically by the encoding steps that follow.

Unsigned upcasts

Unsigned upcasts from smaller to larger unsigned types require no further actions because the newly introduced higher-order bits are set to zero automatically.

Bitwise Logical Operations

We have different possibilities to implement encodable bitwise logical operations. The naive approach is to use shift and addition operations to compute every bit individually using the boolean logical operations for which appropriate encoded base operations exist. That would generate a huge runtime overhead. For a 32-bit logical operation, alone for each operand 31 left and 31 right shifts would be required to obtain the single bits. For each of these bits the boolean logical operations would have to be executed. Further encoded shifts are needed to assemble the result. All these shifts have to be implemented using encoded multiplications and divisions, which are especially expensive.

On the other hand, we could completely tabulate the results. In that way we implemented our encodable `powerOfTwo` function. However, the memory consumption of this approach used for bitwise logical operations makes it unusable. Alone for one 32-bit bitwise logical operation, we would need a table with $2^{32} * 2^{32}$ entries. The encoded version of this table would require $134,217,728$ Tera Byte of memory.

Thus, we decided to combine these two approaches. We use tabulated results of logical operations. However, we only tabulate 8-bit chunks and combine these chunks using shift operations and arithmetic operations. The pseudocode in Listing 4.16 demonstrates this approach for the 32-bit `or` operation. The other bitwise logical operations, `and` and `xor`, are implemented similarly. Note that the required shift operations are implemented using an encodable multiplication with a power of two. The powers of two used in Listing 4.16 are constants and are only depicted as explicit powers to improve readability.

Unaligned Memory Access

We chose to encode the memory at 32-bit granularity because we assume that most programs mainly operate on 32-bit values. This means every 32-bit word

```
1  // encodable modulo implementation
2  uint32_t urem32 (uint32_t dividend, uint32_t divisor){ ... }
3
4  // encodable get_bytex(uint32_t w) returns the x-th byte of w
5  uint32_t get_byte0(uint32_t w){return urem32(w, 2^8); }
6  uint32_t get_byte1(uint32_t w){return (urem32(w, 2^16) - get_byte0(w))
7                                 / (2^8); }
8  uint32_t get_byte2(uint32_t w){return (urem32(w, 2^24) - urem32(w, 2^16))
9                                 / (2^16); }
10 uint32_t get_byte3(uint32_t w){return w / (2^24); }
11
12 // precomputed results for 8-bit chunks
13 static uint32_t const table_or8[256][256] =
14 {0x00, 0x01, 0x02, ...};
15
16 // encodable bitwise logical or for 32-bit integers
17 uint32_t or32( uint32_t a, uint32_t b ){
18    uint32_t byte0 = table_or8 [get_byte0(a)][get_byte0(b)];
19    uint32_t byte1 = table_or8 [get_byte1(a)][get_byte1(b)] * (2^8);
20    uint32_t byte2 = table_or8 [get_byte2(a)][get_byte2(b)] * (2^16);
21    uint32_t byte3 = table_or8 [get_byte3(a)][get_byte3(b)] * (2^24);
22
23    return byte3 + byte2 + byte1 + byte0;
24 }
```

Listing 4.16: Encodable bitwise logical or implementation for 32-bit values.

in memory is stored as an encoded 64-bit word. Thus, we need to adapt every load to and store from memory because they have to map the original address to the appropriate address of the encoded value. How this address mapping is done depends on the memory implementation of the actual encoding and is described for Software Encoded Processing in Chapter 7 and for Compiler Encoded Processing in Chapter 8.

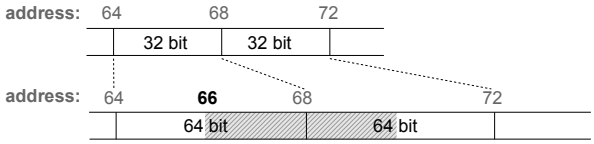
Due to this choice, all memory accesses in the program that we want to encode have to be aligned to 32-bit,i. e., 4-byte, boundaries. Thus, before encoding the program we replace all potentially unaligned loads and stores with implementations that implement these operations using loads and stores that are aligned to 32-bit boundaries.

Figure 4.9 demonstrates how an unaligned 32-bit load from address 66 is executed. First, both aligned addresses that are affected by the 32-bit load at address 66 are read, that is, the 32-bit values from

- the first 32-bit aligned address before 66 (64) and
- the next 32-bit aligned address after 66 (68)

are read. Afterward, the relevant parts of the values read are extracted and put together to one value using our (encodable) bitwise logical operations. Note that the shift operation implemented by the multiplication with 2^{16} automatically removes the upper 16 bits of `a1` during the combination. Unaligned loads of 8- or 16-bit values are implemented similarly. However, an 8-bit load always requires only one aligned 32-bit load instead of two.

For an unaligned store we first have to load the affected addresses. This requires two aligned loads for 16-bit and 32-bit unaligned stores and one aligned load




```
executed instructions for
load32_unaligned( 66 ):
  a1 = load32_aligned(64)
  a2 = load32_aligned(68) / 2^16
  result = (a1 * 2^16) + a2
```

Figure 4.9.: Execution of an unaligned load at address 66. The upper part represents the memory layout of the original program, the lower part that of the encoded program but with unmapped addresses.

for an 8-bit unaligned store. The read words are then modified accordingly and written back. To prevent accessing unallocated memory when executing an unaligned store, we adapt the size of all allocated memory regions to be a multiple of 32 bits. Note that we zero-initialize all allocated memory regions.

In that way we also implement loads and stores for granularities smaller than 32 bit. We provide unaligned loads and stores for 8, 16, and 32 bit.

Normally, we do not know at the time when we have to replace loads and stores with their unaligned implementations if these loads and stores are really unaligned, that is, go to an address that is no multiple of 32. Thus, we would have to replace all loads and stores with our implementation for unaligned loads and stores. However, during our experiments with Compiler Encoded Processing we observed that most of our benchmarks do not contain unaligned 32-bit loads and stores. Thus, we optimized our encoding by usually not using the unaligned load and store implementations presented in this section for 32-bit loads and stores.

Equality and Inequality Comparisons

As explained in Section 4.2.1, for ANB- and ANBD-encoding, equality and inequality are made encodable by realizing these operations using a combination of greater and less comparisons with a boolean `and`. This replacement is directly generated within the code without adding a call to a dedicated function.

For AN-encoding we provide equality and inequality comparisons as encoded base operations because AN-encoded values can be directly compared.

Runtime Overhead

Figure 4.10 depicts the slowdowns of the unencoded versions of our replacement operations compared to the native versions of these operations. As for the encoded base operations the measurements were done on a machine that has two Intel Xeon processors with in total 8 cores and runs a 64-Bit Fedora 10. The measurements were not parallelized and during the experiments as few other load as possible was executed on the machine.

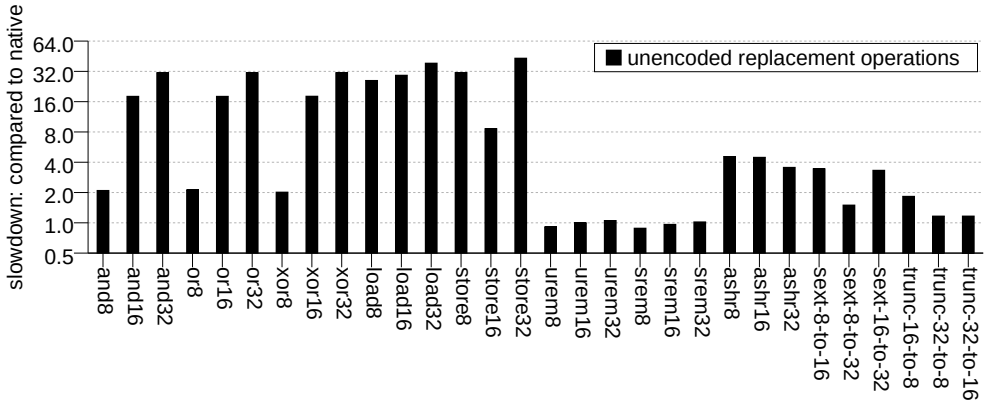


Figure 4.10.: Slowdowns of replacement operations compared to native (unchanged) versions. The following operations were measured each for different integer types (sized 8, 16, and 32 bit): logical bitwise operations (`and`, `or`, and `xor`), unaligned loads and stores (`load` and `store`), unsigned and signed modulo operation (`urem` and `srem`), arithmetic right shift `ashr`, signed upcasts (`sext-8-to-16`, `sext-8-to-32`, and `sext-16-to-32`), and downcasts (`trunc-16-to-8`, `trunc-32-to-8`, and `trunc-32-to-16`).

We see that especially the bitwise logical operations for larger integer types and the unaligned loads and stores are expensive in terms of additional runtime. Shift operations and casts are less expensive, and the modulo operations come virtually at no cost. Sometimes their replacement versions even seem to be faster. That surely is a measurement issue.

Figure 4.11 additionally depicts the slowdowns of the AN-encoded versions of some of our replacement operations. For most operations encoding dramatically worsens the situation, that is, encoding increases the slowdown tremendously.

Thus, we expect applications that make heavy use of the expensive replacement operations (bitwise logical operations and arithmetic right shifts) to be slowed down more than applications that mostly use operations that are encoded as base operation or whose replacement operation is less expensive.

4.2.3. Floating Point Operations

All kinds of AN-codes are only applicable to integers. Thus, we encode floating point operations by replacing them with encodable software implementations that make only use of integers. Currently, the user has to do this replacement by hand, for example, by using the SoftFloat library [sof09].

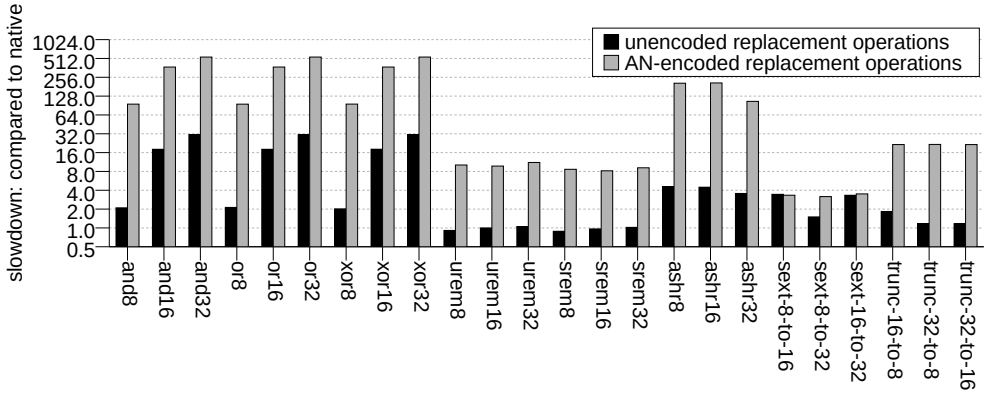


Figure 4.11.: Slowdowns of selective AN-encoded replacement operations compared to the slowdowns of their unencoded versions.

4.3. Encoded Constants

If encoding is done before execution as is the case for Compiler Encoded Processing (see Chapter 8), all constants can be encoded statically before execution. Non-integer constants are encoded according to our rules for memory, that is they are divided into 32-bit chunks. Therefore, we ensure that their size is a multiple of 32 bit.

If encoding is done at runtime as is the case for Software Encoded Processing (see Chapter 7), the constants are also encoded on-the-fly at runtime. They might be modified undetectably before they are encoded.

4.4. Calls to External Libraries

Encoding of binaries at runtime as it is implemented by Software Encoded Processing automatically protects calls to external libraries if they are statically linked into the binary executed. Therefore, their source code is not required.

In contrast, encoding at compile time does not allow for protection of external libraries whose source code is not available at compilation time. For calls to these libraries, we provide hand-coded decoding wrappers which decode parameters and after executing the unencoded original, encode the obtained results. For implementing these wrappers, we rely on the specifications of the external functions.

Thus, with Software Encoded Processing that encodes at runtime external libraries can also be protected. In contrast, Compiler Encoded Processing requires the source code of every library that shall also be protected. However, this is an accomplishable restriction for safety-critical systems. Furthermore, note that this applies only to the parts of the application that shall be protected,

that is, to the parts that process safety-critical data directly. For example, two safety-critical applications that communicate over a network can directly exchange encoded data. However, the network protocol stack needs not to be encoded. This does not decrease the safety, and reduces the overhead introduced by encoding. The safety is not decreased because the encoding of the data transfered makes errors in the transmission detectable. Note, however, that for detecting errors in the message order, encoded sequence numbers in the messages are required.

4.5. Encoded Data and Control Flow

With only an AN-code neither data nor control flow can be protected. Therefore, ANB- and ANBD-codes are required. How these are used depends on the encoding technique applied. Thus, we will introduce the encoding of data and control flow in the respective chapters:

- Chapter 6 for the Vital Coded Processor (VCP),
- Chapter 7 for Software Encoded Processing (SEP), and
- Chapter 8 for Compiler Encoded Processing (CEP).

All have in common that the signatures (the *"Bs"*) are used to detect errors in the data flow. Furthermore, they also use the signatures to encode the control flow either by encoding conditions of loops or if-statements (VCP) or by encoding conditions of conditional jumps (SEP and CEP). We explained the encoding of conditions already in Section 4.2.1.

Furthermore, all three encoding techniques use an implementation of the version D to detect outdated data with a high probability. Outdated data is a special data flow error – a lost update. Detection of lost updates is required for addresses, variables, or registers that might be written several times because they are written within a loop. If one of these writes is lost, this is not detectable by only having a signature that is attached to the address, variable, or register. Thus, a version D (see Section 3.5) is used to count updates of the address, variable, or register. While SEP applies a D to every data item processed, CEP applies it only to dynamically at runtime allocated memory and not to statically at compile time allocated memory. For VCP, it seems as if D is applied to every variable written within a loop. The description of VCP in [For89] is not clear about the usage of D.

4.6. Encoding Dynamic Memory Access

With *dynamic memory* we denote memory that may be dynamically allocated and is dynamically accessed. Of dynamic memory we do not always know at compile time how much memory will be required and how it will be accessed, that is, we do not know at compile time which addresses are accessed by load

and store instructions[4], which are used to read data from and to write data to memory. That is, for dynamic memory we have to protect previously unknown data flow from undetected errors such as lost updates, modifications of the stored data, or the read or written address.

Static and dynamic signatures

To support dynamic memory, we introduce *dynamic signatures* in [WF07b]. In contrast to the *static signatures* also known as *"Bs"* used so far, their value is not determined at encoding time but at runtime. While static signatures are assigned to data items whose access pattern we know at encoding time, dynamic signatures are used for data items for which we do not know at encoding time how they will be accessed at runtime.

An encoded value with a static signature is defined as $x_c = A * x + B_x$. The signature B_x is chosen (statically) at encoding time. An encoded value with a dynamic signature is defined as $x_c = A * x + h(addr, D)$ where $h(addr, D)$ is the dynamic signature. Both the address $addr$ that identifies the data item and the version D that counts the accesses to dynamic memory are not known at encoding time. The function $h(addr, D)$ maps these two values into one which is the dynamic signature. For CEP we use, for example, $h(addr, D) = addr + D$. In CEP, $addr + D$ is even allowed to be larger than A because values stored in memory are never directly decoded or used by encoded operations. Their signature is always previously adapted to a static signature smaller than A (see Section 8.3.3). The version D is incremented with every write operation executed. CEP also supports only ANBD- and ANB-encoding. For the latter, $D = 0$ is assumed. The consequence is that lost updates might remain undetected with a higher probability.

Support for dynamic memory

Forin in [For89] does not mention the problem of dynamic memory. Thus, we assume that VCP has no support for dynamic memory and uses solely static signatures. In Contrast to VCP, SEP and CEP both support dynamic memory. For SEP, all data access patterns are not known until runtime because the executed binaries are encoded at runtime. Thus, no static signatures are used at all for SEP. Every signature is dynamic. CEP allows to distinguish statically known data flow that happens within registers and statically unknown data flow that goes through dynamic memory at compile time. For the first, static signatures are used. For the second, dynamic signatures are used.

4.7. Version Management

Necessity of version management

For using dynamic signatures, we need to be able to determine the signature expected for a memory address to check if it is a valid code word. The naive solution is to use a a *global version counter* that counts updates of memory and to update all values with dynamic signatures after each write to one address to the new expected version. Only the currently written value would not require updating because it is updated by the write automatically. With the help of the

[4] We are assuming here the usage of explicit accessed to memory using load and store instructions. However, languages where also other instructions can access memory can be encoded similarly.

global version counter, we would always know that, no matter from which address we read, the value should have the version given by the global version counter. However, obviously updating every dynamically allocated and accessed data item after any write access to dynamic memory is inefficient. The alternative is to use a data structure that enables us to determine for an address the version that we expect for the value stored at this address.

Using an additional data structure to manage the versions of data items, the dynamic signature of a value needs only to be updated if the value is updated. Instead of updating all values stored in dynamic memory, the version management structure and the global version counter are updated. As measurements presented in Section 4.7.3 have shown, the latter requires much less time.

Whenever the version expected for a data item is required, it is retrieved using the version management data structure and the global version counter. For this purpose, all our version management data structures implement the function `getVersion(Addr addr)`. This function returns the value the global version counter had when *addr* was written the last time.

The version management data structures used have to be self-checking, that is, if for example an update to the structure is lost or written to a wrong location, the version information obtained from the structure has to be wrong. That means it must not match the version used to encode a value. Thereby, the error that influenced the execution becomes detectable. Thus, we decided to not use a simple table mapping addresses to versions because for this approach already two lost updates might be undetectable. If the update to the table and to the memory were lost, this would not be detectable because both, memory and table would be out-dated and match each other.

In the following we describe two possible data structures, a list- and a tree-based one, and compare their impact on the runtime. Both have in common that we provide one function for adding a just updated address and one function for retrieving the version of an address.

4.7.1. The List

For this approach, we use a linked list to store which version belongs to which address. Additionally a global version counter is used that counts how many updates to dynamic memory occurred. The list nodes additionally to the pointer to the next node contain two data fields for:

- an address that identifies an updated data item and
- the version difference between the data item identified by this node and the item identified by the next node.

At the beginning, the list is empty because no dynamic memory modifications took place yet. After the first instruction that updates dynamic memory, one element is added to the list. This element contains the address of the updated element and the version difference 1:

$$(addr_1, 1)$$

For each update to another address of the dynamic memory, another node is added at the beginning of the list containing the address that was updated and the version difference 1. The version difference 1 in each node means that the address identified by the next node has a version that is by 1 smaller than the version of the address identified by this node. Thus, if we want to determine

the version of an address *addr*, we walk through the list and sum up all version differences we encounter until finding *addr* in a node. By subtracting the sum obtained from the global version counter we determine the version the value stored at *addr* should have. Listing 4.17 demonstrates this. This, construction ensures that three matching lost updates are required for an undetectable lost update.

```
1  static uint64_t globalVersionCounter;
2
3  uint32_t getVersion(Addr addr){
4     int sumOfDiffs = 0;
5     Iterator iterator = list.iterator();
6     ListElement current = nil;
7     while(iterator.hasNext()){
8        current = iterator.next();
9        if(current.addr == addr){
10          found = true;
11          return globalVersionCounter - sumOfDiffs;
12       }else{
13          sumOfDiffs = sumOfDiffs + current.versionDiff;
14       }
15    }
16    return 0;  // addr was never written
17 }
```

Listing 4.17: Version retrieval using the list approach.

After n updates to dynamic memory where always another address was modified the list looks like this:

$$(addr_n, 1) \rightarrow (addr_{n-1}, 1) \rightarrow (addr_{n-2}, 1) \rightarrow ... \rightarrow (addr_1, 1)$$

If now an address is updated that was already written, that is, for which already a node exists in the list, we

- remove the node already existing for the written address,
- increment the version difference of the previous node, and
- add a new node with version difference 1 for this address at the beginning of the list.

For example, assume that in our example now the address $addr_{n-1}$ is changed again. The list will afterwards look like this:

$$(addr_{n-1}, 1) \rightarrow (addr_n, 2) \rightarrow (addr_{n-2}, 1) \rightarrow ... \rightarrow (addr_1, 1)$$

A new node was added for $addr_{n-1}$ at the beginning of the list. The old node was removed and the version difference of its predecessor was incremented. The version difference 2 stored now for $addr_n$ means that the version of the address identified by the next node is by 2 smaller than the version of $addr_n$.

Performance

We have to access the list for every access to dynamic memory. It does not matter if it is a read or a write access, each of the accesses has in worst case a complexity of $O(lengthOfList)$ because for list updating and version retrieval as well it has to be checked if the address is stored in the list. In the worst

case, the list will contain as many entries as the supported address space has addresses. However, a program does only modify a constrained set of data items. Thus, the list should never become that large and hopefully locality will lead to less overhead. For performance measurements see Section 4.7.3.

Undetectable errors

Versions reduce the risk of undetected lost updates. However, there exist still scenarios where lost updates can remain undetected. We determine the quality of a version management approach by the number of lost updates that are required to remain undetected. For the list approach these are three:

- the global version counter is not incremented,
- the list is not updated, and
- the updated data item is not saved to memory.

Checkpointing

To prevent an infinitely growing list, we introduce checkpointing. When a checkpoint is done all addresses of the dynamic memory are updated to the current global version counter and the list is cleared. If any error happens during this update procedure, this will result in the retrieval of wrong version information in the future and, thus, will lead to an invalid code word and error detection. Note that this is no false positive because there really was an execution error.

Thomas Knauth proposed in his *Großer Beleg* [Kna06] the usage of *linear addresses (LA)* to improve performance of the list approach. Instead of the addresses that directly point to a location in the address space, a so-called *linear address* is stored in the list. This address is determined dynamically on runtime by the order in which addresses are *written to*. An additional data structure is used to map location dependent addresses onto linear addresses. With a linear address it is much more straight forward to detect if an address was updated at all: Either there is a mapping to a linear address stored or not. Thus, the list has only to be accessed if such a mapping exists. Any errors within the required additional data structure will lead to the retrieval of erroneous version information and, thus, again are detected. Thus, the usage of linear addresses does not reduce the safety.

4.7.2. The Tree

The list approach results in a worst case complexity of $O(lengthOfList)$. Storing the required data in a tree will result in $O(\log(lengthOfList))$ complexity independent of any locality.

For implementing version management using a tree, we store absolute version numbers in contrast to the relative ones of the list approach. However, as explained before, we have to include a connection to the current global version counter. Otherwise, just loosing the update to the tree and the memory are sufficient for an undetectable lost update.

We propose the following algorithm that we first demonstrate on the changing tree data structure. A binary tree is used which is expanded as more and more

addresses are modified. The tree size depends directly on the amount of accessed addresses of dynamic memory.

Every modified address is mapped to a *linear address (LA)*. As explained in the previous section, linear addresses mirror the access order, that is, the first value written gets address 0, the next 1, and so on. In contrast to the *LA*s used for the list approach, for the tree, the once determined mappings from the real address to the *LA* remain static during the whole program execution. The LA is used to access the version information in the tree. Therefore, the LA's bit pattern is used as address information in the binary tree.

Directly after start-up the tree is empty. After modification of the first two addresses the tree will look like the tree given in Figure 4.12. Each of the leaves stores the version information of its LA. In Figure 4.12, the LA is denoted below the leaves. This address is also the address this information has in the tree. Therefore, the address is read from right to left. We start at the root node. If a zero is read, we descend to the left node. If a one is read, we descend to the right node. This addressing is finished when a leave is reached. In that case, the remaining unread address bits must be zero.

We determine that a parent node always contains the larger version number of its two leaves. This forms the connection to the current global version counter because the version contained in the root node has to be equal to the global version counter.

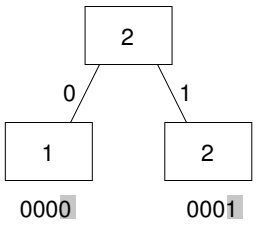


Figure 4.12.: Tree after using two data items. The bit patterns below the leaves denote the addresses for which this leave stores the version information.

Figure 4.13 shows the tree after modifying four different data items and Figure 4.14 shows the tree after updating address 0 again.

For the resulting tree, the following properties have to hold:

node consistency a parent's version number is always equal to the greater version number of its children and equal or greater than the smaller version number.

root consistency the root node's version number is always equal to the global current version counter.

If one of these properties is not fulfilled, an execution error must have occurred. Thus, every time the tree is accessed it has to be checked if these two properties hold.

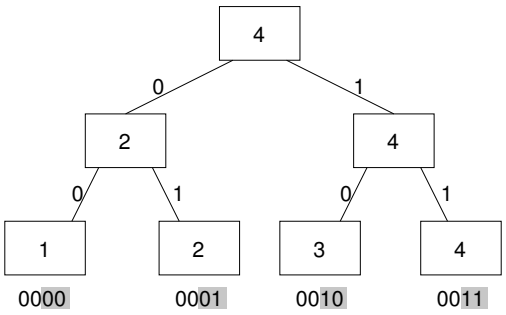


Figure 4.13.: Tree after using four data items.

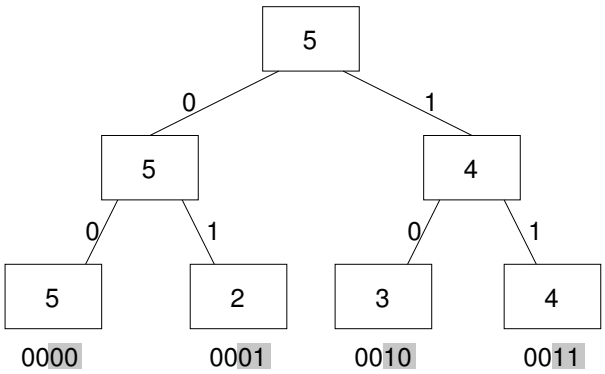


Figure 4.14.: Tree after updating the already existing address 0000.

Performance

An update of the tree always takes $O(\log(numberOfAddresses))$ steps. This is less then the worst case complexity of $O(lengthOfList)$ for the list approach. However, the average complexity for the list can be less, depending on the data locality of the executed program. Obtaining the expected version of an address also requires $O(\log(numberOfAddresses))$ steps.

The memory requirement of the tree is in worst case in the order of $2^{\log(numberOfAddresses)} - 1$, while the list requires only $lengthOfList$ entries.

Undetectable errors

For the tree, the following error patterns lead to undetectable lost updates:

- The following two lost updates and erroneous consistency checking:
  - the data item modified and
  - the tree are not updated, and
  - the procedure checking the consistency of the tree is executed in a faulty way so that it reports that the tree is consistent, although it is not.
- or the following three lost updates:
  - the data item modified,
  - the tree, and
  - the global version counter are not updated.

4.7.3. Performance Evaluation

All three approaches (list, list with LA, and tree) introduced in the previous sections and the naive version management that uses only a global version counter and updates all data items after each store were implemented within our SEP interpreter (see Chapter 7) by Thomas Knauth during his *Großer Beleg* [Kna06]. He also analyzed their performance impact. Therefore, he used different basic algorithms:

loop-x A program repeatedly accessing x elements of an array.

prime-x The computation of prime numbers up to x.

quicksort-x Sorting x randomly generated numbers.

md5-x Computing the MD5 hash of a string consisting of x characters.

matrix-x Multiplying two quadratic matrices of the size x.

Table 4.2 shows the results presented in [Kna06]. The column **unencoded** contains the runtimes for programs using an interpreter which executed binaries without any encoding. The column **naive** presents runs using the naive version update scheme that updates the versions of the complete dynamically accessed memory after each memory update. For obtaining the data presented in the columns **list**, **list with LA**, and **tree**, the respective scheme as presented in the previous sections were used.

program	unencoded	naive	list	list with LA	tree
loop-10	0.31	3.32	1.36	1.38	1.69
prime-5000	0.27	348.39	6.77	6.65	1.35
quicksort-1000	1.04	71.11	2.30	2.50	3.68
md5-10000	0.74	124.64	2.49	2.65	3.46
matrix-50	4.63	1819.54	19.74	22.93	19.83

Table 4.2.: Runtimes for different version management schemes in seconds.

For natively compiled runs, i. e., without any interpreter, no runtimes were measurable with a precision of $10^{-2}s$. Compared to that already the unencoded version of the interpreter induces large overhead. That is due to its simple and straight-forward implementation.

As was already expected, from the encoded variants the naive version generates the highest overhead. Every other version management scheme performs better. In most cases the list approach is the best. Only for the prime number computation the tree is faster. The reason might be that the computation of prime numbers in comparison to the other tested algorithms has not much locality which is essential for the performance of the list approach but not for the tree. Furthermore, the prime number computation handles by far the largest amount of data compared to the other benchmarks.

Figure 4.15 depicts the runtimes for executions of loop-x for different x. These measurements show that with decreasing locality the runtimes for all three

schemes increase. While the increase for the lists rises linearly, it increases in steps for the tree approach. The reason is that every time the tree has to be expanded the search depth increases.

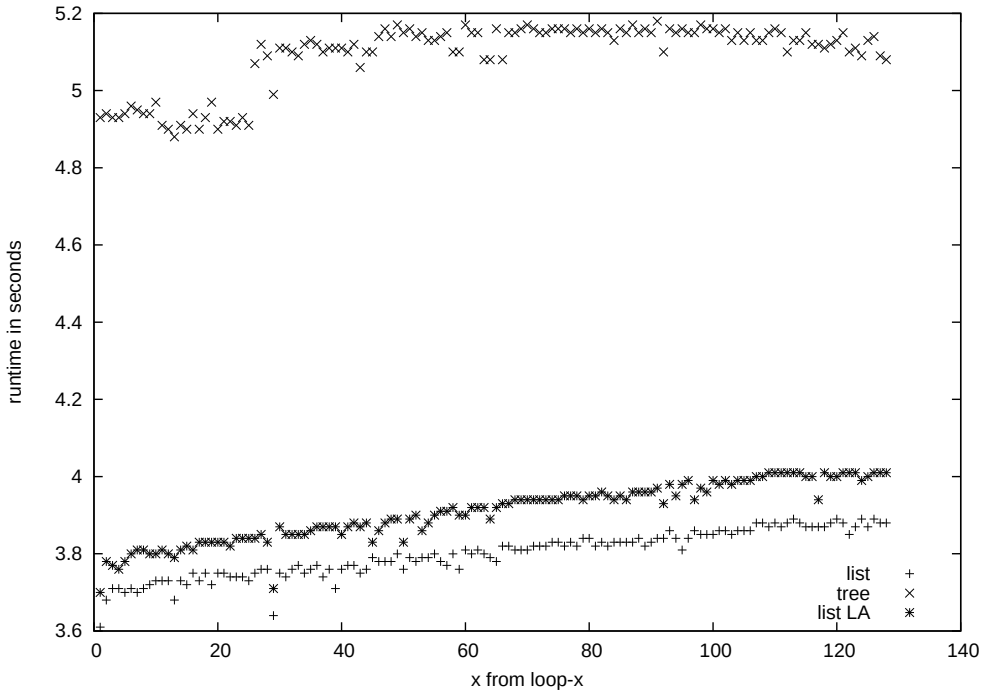


Figure 4.15.: Impact of locality on runtime [Kna06].

[Kna06] has also shown that checkpoints further decrease runtimes, but that it is difficult to determine an appropriate checkpoint size, i. e., a number of updates of the structure after which to do a checkpoint. The measurements presented in [Kna06] show that the optimal size depends on the executed program.

Future work

In the future, the sensitivity of the different version management schemes to errors should be evaluated. This evaluation especially should include the simple mapping of addresses to versions using a table. If its sensitivity to errors is not much worse than that of the more complicated structures, its usage could improve the runtime of encoded applications.

4.8. Outlook: Application of Encoded Basic Building Blocks

In this chapter we introduced and evaluated all basic building blocks that we will need to encode applications. However, these building blocks can be applied in different ways. For example, arithmetic encoding can be done on different

levels of abstraction in a computer architecture and on different points in the workflow used to produce an executable. These variants differ in

- the sphere of protection, that is, which parts of the workflow and the program execution are protected from unrecognized errors,
- the workflow required to obtain and execute a protected program,
- the generated runtime overhead,
- the principles used to protect the control flow from unrecognized errors, and
- the realization of lost update protection.

Levels of encoding

In [WM08a] we discussed the following possibilities to realize an encoded system:

- Encoding can be done on *source code level*. This type of encoding is done before compiling the program into its binary format.
- Encoding on *intermediate code level* is implemented by modifying the used compiler in a way that it generates appropriately encoded intermediate code which then is compiled into the required binary format.
- Encoding on *machine language level* is done at runtime by an encoded interpreter It can be applied to binaries without the requirement that the source code is available.
- Of course, arithmetic codes can be used to directly protect arithmetic operations in *hardware*. This should be by far the fastest approach, but development costs for such special hardware are high and their usage is not cost-effective.

Sphere of protection

The decision at which level of abstraction encoding is done influences not only the overhead produced but also the sphere of protection, that is, the part of the system that is protected from undetected errors:

source code level: In that case the provided protection is the highest possible when compared to the other encoding variants introduced above. Not only the execution on the CPU is protected, but also the transformation process producing the executable from an encoded version of the source code.

intermediate code level: Encoding at intermediate code level does protect the binary from the point of encoding on. That is not only the execution is protected but also the binary during storage. Any modifications to an encoded binary will result in broken codes at runtime with high probability.

machine language level: Encoding at machine level can only be done at runtime. The reason is that encoding requires complete disassembling. This is not possible for most binary formats. Nevertheless, the protection can be extended onto the binary itself by providing an encoded hashsum of the binary. This hashsum can be used to check the consistency of the dynamically encoded process image at program start. This check can be implemented encoded.

hardware level: Encoding at hardware level has the smallest sphere of protection. Usually in hardware arithmetic codes without signatures are applied and, thus, only the execution of a single operation and the data storage in memory are protected by the code. The binary and control and data flow

are unprotected by hardware level approaches.

Hardware level encoding would surely be the fastest solution, but it has obvious disadvantages such as the small sphere of protection and the requirement of specially adapted hardware which will be very expensive and inflexible. An implementation for each of the three remaining approaches is discussed in the following chapters.

These different encoding schemes presented in the chapters 6, 7, and 8 can all be based on the same encoding, decoding, and code checking functions that we introduced in this chapter and can use the same encoded base and replacement operations that we also introduced in this chapter.

But different encoding schemes are different with respect to how encoding of control and data flow is realized. For these tasks we introduced general principles such as encoding of if-statements, static and dynamic signatures, and version management that are used by all encoding schemes.

5. Choice of Encoding Parameters

The encoding approaches described in this thesis use one of the following AN-codes to facilitate the detection of execution errors:

- AN-code,
- ANB-code, or
- ANBD-code.

We introduced these codes in Chapter 3.

All these codes are parameterized. When using these codes we have to choose the following parameters depending on the code used:

- the code parameter A,
- signatures (Bs), and
- restrictions for the version D.

This chapter discusses how these parameters should be chosen. First, we will discuss the choice of A. The results apply to all three codes. Second, the choice of the signatures is discussed. This only applies to the ANB- and the ANBD-codes. Last, we discuss necessary restrictions for the version D used by ANBD-codes.

5.1. Choice of A

To ensure that encoding does not destroy the information contained in the functional values encoded, A has to be an integer that is

- larger than zero and
- smaller than 2^{m-n} if the functional values are n-bit and the code words are m-bit values.

Another question is if A can be chosen in a way that maximizes the probability of detecting execution errors. This section shows that indeed different values for A result in different error detection capabilities of the AN-codes. This section provides some general rules for choosing a good A. However, it cannot provide *the* perfect A. We will also explain why this is not possible.

First, we provide a general theoretic discussion how the choice of A influences the probability of detecting errors. Second, we experimentally evaluate different As with respect to their probability that random modifications of code words result in undetectable errors.

5.1.1. How A Influences the Probability of Detecting Errors

No power of two

First, A should not be a power of two. If A is a power of two, multiplication with A is equivalent to a left shift of the functional value. The resulting code words have a minimal Hamming distance of one because the redundant bits introduced by the multiplication with A are zero for all code words. Thus, errors that change only the higher order bits, that is, all bits that were not introduced by the left shift, are undetectable because their result still will be the expected power of two as long as the least significant bits added by the shift remain untouched. Only errors that change the redundant bits (the zeros introduced by the multiplication with an A that is a power of two) are detectable. However, the larger the power of two used is the smaller is the probability of undetected errors because the larger the power of two is the more of the least significant bits have to be zero and cannot be changed undetectably.

One could assume that we should also avoid As whose factorization contains any powers of two because any two that is part of A's factorization introduces one fixed zero in the least significant bits of the code words. However, we cannot apply the same argumentation as used for powers of two to As whose factorization contains twos. For example, for an A whose factorization contains one two, the following applies:

- The last bit of a code word always has to be zero and cannot be changed undetectably by an error.
- The higher order bits can also not be changed undetectably because they have to be the product of the functional value and the remaining factorization of A without the factor that is two ($\frac{A}{2}$).

How these probabilities for error detection/non-detection can be combined is not easy to determine. Therefore, we will in Section 5.1.2 also analyze As whose factorization contains twos.

The larger, the better

Also the size of A influences the probability of detecting errors. Code words consist of n *functional bits* and k *redundant bits*. The number of redundant bits is determined by the size that A has in its binary representation: $k = \log_2(A) + 1$. Obviously, k is the larger the larger A is because the logarithm function is strictly monotonically growing.

If we assume that errors lead to a uniformly distributed random modification, such a modification will result in another valid code word, that is, in an undetected error, with the probability:

$$p_{undetected} = \frac{\text{number of valid code words-1}}{\text{number of possible words}} = \frac{2^n - 1}{2^{n+k}} \approx \frac{1}{2^k} \tag{5.1}$$

The larger A and thus its size k is, the smaller is the probability that errors remain undetected. The probability $p_{undetected}$ decreases exponentially with the number of redundant bits k.

We experimentally verified Equation 5.1 by injecting random uniformly distributed errors into random encoded numbers and testing if the injected errors

destroyed the code and, thus, could be recognized. Therefore, we used 32-bit functional values and 64-bit variables for representing encoded values because these are the sizes of functional values and code words that we use in our encoding approaches presented in chapters 7 and 8. The functional values and encoding parameters are randomly chosen within the given boundaries for their size. Errors were injected by randomly choosing a 64-bit number and using it for a bitwise xor with the randomly chosen encoded number. For all random choices, we use a uniform distribution. Figure 5.1 depicts the results, which confirm Equation 5.1

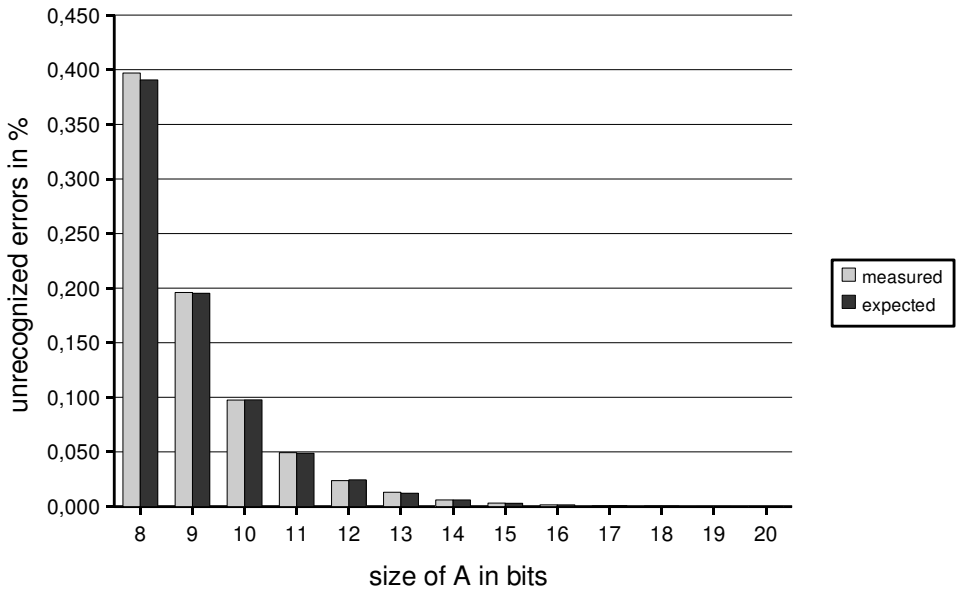


Figure 5.1.: *Measured* error detection probability vs probability *expected* by Equation 5.1.

The probability $p_{undetected}$ as defined in Equation 5.1 determines how probable it is that a uniformly distributed random modification of a code word results in another valid code word. But at least for erroneous modifications of data during storage it is known that the errors are not following a uniform distribution. For these errors, one flipped bit is more probable than multiple bit errors [LSHC07]. However, we were not able to obtain any reliable data about error patterns occurring during data storage, transport, or computation and the probabilities of these errors for current hardware. If a more precise error model would be available, we could provide a more accurate hardware-dependent approximation for $p_{undetected}$ then provided by Equation 5.1.

Furthermore, Equation 5.1 assumes that the code words generated by A are equally distributed over the available bit patterns, that is, that the minimal Hamming distance between different code words is as large as possible. However, so far it is not easy to choose an A which will result in a specific predetermined

minimal Hamming distance between all code words. Nothing can be done but exhaustive testing of all possible error patterns, that is, of all modifications that might be applied to encoded data by an error. If a modification caused by an error changes a code word by the addition or subtraction of a multiple of A, the error is undetectable because the result of the modification still is a valid code word. Thus, if we know how many of the error patterns that are possible result in a modification that changes a code word by a multiple of A, we can also provide a better approximation for $p_{undetected}$. Connecting this information with a more detailed error model of the hardware used, would result in the most accurate approximation for $p_{undetected}$. In Section 5.1.2 we determine the number of error patterns that result in a modification that is a multiple of A for several different As. Therewith, we try to provide more rules for choosing a good A that provides a high probability for detecting errors.

Should A be prime?

Forin [For89] chooses A to be a prime and motivates this choice with a maximized error detection probability for multiplications without any further clarifying explanations or providing an implementation of an encoded multiplication.

For our encoded operations, which we introduce in Chapter 4, it is not required that A is a prime from the point of view of correctness, that is, that the operations provide the correct and expected result in an error-free execution.

However, our opinion is that A should have as few factors as possible to reduce the probability of undetected operation errors, which might influence the execution in a more systematic manner than the other error symptoms described in Section 2.5. This is just a heuristic which could only be analyzed for a specific hardware architecture. However, to be on the safe side a large prime A should be used.

5.1.2. Practical Evaluation: How Many Errors Are Undetectable?

The experiments presented in Figure 5.1 assume that the errors that modify code words follow a uniform random distribution. That means that, for example, the modification of one bit occurs with the same probability as the modification of all bits that form the code word. However, at least for memory errors, mostly only a small number of bits are modified [LSHC07]. Thus, As should be preferred that ensure especially the detection of errors that modify only a few bits. In the following, we will analyze the error detection capabilities of different As for different numbers of bits that are flipped due to an error.

Note that here we provide no discussion of errors occurring in logical circuits, for example, in the arithmetic logical unit (ALU) or circuits realizing memory addressing. The reason is that these are hardware-dependent and no error models are available to us. For reducing the probability of undetected errors occurring in logical circuits, we recommend, as we explain in the previous section, the usage of As that are prime.

Error

In the following, an *error* e is the difference between the unmodified, error-free code word a_c and its erroneous, modified version a'_c. An error e is not detectable by an AN-, ANB-, or ANBD-code if the error e is a multiple of A because in that case $(a_c \mod A) \equiv (a'_c \mod A)$ holds. Thus, an A should be chosen for which

As few errors as possible should be multiples of A

only a small amount of the errors which can be formed by bitflips are multiples of A. The less errors are multiples of A the smaller is the probability of an undetectable error and, thus, failure of an error detection that uses AN-codes.

We know of no direct mapping from A to the probability that an error is a multiple of A. Thus, we try out each possible error and check if it is a multiple of A. However, this brute force computation of the probability that for a specific A an error is a multiple of this A is very computation intensive. Therefore, we compute this probability for a specific number of bits that are flipped. Thereby, we start with only one flipped bit. Next, we compute the probability for two flipped bits and so on. This approach allows us to compute the probability that an error is a multiple of A for the most probable error patterns, that is, these patterns that modify only a few bits.

To compute the probability that an error for a specific number of bits flipped is a multiple of A, we check each combination in that these bitflips can modify a code word. These combinations are formed by variations of

- on which positions in the code word the bits are flipped and
- into which direction ($0 \rightarrow 1$ or $1 \rightarrow 0$) the bits are flipped.

For each possible combination, we compute the resulting error e and check if e is a multiple of A. Furthermore, we take into account onto how many code words the error e analyzed can be applied. For example, for an error e that flips the highest order bit from 0 to 1, we consider only these code words whose highest order bit is 0.

In the following, we present the computation of the probability that a random error is undetectable, i. e., is a multiple of A. We do this analysis for several different As for an ANB-code and ANBD-code. For our computations, we are assuming code words that have a size of 64 bits and 32-bit functional values because these are the sizes of functional values and code words that we use in our encoding approaches presented in chapters 7 and 8.

If code words have a length of m bits, we have to consider all words that are representable with these m bits as possible target of an error. The reason is that for an ANB-code (and for an ANBD-code) depending on the choice of the signature B (and the version D) every word representable with these m bits can be a code word as long as the signature B (the sum of signature B and version D) can be chosen from the interval $[0, A[$. If the signatures are further restricted, the set of possible code words will be reduced. A special case of this restriction is the AN-code where the signatures are restricted to 0. For an AN-code, only multiples of A have to be considered for computing the error probabilities.

Figures 5.2 to 5.5 depict the results of our probability evaluations for ANB- and ANBD-codes for different As. We calculated the probabilities for 1 to 7 flipped bits. Due to the exponential growth of the problem size, we were not able to calculate further probabilities in reasonable time.

Powers of two

Figure 5.2 shows the results for As that are powers of two that add 28 to 31 bits of redundancy to the functional part of the code word. For all four different As in Figure 5.2, the probability for an undetected error decreases with the number of

bits that are flipped by the error. Furthermore, as we predicted in Section 5.1.1, the larger the A is the smaller is the probability for an undetectable error. This holds for all numbers of flipped bits that we analyzed.

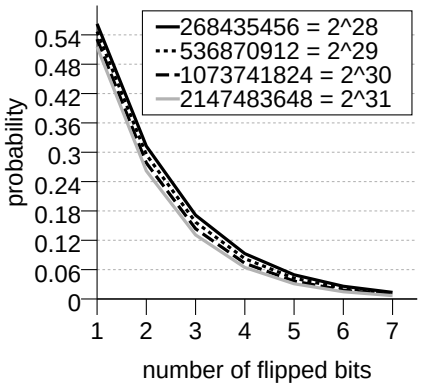
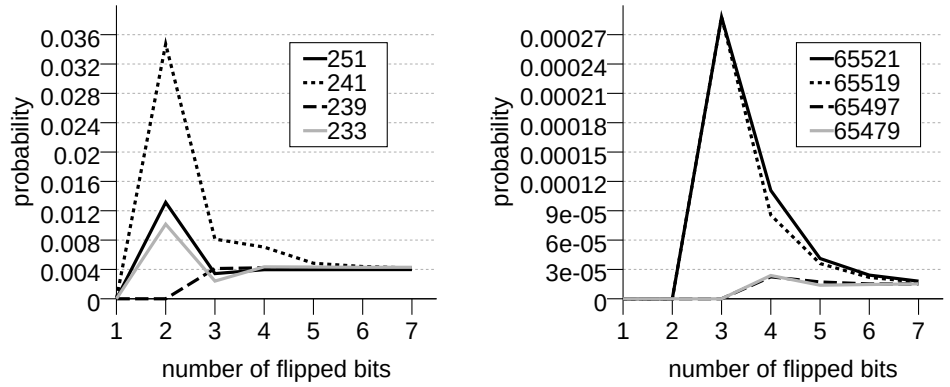


Figure 5.2.: Probability of undetectable errors for As that are powers of two.

Small primes

Figure 5.3 depicts the error probabilities for As that are prime numbers requiring either 8 or 16 bits for storage. The 8- and 16-bit prime numbers as well are much smaller than the powers of two whose results we present in Figure 5.2. However, their probabilities for undetectable errors are by several magnitudes smaller than that of the powers of two in Figure 5.2. When we compare the 8-bit with the 16-bit As, the general rule *the larger the A the less errors are undetectable* applies. However, the variations between different As – especially for lower numbers of flipped bits – are considerable with one order of magnitude.



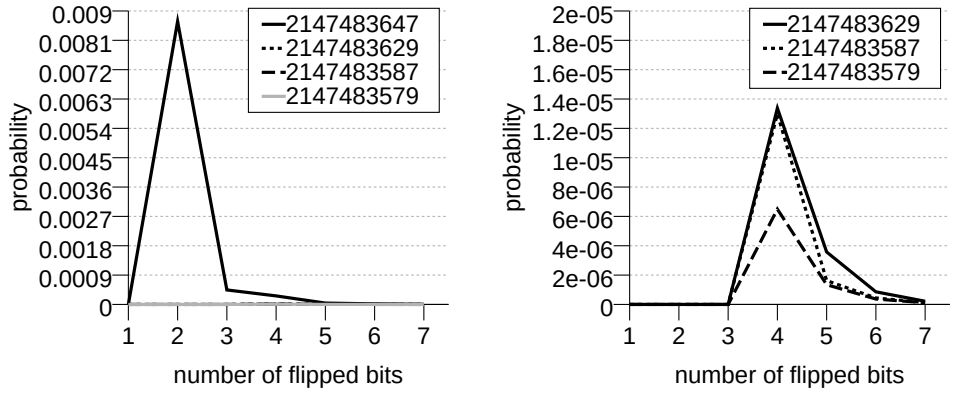
(a) As that require 8 bits for storage. (b) As that require 16 bits for storage.

Figure 5.3.: Probability of undetectable errors for As that are *small* primes.

Large primes

Figure 5.4 presents our results for As that are also primes and add 31 bits of redundancy. Figure 5.4(a) shows the result of four different As. However, $A = 2147483647$ shows especially high probabilities for undetected errors for only a small amount of flipped bits. For this reason, we depict in Figure 5.4(b) only the three best As of Figure 5.4(a). These three As show very similar

characteristics. However, there are also variations by one order of magnitude. Last, note that when comparing the results presented in the figures 5.3 and 5.4, again the rule *the larger the A the less errors are undetectable* applies under the condition that especially poorly performing As are not considered.



(a) All 31-bit primes.

(b) 31-bit primes without worst performing A of Figure 5.4(a).

Figure 5.4.: Probability of undetectable errors for As that are *large* primes requiring 31 bits for storage.

Non-primes

Last, we evaluated several 31- to 32-bit As that are not prime but can be factorized, that is, represented as a product of primes. For the different As evaluated we increased the amount of twos that are part of the factorization of the A. The results we present in Figure 5.5. We cannot see that the number of twos contained in A's factorization influences the error detection probability. However, the As analyzed are very large and the differences might be very small. On the other hand, we were surprised to observe that all four non-prime As whose results are presented in Figure 5.5 perform by several orders of magnitude better than the similarly sized As whose results are presented in Figure 5.4. Furthermore, it stands out that for all four non-prime As 1 to 5 flipped bits have a very low probability to produce an undetectable error. This is not the case for any of the previously analyzed As – even the similarly sized ones – that are prime numbers. Here only 1 to 3 flipped bits have a very low probability for undetectable errors.

Summary

To summarize, we have seen that for prime As and As that are powers of two the general rule *the larger the better* applies. As expected, As that are powers of two perform much worse than other As. However, there are large differences between different prime As. Furthermore, we were surprised to see that non-prime As seem to perform by several orders of magnitude better than prime As (at least for uniformly distributed bitflips). Of course, we researched only a small subset of all possible As. To our opinion, before choosing an A, such an analysis as presented in this section should be done to compare different candidates.

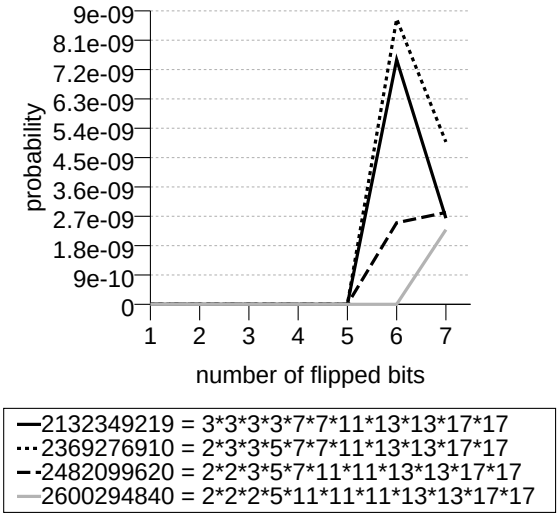


Figure 5.5.: Probability of undetectable errors for As that require 31 bits for storage and are not prime.

Future Work

In the future, the presented evaluation should be extended with

- the analysis of further – especially also non-prime – As and
- an analysis for AN-codes, which requires to restrict the set of possible code words to which the errors are applied. For AN-codes, only multiples of A are possible code words.

5.2. Choice of the Signatures

The assignment of the signatures B in an ANB- or ANBD-code influences the safety of encoded processing as well as the choice of A. Thus, we discuss rules for choosing signatures in this section.

$0 < B < A$

In general, the following equation should be fulfilled for every signature B: $0 < B < A$. If B were equal to A or larger than A, its addition would result in a different code word than intended. Note that B could be chosen to be zero. However, as we detail later for several encoded operations the parameters should not have a signature equal to zero. Furthermore, some intermediate results in the encoded multiplication are multiples of A (See Section 4.2.1). To prevent undetectable exchanges of variables with these multiples of A, we forbid signatures that are equal to zero.

Equal distribution

The signatures are used to detect exchanged operand and exchanged operator errors (see Section 3.4). Signatures should be equally distributed. This reduces the probability of undetected exchanged operand errors. If signatures are not equally distributed but some signatures are preferred, than it is more probable that operands with the same preferred signature exist and can be exchanged with each other.

On the probability of detecting an exchanged operator the choice of the signatures has no influence because encoded operators should be designed in a way that different operators used with the same signatures generate results with different signatures (See Section 4.2.1).

Adapt signatures at runtime instead of sophisticated choice

Signatures of encoded variables that were computed using other encoded variables depend on the signatures of the variables used for the computation. However, the signatures of the results of encoded operations should also obey the general restriction for signatures to be smaller than A and larger than 0. This is not always the case. For example, the signature of our encoded addition of the two encoded values x_c and y_c might be larger than A or also smaller than zero because its value is $2 * (B_x + B_y) - B_{OC}$ where B_x and B_y are the signatures of the two input values and B_{OC} is the signature of the overflow correction, which is the same for the whole program (see Section 4.2.1).

The signature of a computation result can be influenced by the choice of the signatures of the input values. However, if signatures of input values are chosen in such a way that the signatures of output values are valid, this can lead to unequally distributed signatures. For example, consider a program that contains mostly additions and uses a rather small value for B_{OC}, for example 1. In this case the signature of a computed result will often be larger than the ones of the input values. Especially, it might be larger or equal to A. To ensure that the signatures of the results are still smaller than A, the signatures of the input values will be chosen sufficiently small. Thus, smaller signatures become more probable and the signatures are not equally distributed anymore. Thereby, the probability of undetectable exchanged operand errors might increase.

Alternatively, signatures of values can be adapted during the program execution. If we, for example, want to change the Signature of $x_c = A * x + B1_x$ from $B1_x$ to $B2_x$, we have to add $B2_x - B1_x$ to x_c. Such adaptations allow us to choose the signatures for all variables randomly with a uniform distribution. For example, our Compiler Encoded Processing (see Chapter 8) assigns signatures randomly using a uniform distribution and adds adaptations to the encoded program whenever the signature of an input value would lead to an invalid signature of the result of an instruction.

Forbidden signatures depending on the operation

Apart from the above described restrictions for signatures of input values, most encoded operations have values for signatures that should not be used for the operands involved. For example:

addition & subtraction For additions and subtractions signatures should never be zero because the result would have the same signature as the operand with the non-zero signature. Thus, result and operand could be exchanged. This even applies when the result's signature is adapted because the exchange could happen before the adaptation is executed.

multiplication The signatures of operands of a multiplication should never be zero or one because the result's signature then would be either zero or the signature of the operand whose signature is unequal to one. Both scenarios might lead to undetectable exchanged operand errors.

division The signature of the divisor used in an encoded division should not be zero to prevent divisions by zero during the signature correction required. Furthermore, the signature of the divisor should not be one. Otherwise the result's signature would equal the signature of the dividend. This again could lead to undetectable exchanged operand errors.

Similar restrictions can be found for most encoded operations described in Section 4.2.1. These restriction are also much easier to enforce if signatures are assigned randomly and are adapted at runtime whenever required.

5.3. Version

For an ANBD-code, code words have an additional version D and are defined as $x_c = A * x + B_x + D$ whereby $0 < B_x + D < A$ has to hold. Otherwise, the addition of $B_x + D$ results in an invalid code word for the intended B_x and D.

However, from the point of safety, that is, the capability of detecting lost updates,

- each (single assignment) variable x should have a unique signature B_x and
- for each update operation a new D should be chosen.

Use a new D for each update

If a new D is used for each updated value, undetectable exchanges with other older code words that use the same D are the most improbable. Similarly, if for every variable ever written a unique signature is used, undetectable exchanged operand errors become the most improbable.

However, the size restrictions for $B_x + D$ prevent achieving these properties for normally sized programs with reasonable sized code words. Thus, instead of always using a new D as many different Ds as possible should be supported. The same applies to the signatures: As many different ones as possible should be used. However, the more different Ds are supported the less different signatures are supported and vice versa.

Handling size restrictions

The easiest way to ensure the size restrictions for the signatures and the version is to statically assign the available space to the signatures and the version. For example, half of the space can be used for signatures and half of the space for the version. If exchanged operand or operator errors are assumed to be more often than lost updates, more space could be used for signatures. On the other hand, if lost updates are assumed to occur more often, more space should be assigned to the version.

Note that the dynamic signatures as introduced in Section 4.6 combine the address-dependent signature $addr$ and the version v into one signature $h(addr, v)$. This dynamic signature is than used for encoding with an ANB-code. Thus, the resulting code word looks as follows: $x_c = A * x + h(addr_x, v)$ where $addr_x$ is the variable-dependent signature of x. Dynamic signatures should be uniformly distributed over an as large as possible set of possible signatures. Dynamic signatures are also restricted in their size: $0 < h(addr, v) < A$ has to hold. However, in principle, neither $addr$ nor v are restricted in their size.

Versioning in SEP and CEP

Our encoding approaches Software Encoded Processing (see Chapter 7) and Compiler Encoded Processing (see Chapter 8) use versioning only in connection with dynamic signatures. While in Software Encoded Processing every data value is versioned, in Compiler Encoded Processing only dynamically accessed memory is versioned. For registers, for which the data flow is statically predictable, no versioning is implemented. Since, we implement versioning only in connection with dynamic signatures, in principle no restrictions apply to the version used. However, our implementations use an 32-bit integer counter to determine the version information for updates. This counter wraps around after $2^{32} - 1$ increments. Then versions are reused that were already used $2^{32} - 1$ updates ago.

Note that also other approaches than a simple counter can be used to assign version information. For example, the versions could be chosen from a list of unique values. In the future, it should be researched if other algorithms for assigning values to D increase the safety achieved.

5.4. Conclusion

In this chapter, we discussed how the different parameters of an ANB- and ANBD-code influence the error detection capabilities of these codes.

For choosing A, the general rules that should be followed are:

- The larger the better.
- Do not use a power of two.
- Additionally check the probability for undetectable errors due to a few bitflips for the chosen A.

Note that we did not use an optimal A for our evaluations presented in the chapters 7 and 8 because the results presented in these chapters were obtained and published before the ones presented in this chapter.

For choosing the signatures, the rule is the less registers share a signature the lower is the probability of undetectable exchanged operand errors. Thus, signatures should be assigned randomly using a uniform distribution and the set of possible signatures should be as large as possible.

For assigning the version, the rule is a version should only repeat after as much versioned updates as possible. This reduces the probability of undetectable lost updates to a minimum. The version can be incremented with each update or chosen from a predetermined list of random values. More research is required to answer the question which of the two variants is better with respect to safety.

6. The Vital Coded Processor (VCP)

In this chapter we present the Vital Coded Processor (VCP) as it was described by Forin in [For89]. According to our knowledge, Forin was the first that introduced and used ANB- and ANBD-codes. Previous publications had only presented AN-codes. Furthermore, VCP is the first encoding approach that implements the arithmetic code to large parts in software instead of in hardware.

However, we observed several drawbacks of VCP that prevent its usage in modern computing systems. As we will detail in this chapter, VCP requires to know the complete data flow of a program statically at encoding time. This prevents the usage of dynamically accessed memory. Furthermore, the description of VCP in the available publications [For89, Dol06, Oze92] is rather vague and incomplete. Neither the encoding of operations is described in sufficient detail, nor how the encoding is applied to programs. It seems that large parts of the encoding – namely the encoding of control flow – have to be done by hand.

However, encoding using ANB- and ANBD-codes provides good error detection capabilities that are independent of the hardware used and that can be realized without the need for special hardware. Thus, we decided to improve VCP with the goal to facilitate its usage in modern systems.

In this chapter, we give an overview of VCP because we want to motivate why it is not usable in today's systems. Furthermore, VCP is the foundation for our own encoding approaches SEP (see Chapter 7) and CEP (see Chapter 8). Thus, knowing its foundations is essential for assessing SEP and CEP.

First, we will give an overview of VCP's structure and the workflow that is required to obtain an encoded program that can be executed by the VCP. Second, we will describe the specifics of encoding programs for the VCP. We conclude the chapter with a discussion of VCP's disadvantages and advantages. The disadvantages that we will describe led us to the implementation of our encoding schemes, first SEP and later CEP, which aim at removing these disadvantages and making encoding more generally applicable.

6.1. System Overview

The VCP applies ANBD-encoding as described in Section 3.5 on source code level to recognize transient and permanent errors disturbing program execution. Therefore, data as well as the program executed are modified. Programs executed by the VCP are encoded on source code level before compilation. The input data is encoded when it enters the system at runtime. The advantage of applying the encoding to source code before compilation is that this enables also the

detection of errors made by the compiler because these errors will destroy the encoding of data at runtime with high probability.

Program execution

The VCP executes programs in a loop and every program has the same structure. First, it reads in inputs that are encoded by a specific *encoder* hardware. These inputs are then processed by the encoded program executed by a CPU. In the end, encoded outputs are produced. These are checked and decoded by a specific code *checker and decoder* hardware. If the code is valid, output is allowed to leave the VCP and the next iteration, i. e., execution, of the encoded program starts. Otherwise, the output is prevented and for this iteration no output is available. Instead the output pins are set according to Forin to a safe state. After such a detected error, the next iteration is started in the hope that the error is transient. In principle, the ANBD-code used can also detect permanent hardware failures. However, Forin does not state how these are handled.

Signature handling

To every input a signature is assigned at encoding time, i. e., at compile time. The signatures of the output values are computed at encoding time. Therefore, the data flow implemented by the program is analyzed. VCP does not use dynamic signatures as introduced by us in Section 4.6. Thus, no dynamic data flow is supported by VCP. The assignment of signatures to the input and output variables is determined at encoding time and is stored in a specific signature memory of the VCP at runtime. *Encoder* and *checker and decoder* have both access to this memory. The encoder uses the signatures assigned to the input values to encode the latter. The *checker and decoder* use the signatures precomputed for the output variables to check their code and to obtain their functional values for output.

Versioning

The version D of the ANBD-encoded variables contains the number of already executed iterations of the program, that is, it counts how often the program was executed and new input values were processed. Therefore, a hardware counter (the *clock*) is used to which *encoder* and *checker and decoder* also have access. For each iteration a new clock value D is used as a version in encoding the input values. The output values produced by this iteration are expected to have the same D. Thereby, the usage of outdated values becomes detectable. Furthermore, D is used to check the timeliness of the iteration executed. Therefore, the clock value is incremented when the iteration should have been finished and a new one should start. Thus, if output is generated too late, it will be encoded with the old D and the output values generated will not be valid code words.

Figure 6.1 summarizes the described structure of the VCP. In the example depicted, VCP executes the encoded version of the very small program $return\ z = x + y$.

System requirements

Some parts of the VCP architecture must function correctly, that is, they must not produce erroneous executions. This is the case for the code *checker and decoder* because if they are erroneous, they might permit erroneous output. Furthermore, the clock that determines D has to be correct because an erroneous clock might lead to undetected late outputs or undetected usage of outdated variables.

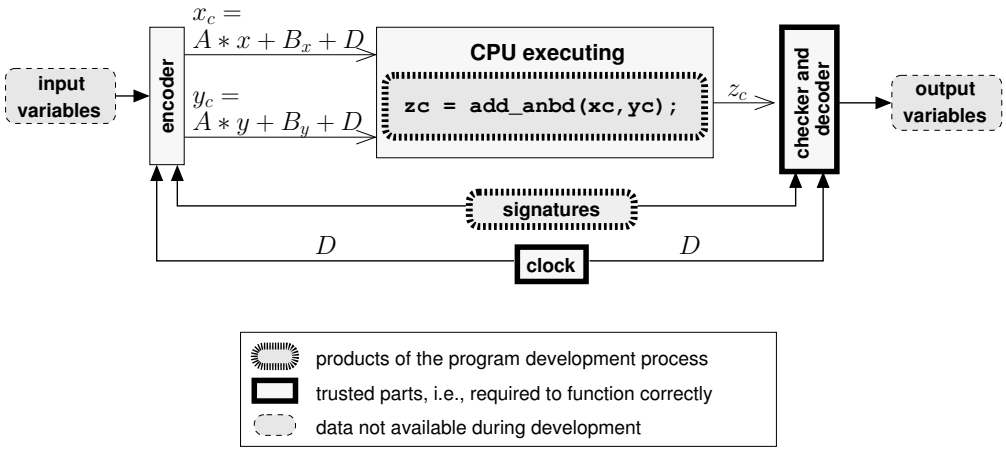


Figure 6.1.: How VCP executes an ANBD-encoded program.

6.2. Workflow

Different approaches to apply encoding are also different with respect to the workflow that is required for obtaining encoded programs. These differ with respect to the complexity of the process of encoding and the safety obtained. For the VCP, Figure 6.2 depicts the workflow required to produce an encoded program and the assignment of signatures to input and output variables. Forin in [For89] remains rather unclear with respect to the details of the workflow. However, according to [Oze92] some help of the programmer is required. How much of the encoding itself is automated in the VCP implementation described by Forin we do not know. We assume that at least the control flow is encoded by the programmer and that the programmer has to use special encoded data types and instructions, for example, a special encoded addition instead of the normal one. [Dol06] to some extent confirms this assumption. The resulting partially encoded source code is then processed by the *signature assigner* that assigns signatures and computes correctional values and the output signatures [Dol06]. The signature assigner includes the correctional values into the partially encoded program. Thereby, it makes the latter completely encoded. After compilation of the resulting encoded program, it has to be linked to the library of the *encoded base operations*. As a result an encoded binary is obtained that can be executed by the VCP.

We assume that the VCP is not only comprised by the encoding tool that encodes the source code, but also by the hardware infrastructure. At least for the encoder, the checker and decoder, the clock, and the memory for storing the signatures expected for the output values special hardware implementations are used.

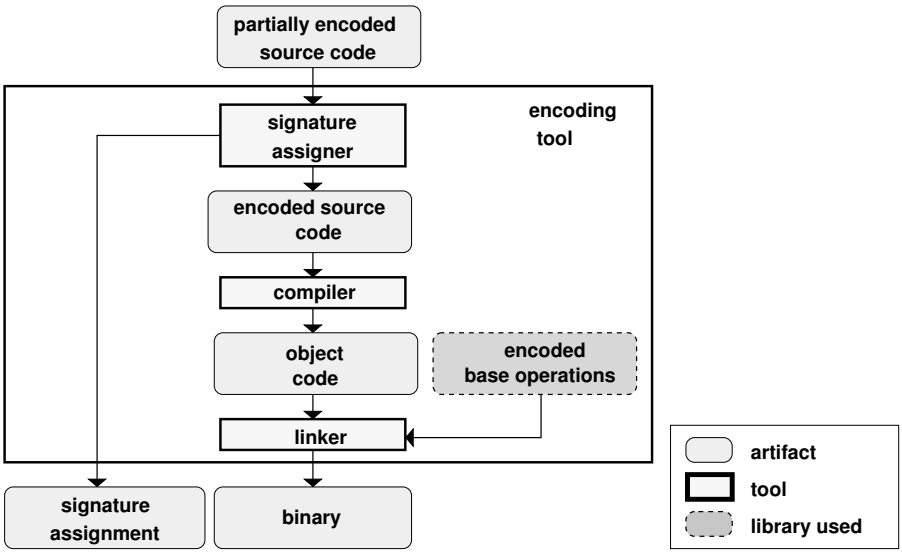


Figure 6.2.: VCP workflow that generates an encoded executable and the signature assignment to input and output variables.

6.3. Program Encoding

Basic instructions

All non-control-flow operations can be encoded as we described in Chapter 4. We do not know how Forin encoded these operations because [For89] only describes an encoded addition operation that does not support integer overflows. The programmer had to ensure that the program is free of overflows. Forin does not describe the implementation of any other operation. We suppose VCP does not support arbitrary C programs, but requires very specific features for the programs encoded, for example, freeness of overflows and no usage of dynamic memory.

Control flow

Encoded control structures such as branches or loops are implemented in a way such that the signatures of all variables whose values depend on the control structure are independent of the chosen branch or the number of iterations. However, if an error leads to an erroneous branch or to the wrong number of executed iterations the signatures will be destroyed with high probability. How Forin encodes if-statements, we introduced already in Section 4.2.1.

For loops, the branch instruction that decides about leaving the loop or not is encoded in the same way as an if-statement. For checking the number of executed iterations for each loop `x` another version D_x is introduced and added to the data items processed by the loop. With each loop iteration D_x is incremented by a fixed value Δ_{D_x}. After the loop $numberOfIterations * \Delta_{D_x}$ is subtracted from the variables to check that the correct number of iterations was executed. Note that with that solution the loop could be left out completely without the possibility to detect this. Forin neither states this problem nor does he provide a solution.

Furthermore, if several loops are nested, several D_x are added to the processed code words. It has to be made sure that this will not lead to overflows in the domain of encoded values because these overflows would destroy the code. Therefore, upper bounds for the loop versions or the sum of the loop versions are required. If the iteration counters would become larger than these upper bounds, they be set back to zero. If that happens for a loop, a bunch of iterations can be lost unnoticedly. Thus, restricting the D_x to a specific range reduces the safety because it might lead to undetected lost iterations. All these issues are not discussed by Forin and to the best of our knowledge also by no other publication.

Unsupported features

Forin's description of the VCP in [For89] is rather general and incomplete. Forin claims that the VCP supports the following operations: addition, subtraction, multiplication, truncation, and the boolean operations `and`, `or`, `xor`, and `not`. Note that Forin does not describe or mention an encoded division, which to our opinion is required for implementing the encoded modulo operation that is required to implement the trunc operation he claims to support.

Furthermore, apart from the addition Forin does not describe any implementation. Thus, we developed our own encoded operations (see Chapter 4). In contrast to Forin's addition, ours supports integer overflows. Due to the lack of information we cannot compare the remaining operations.

Of the control flow structures, Forin describes the if-statement in detail and the general idea for a loop. He does not discuss the problems that occur when these structures are nested in a program. Furthermore, Forin does not describe the encoding of dynamically accessed memory and internal and external function calls. We assume that the VCP does not support these.

Furthermore, Forin does not provide any evaluation of the VCP. Neither measurements for the error detection capabilities of the VCP nor for its runtime overhead are available.

6.4. Discussion of VCP

The VCP described in this chapter is a powerful tool for detection of execution errors because:

1. It uses of the powerful ANBD-code that enables the detection of all the error symptoms described in Section 2.5.
2. The VCP applies the encoding on source code level and, thus, facilitates also the detection of erroneous transformations by the following tools, for example, the compiler.

The usefulness of the VCP is also shown by the fact that it is in use in several railway systems, for example, in the trains of the metro in Paris and Lyon [For89].

However, as we have described in the previous sections, the VCP has major disadvantages that restrict its usability:

1. The complete data flow of the encoded program has to be known before the execution to be able to precompute the signatures of all output variables. That excludes the usage of dynamically allocated memory.
2. Special hardware is required to encode input variables, to store signatures, and to check the signatures of output variables.
3. To the best of our knowledge, encoding for the VCP is not completely automated. The user has to write programs using special encoded operations. It seems that the process of replacing normal operations with these special encoded ones and the assignment of signatures are automated [Dol06]. However, it seems that the encoding of control flow has to be done by hand. Encoding of nested control flow statements is a rather complex task because several version counters (*"Ds"*) and condition checks have to be considered. Thus, if done by hand, encoding control flow is error-prone and might have severe impact on the safety of the VCP.
4. The information available about the VCP is vague and incomplete. The implementation is not described in sufficient detail. Furthermore, no evaluation of the VCP is available, neither for its runtime overhead nor for its error detection capabilities. Thus, it is impossible to completely assess its capabilities and to use the approach.

After analyzing the VCP, we decided to provide our own encoding implementation. The objective of our work is to provide encoding for C programs. In contrast to the VCP, our solution shall be as usable as possible and require as few cooperation by the user, i. e., programmer, as possible. Thus, we first tried to encode binaries because then no cooperation at all is required. The resulting Software Encoded Processing (SEP) we describe in the next chapter.

7. Software Encoded Processing (SEP)

When developing Software Encoded Processing (SEP) our goal was to implement an ANBD-encoding without requiring any help from the developer of the software that is to be encoded. Thus, we wanted to implement encoding that requires no modifications to the program and as few modifications as possible to the workflow that produces an executable. For realizing this, we decided to encode binaries directly during execution, that is, on machine language level. We presented the SEP implementation in [WF07b].

In the following, we first give a general overview of SEP and describe the workflow required for developing binaries that can be executed using SEP. Next, we provide a detailed description of the encoding implemented by SEP. Thereby, we focus on the encoding of data and control flow because the other encoded operations we described already in Chapter 4. We conclude the chapter with our evaluation of SEP's error detection capabilities and the induced runtime overhead with respect to execution time.

7.1. System Overview

Encoded interpreter

The main idea of SEP is to use an interpreter to execute the binary that is to be encoded. This interpreter itself is encoded using the principles of the VCP [For89]. Therefore, every variable used by the *encoded interpreter* that is crucial to the correct execution of the binary is encoded. This includes

- the code executed,
- the data processed, and
- also data used by the interpreter to manage program execution, for example, the instruction pointer that points to the next instruction to execute.

The interpreter executes the binary in an encoded fashion. Thereby, it generates encoded outputs that can be checked by another (standard) hardware unit to determine if they are valid code words.

Signature handling

We do not know the binaries that will be executed by the encoded interpreter beforehand. Thus, it is not possible to precompute any signatures for the data processed before executing a binary because we do not know which data is processed when and how. Instead, all signatures for input are assigned at execution time, and the expected signatures of intermediate results and output values are precomputed at execution time. Therefore, the encoded interpreter uses for all encoded values dynamic signatures (see Section 4.6) of the form $h(addr, D) = addr + D$. $addr$ is a unique address that identifies the encoded value. It can identify a register and a memory address as well. D is the version

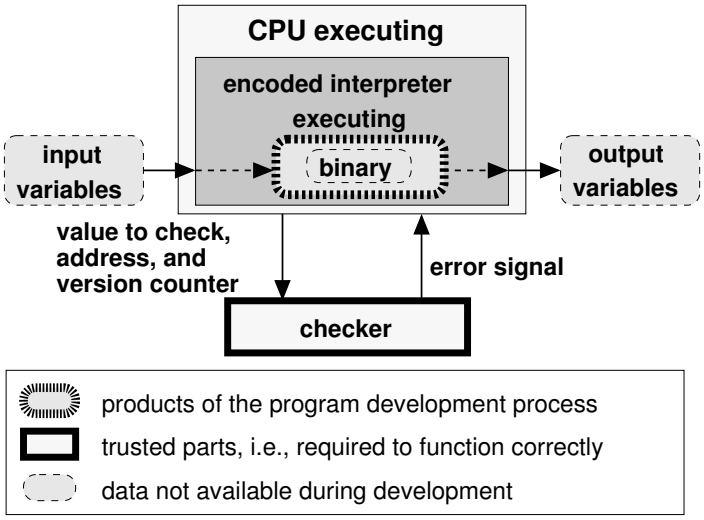


Figure 7.1.: How SEP is used to execute an unencoded binary.

counter that counts updates to dynamic memory. For SEP, all data with respect to the signatures is handled as if it is accessed dynamically. The reason is that the encoding is done at runtime and, thus, no data flows can be predicted before. After each instruction that might update data in the program's state, D is incremented. Otherwise, updates might be lost without the possibility to detect the loss.

Checker

The code *checker* checks that all output values are valid code words. Therefore, the encoded interpreter sends the encoded output values to the code checker before it decodes them for output. Furthermore, the interpreter sends the address $addr$ at which the checked value is stored and the current version counter D to the checker. Using this information, the checker checks if the value it received has the expected signature $h(addr, D)$. If that is not the case, an error is detected and the execution of the encoded interpreter should be terminated to prevent erroneous output.

The checker can be implemented as part of the encoded interpreter, or it can be realized as separate software that is executed by another hardware unit. The latter is the preferred, safer option because hardware errors having a common cause are less probable for different hardware units.

Figure 7.1 summarizes the described structure of SEP.

System requirements

As the VCP, SEP also requires some parts of the system to function correctly, that is, they must not produce erroneous executions. For SEP, this applies to the checker that checks the encoding of all output values and the decoding of the output values that is done by the encoded interpreter. We discuss the realization of the code checker more detailed in Section 7.3.6.

Sphere of protection

Note that the sphere of protection provided by SEP in comparison to the VCP is reduced. Errors introduced by the transformations producing a binary from given source code cannot be detected. Furthermore, additional measures have to

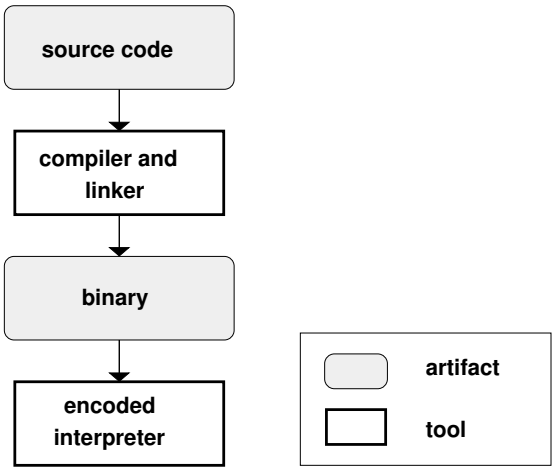


Figure 7.2.: Workflow required for producing and executing a binary that can be executed by the encoded interpreter.

be taken to protect the binary from unrecognized modifications during storage and during program loading and encoding.

7.2. Workflow

Figure 7.2 depicts the workflow required to generate a binary that can be executed using SEP. As depicted the binary is compiled and linked using tools – usually compiler and linker – that are not aware of the following encoded execution that is realized using the encoded interpreter. Thus, no changes to the original workflow are required.

However, the developer of a program still is restricted if SEP shall be used to detect execution errors. Currently, he cannot use floating point operations because these are not supported by our current SEP implementation. Their usage will lead to a runtime error at execution time because the interpreter encounters an unsupported operation. Other operations such as bitwise logical operations or unaligned loads and stores are supported, but they are executed unencodedly by our current proof-of-concept SEP interpreter.

Of course to support such an unmodified binary production process, the interpreter used has to support the binary format produced by the tools used. Our proof-of-concept implementation of the encoded interpreter executes DLX-binaries. DLX is an academic RISC instruction set developed by Hennessy and Patterson [PH90]. We used the DLX instruction set for our proof-of-concept implementation because of its manageable amount of operations that have to be encoded.

For compiling, we use the DLX compiler that is based on gcc. This compiler is provided by UC Santa Cruz for a student project [Mil]. Furthermore, [Mil]

provides also an interpreter for DLX binaries that we extended to support our encoding.

7.3. Program Encoding

In this section we describe the encoded interpreter that in SEP is used to execute binaries in an encoded fashion. This interpreter is encoded following the general principles presented by Forin in [For89] for the Vital Coded Processor (VCP). The SEP interpreter facilitates the encoded execution of unencoded DLX-binaries and thereby supports an unmodified binary production process.

When encoding the interpreter we have to pay attention to some critical combinations and occurrences of the error symptoms that we presented in Section 2.5. These combinations we will first briefly introduce. Second, we describe the encoding of the data processed by the interpreter. For a DLX binary, this data is composed of the executed binary itself, the data processed by this binary, and the instruction pointer that identifies the instruction that is executed next. Every other program state is part of the data processed. The DLX compiler makes, for example, stack handling explicit in the binary. Thus, the stack pointer is encoded automatically because it is part of the data processed by the binary. Furthermore, we show how during program start-up the encoded process image is constructed, that is, how the executed instructions and constants stored in the binary are safely encoded during loading the binary into memory. Afterwards, we present the body of the main loop of the encoded interpreter that executes the program. This is followed, by the description of the implementation of input and output functionality. Last, we describe the code checker implementation.

7.3.1. Critical Combinations of Error Symptoms

Some possible combinations of different error symptoms are critical for an execution protected by SEP. These we have to consider during the design of the encoded interpreter in addition to the symptoms described in Section 2.5. These critical combinations of symptoms are either a composition of the basic symptoms or a special occurrence of a symptom hitting, for example, a specific variable of the encoded interpreter. These special symptoms are:

Instruction loading error We have to check if the instruction that we loaded from memory is the correct instruction which we should have loaded using the current instruction pointer. A wrong instruction could be loaded because the address stored in the instruction pointer is modified or the instruction itself is modified in memory or during loading.

Note that erroneously executed load instructions that are part of the binary are detected because they are executed encodedly by the interpreter. However, the load of an instruction is a functionality realized implicitly by the interpreter. It is not part of the instructions forming the binary executed. Thus, it has to be protected separately.

Instruction execution error To execute an instruction we have to obtain the contained execution information such as which operation is to be executed with which operands and to which destination the result has to be stored. Obtaining this information is called *instruction decoding*. SEP has to detect if anything with this instruction decoding went wrong. This could be the case if any of the interpreter variables storing the decoded information is modified during execution, or if during instruction decoding an error occurs.

Erroneous instruction pointer The instruction pointer is crucial for the control flow because it contains the address of the instruction that is executed next. It is just a variable of the interpreter and, thus, it is susceptible to soft and permanent errors as every other variable. For this reason, we have to protect it from unrecognized erroneous modifications.

7.3.2. Encoding of the Process Image and the Instruction Pointer

The process image of an executed program consists of the data processed and the program code. For encoding both, the same ANB-code (see Section 3.4) with dynamic signatures (see Section 4.6) is used. In DLX all instructions are equally sized 32-bit words. Furthermore, all registers are 32 bits width. For encoding the memory, we divide it into 32-bit blocks.

Note that the usage of dynamic signatures enables us to detect lost updates despite using in principle an ANB-code. The reason is that the dynamic signature includes the version counter D, which is updated after each instruction executed.

We encode

- data *data* as $data_c = A * data + h(\&data_c, D)$ and
- an instruction *instr* as $instr_c = A * instr + h(\&instr_c, D)$

whereby

&x denotes the address of x that either identifies an address in memory or a register,

D is the version number that is incremented by the interpreter after each instruction executed, and

h(addr, D) is a function mapping its input to a number smaller than A. We use $h(addr, D) = (addr * D) \mod A$. The modulo with A ensures that the dynamic signature is never larger than A, which is a requirement for ANB-codes.

To facilitate the detection of instruction loading errors and an erroneous instruction pointer, we have to encode the instruction pointer IP as well. The IP is part of the interpreter and points to the next instruction to be executed. The instruction pointer IP is encoded differently than code and data: $IP_c = A * IP + h(IP, D)$. Its encoding enables us to detect instruction loading errors. IP has to equal the address $\&instr$ used to compute the dynamic signature of the encoded instruction that the current IP_c points to. Thus, $h(IP, D)$ the signature of IP_c and $h(\&instr_c, D)$ the signature of the instruction have to

be equal. How this is used to detect instruction loading errors we explain in Section 7.3.3 where we describe the main loop of the encoded interpreter.

However, before executing the binary, we have to ensure that the process image used for execution by the interpreter is encoded as described. The encoding of the process image can be achieved by either loading an already encoded process image from a *pre-encoded binary* into memory on program start-up, or by encoding the process image during start-up, i. e., during loading the process image from the binary into the memory. The first solution – usage of a pre-encoded binary – always ensures that modifications of the binary or the resulting encoded process image can be detected. The second solution – encoding during load – can be extended in a way that modifications of the binary are detectable. Note that if we want to be able to detect modifications of the binary, it is always required to extend or change the workflow used to generate a binary.

Pre-encoded binary To pre-encode a binary, we need to use specific addresses and a version number D to encode instructions and data contained in the binary. The version D can be initialized with zero. The program addresses can start on any value because the interpreter has to emulate memory anyway. Thus, it can put the program without problems at any address. The DLX-interpreter that we extended locates the text segment of the binary at address zero.

Impact of errors

If the pre-encoded binary or later the encoded process image is modified by a random error, for example, a bitflip, this destroys the code with high probability, that is, a modification of the process image results in an invalid code word. Such a modification could hit encoded data or encoded instructions. Both modifications will be detected if the encoded interpreter either uses the invalidly encoded data or executes the invalidly encoded instruction. Invalidly encoded data is detected because if encoded operations use invalidly encoded operands, the result will also be invalid with high probability. If such a result is externalized and, thus, its code is checked the error will be detected. Invalidly encoded instructions will be detected because the encoded interpreter is implemented in a way that the encoding of each instruction is checked during its execution.

Note further that if a pre-encoded binary is used, it is not required to digitally sign the encoded program unless it should be protected against malicious attackers. As explained above, random faults destroy the encoding of instructions or data with high probability and, thus, are detectable.

Required changes

For pre-encoding a binary, we have to consider the used binary format. We have to distinguish between process image and the binary itself. The first one is created from the binary by a program loader and resides in memory during execution. We have to encode the process image – not the binary that stores information required to execute a program such as the instructions to execute and initialization values for variables. Hence, for supporting pre-encoding, it is required to change the binary format and the program loader used.

Encoding during load If we want to support the detection of modifications

Required changes

of the binary loaded or modifications occurring during the load, we need to

know the hash value that the process image generated by loading the binary will have directly after its generation. This hash value has to be encoded using a previously known signature. Therefore, the workflow for producing a binary has to be extended with the computation of this expected hash value.

If such an encoded, expected hash value is available, the following procedure for constructing and checking the encoded process image is sufficient:

1. We load the program into memory and encode it during loading using the appropriate addresses and $D = 0$ for computing the dynamic signatures.
2. We compute the hash value of the resulting encoded process image. Therefore, we use a hash value computation that is (hand-)encoded using the principles of VCP and our dynamic signatures. It uses the encoded process image as set of encoded input values.
3. Last, the computed and the expected hash value are compared. If both are equal, the program can be executed. If they are not equal, the execution has to be aborted. This comparison should also be executed by the code checker that is required to be safe.

Impact of errors

The encoded computation of the hash of the encoded process image and its encoded comparison with the hash that is expected ensures that any modifications of the binary before and during loading will be detected. Later, during the execution, the process image is protected from undetectable modifications by the encoding of the instructions that influences each result produced.

7.3.3. Encoded Program Execution

After we introduced the encoding of the process image, we will now describe how the encoded interpreter executes the program represented by the encoded process image. Therefore, Listing 7.1 describes the body of the interpreter main loop that loads one instruction from the encoded process image and executes this instruction. After executing the instruction, the version counter `D` is incremented, the signature management data structure is updated, and the instruction pointer `IP` is set to the next instruction to be executed.

Instruction loading

Line 3 of Listing 7.1 loads the instruction to be executed from memory. Therefore, the instruction pointer is decoded in the previous line. Lines 6 to 9 extract which operation (`OpCode`) is to be executed and the addresses of the operands (`O1_addr` and `O2_addr`) and the result (`Res_addr`). These addresses identify registers or memory locations. Immediate values are handled similarly. We extract this information from the AN-encoded instruction which we obtain by subtracting the signature from `Instruction_c`. Alternatively, for obtaining the information needed to execute the instruction, we could completely decode the encoded instruction `Instruction_c` by dividing it by A. Which option is the faster one depends on the platform used for execution. If on the platform used divisions are more expensive than subtractions, obtaining the execution information from the AN-encoded version of `Instruction_c` is less expensive with respect to execution time.

```
1  // decode IP and load instruction from memory
2  IP = IP_c/A;  // IP_c = A*IP + h(IP, D) with h(IP, D) < A
3  Instruction_c = loadFromMemory( IP );
4
5  // extract execution information from Instruction_c
6  OpCode = getOpcode( Instruction_c - h(IP, D) );
7  O1_addr = getOperand1Addr( Instruction_c - h(IP, D) );
8  O2_addr = getOperand2Addr( Instruction_c - h(IP, D) );
9  Res_addr = getResultAddr( Instruction_c - h(IP, D) );
10
11 // Did an error occur during loading the instruction?
12 // Compute check values C_load and C_ip for later usage.
13 // Note that C_load and C_ip are equal to zero
14 //   if no error modified Instruction_c and IP_c and
15 //   if the correct instruction matching IP was loaded.
16 C_load = Instruction_c % A - h(IP, D);
17 C_ip = IP_c % A - h(IP, D);
18
19 // execute instruction, adapt the signature of the result, and
20 // apply check values to the result.
21 switch( OpCode ){
22   case ADD:
23     // execute instruction and adapt result's signature
24     *Res_addr = add_anb(*O1_addr, h(O1_addr,D), *O2_addr, h(O2_addr,D));
25     *Res_addr += h(Res_addr, D+1);
26     *Res_addr -= ( h(O1_addr, D) + h(O2_addr, D) );
27
28     // check if
29     // ---------
30     // IP was unmodified and matching, unmodified instruction was loaded
31     *Res_addr += C_ip + C_load;
32     // correct instruction was executed
33     C_op=formOp(ADD, Res_addr, O1_addr, O2_addr);
34     *Res_addr += Instruction_c/A-C_op;
35
36     break;
37   case SUB:
38     *Res_addr = sub_anb(*O1_addr, h(O1_addr,D), *O2_addr, h(O2_addr,D));
39     *Res_addr += h(Res_addr, D+1)
40     *Res_addr -= ( h(O1_addr, D) - h(O2_addr, D) );
41     *Res_addr += C_ip + C_load;
42     C_op=formOp(SUB, Res_addr, O1_addr, O2_addr);
43     *Res_addr  += Instruction_c/A-C_op;
44
45     break;
46   case MULT:
47     ...
48 }
49
50 // increment version counter D, update all other signatures accordingly,
51 // and increment IP so that it points to the next instruction
52 D++;
53 incrementAllVersionsApartFrom( Res_addr );
54 incrementIP();
```

Listing 7.1: Body of the main loop of the encoded SEP-interpreter. It loads one instruction from the encoded process image, executes it, increments the version counter D, updates the signatures of all unchanged data values to the current D, and sets the IP to the next instruction to be executed. Note that the representation is simplified and does not represent the optimal implementation that, for example, uses a version management structure as described in Section 4.7 or summarizes functionality in functions.

Lines 16 and 17 calculate check values that are used to check

- if the instruction pointer `IP_c` is a valid code word and
- if the loaded instruction `Instruction_c` is a valid code word and if it matches the current instruction pointer.

If these conditions are fulfilled, the computed values `C_load` and `C_ip` are zero. Later both values are added in lines 31 and 41 to the result produced by the instruction executed. If the check values are not zero, that is, if an error modified `Instruction_c` or `IP_c` or the wrong instruction was loaded, this addition will destroy the encoding of the result.

Instruction execution

The following switch statement uses the `OpCode` to select the code implementing the instruction that has to be executed. Its selection is checked by the lines 33/34 and 42/43. If the correct branch was chosen `Instruction_c/A-C_op` should evaluate to zero. Otherwise, its addition to the result will destroy the code of the result.

For actually executing the encoded operations, the implementations described in Section 4.2 can be used. Note, however, that our current implementation of the SEP interpreter does not use the replacement operations. Instead, in SEP currently logical bitwise operations, shift operations and unaligned loads and stores are executed unencoded, i. e., unprotected.

The remaining errors that we defined in Section 2.5 (exchanged and modified operand, operation, operator and lost update errors) are handled in a similar way as in the VCP. Lines 24 and 38 use encoded numbers for the computations. Thus operation and modified operand errors are detectable. Lines 25/26 and 39/40 correct the signature of the result to match the signature that is expected for the result address. Therefore, first the expected signature for the result address and the incremented version counter `D` is added. Next, the signature that results from the expected signatures of the operands and the operation executed is subtracted from the result. Note that the computation of the expected signature is different for the different operations. For example, for addition and subtraction lines 26 and 40 are different. If an exchanged or modified operand, operator or lost update error had occurred, these correctional steps would destroy the code of the result because the dynamically computed expected signatures used for correction would not match the actual existing ones.

Version management

The version counter `D` is incremented in each iteration of the interpreter main loop (line 52). This enables us to detect lost updates because operands have to contain the expected `D`. Therefore, the version of each result computed is updated to $D + 1$ during the computation of this result (see lines 25 and 39). Of course then the versions of all other encoded values do not match the version counter `D`. Listing 7.1 uses the naive approach to handle this problem. It updates the version of all other encoded values to ensure that they match the current version information `D`. However, this is too expensive. Thus, we presented in Section 4.7 more efficient approaches, which are used by the proof-of-concept implementation of our encoded interpreter.

Incrementing the instruction pointer

Last, we increment the encoded instruction pointer stored in `IP_c` using a special encoded addition of an encoded one for moving the pointer to the next instruction. This addition also increases the version D of the encoded instruction pointer by one and adapts its signature to the new functional value of `IP_c`. As a result `IP_c` points to the next instruction to be executed and has the correct signature $h(IP, D)$. The encoding of the instruction pointer and the checks of its encoding that are part of each instruction execution ensure that errors during the incrementation or storage of `IP_c` are detectable.

7.3.4. Encoding of Control Flow Instructions

The DLX instruction set, which is executed by our encoded interpreter, contains jumps and conditional jump as instructions for managing the control flow of a program. These instructions modify the instruction pointer `IP`. For all these jump instructions we provide encoded versions. Encoded unconditional jumps use an encoded addition operation to apply the also encoded offset to the encoded instruction pointer `IP_c`, and encoded conditional jumps additionally use an encoded if-statement as described in Section 4.2.1 to implement the decision if a jump is executed or not. Furthermore, conditional and unconditional jumps as well update the signature of the instruction pointer `IP_c`.

The DLX instruction set used by the DLX compiler we use does not contain call instructions. It implements calls to internal function, that is, calls to functions whose source code is part of the program, using the available jump instructions. Calls to external functions defined in a library are not supported currently. Thus, no encoding of call instructions is required for our SEP proof-of-concept implementation.

7.3.5. Input and Output

Input and output functions – usually provided by system calls – are the boundaries of the sphere of protection provided by SEP. For both we implement wrapper functions. While the wrappers of input functions execute the original unencoded system call and encode the returned values, the wrappers of output functions decode the inputs and use these decoded inputs for calling the original unencoded system call. Before a value is decoded, the value, its address, and the current version information D is sent to the code checker for approval. We present more details about the checking of these values in the following Section 7.3.6.

The DLX interpreter that we extend supports system calls for

- opening, reading, and writing files,
- allocating and freeing memory,
- writing output to an output stream, and
- exiting the program.

We provide encoding and decoding wrappers for all those system calls.

7.3.6. Code Checking

Encoding alone will not result in the recognition of errors. Therefore, the code has to be checked.

When to check the code?

When and how often code checking is performed influences performance and the latency of error detection. The more often the encoding of instruction results is checked, the faster an erroneous execution is detected. However, code checking has to be done at least before data becomes externally visible. This generates the smallest overhead but results in the largest detection latency. We implemented the checking of externalized values only.

How is the code checked?

For checking the encoding of data, we send the encoded data item, the address of the data item, and its expected version to the checker. A code word in SEP is valid if the condition $x_c \mod A == h(\&x_c, D)$ holds where $\&x_c$ denotes the address where x_c is stored. The address $\&x_c$ either identifies a register or a memory location. D is the expected version that either equals the global version counter or is determined using a version management data structure as introduced in Section 4.7.

In addition to checking if a data value is a valid code word, the validity of the encoded instruction pointer IP_c has to be checked because control flow errors might only manifest in destroying the IP_c's signature. Thus, with every check of an encoded value also IP_c is checked.

Implementation of the code checker

In the VCP, code checking is done by a trusted hardware part. For SEP, code checking can be implemented using another (standard) hardware unit, e. g., an FPGA or a graphics card. Of course the standard hardware used might also be prone to execution errors such as transient or permanent hardware failures. However, the code checking itself is a very simple process that just requires

- the computation of the signature expected expected for this code word: $h(addr, D) = (addr * D) \mod A$,
- the computation of the signature contained in the value checked using a modulo operation, and
- finally the comparison of these two values and an appropriate reaction.

Thus, several methods to make this process safe, i. e., to ensure that execution errors influencing it are detected, are applicable. For example, redundancy schemes such as triple or double modular redundancy could be used or the checker could be even encoded. For all these approaches, in the end a small trusted part is required that is guaranteed to be executed error-free. For the redundancy schemes that is the voter required or for an encoded checker a simple code checker is necessary. However, note that for a false negative, that is, an undetected error, it is required that first an error in the execution happens and second a matching error in the code checker used occurs. In contrast to false negatives, false positives, that is, the detection of errors in a correct execution, from a safety point of view, are acceptable as long as their frequency is low enough.

Furthermore, the usage of several checker implementations in parallel is possible to further reduce the risk of not detecting an invalid code word. Each of the

checkers can independently interrupt the execution of the main CPU if it detects an erroneous execution. It is also possible that SEP generates encoded output that is used by another SEP interpreter or whose encoding is checked by its user.

Remember the actual decoding in our implementation is done by the encoded interpreter. Any function that externalizes data is replaced with a decoding wrapper. Before executing the externalization, this wrapper sends the values that are about to be externalized together with the current encoded instruction pointer IP_c and the data required for code checking to the checker. After approval, it decodes the values and outputs them, for example, by printing them on a display. The checker itself cannot perform the output because it has no access to the system resources required for performing the output. Thus, it is not possible to protect the output itself in our current SEP implementation. The sphere of protection for our current SEP implementation ends with sending the data to the code checker. However, if the safety requirements make it necessary, this can be changed in a way that ensures that the decoder produces the output as it is the case in the VCP.

Aliveness checking

For detecting if the interpreter has crashed or hangs, the code checker also has to implement a watchdog functionality. Therefore, the interpreter periodically sends an alive-signal to the checker to reset the watchdog. The timeout of the watchdog and, thus, the frequency required for the alive-signal depend on the application's requirements. For example, for realtime applications, the deadlines for the outputs of the application can be used. In this case, the alive signal can be combined with the checking of the application's outputs.

Note that this approach does not support applications that block by design. This behavior should be avoided in safety-critical systems.

Error handling

If an invalid code word is detected by the checker or the watchdog is not reset by the interpreter, the execution of the interpreter is stopped, that is, fail-stop is realized instead of arbitrarily erroneous output. Using this fail-stop behavior fault tolerance can be implemented for example by going back to a checkpoint or by recomputing invalid data. However, fault tolerance is not in the scope of this thesis.

Our current SEP implementation contains a software-based implementation of the code checker. Currently, this code checker runs on the same hardware as the encoded interpreter. This checker was excluded from the error injection that we used for evaluating the error detection capabilities of SEP.

7.4. Evaluation

We evaluated the error detection capabilities of SEP and the impact that SEP has on the runtime of applications. The following two section present our experiment setups and the results we obtained.

7.4.1. Error Detection Capabilities

The error injector

To test the capabilities of SEP to detect errors, we use our error injection tool FITgrind [WF06] that we will introduce in Chapter 9. FITgrind probabilistically injects errors of the following types:

- bitflips in memory,
- bitflips on results of operations, that is, bitflips in registers, and
- execution of different instructions.

The number of bits flipped is chosen according to an exponential probability distribution. Thus, mostly one bit is flipped and the probability of n bits flipping decreases exponentially with n. Note that FITgrind does not inject all symptoms defined in Section 2.5.

We compare the three following execution variants:

encoded interpreter The encoded interpreter was used for executing the binary. We used a prime A that added 16 bits of redundancy.

unencoded interpreter The unencoded version of our interpreter was used to execute the binary.

native The executed program was compiled to the native architecture and executed.

FITgrind executes an error-free *golden run* before the execution of the injection runs. The results of an injection run are compared with the results of this golden run. We checked if any erroneous output, i. e., output differing from the golden run, was generated, and if an error was recognized either by the OS or the interpreter.

Experiment setup

For the error injection experiments, we used `md5` that computes the MD5 hash of a string. We executed `md5` around 8000 times for each execution variant. In each of these runs errors were injected probabilistically. For each run, FITgrind was initialized with another random number and, thus, generated different error patterns. The results we classified as follows:

correct and complete The generated output does not differ from the golden run, despite the injected errors.

correct but incomplete The generated output was not complete, but it was a prefix of the output of the error-free golden run. In this case, the application crashed during execution or was stopped. For most applications, this crashed or stopped execution can be detected and, thus, we consider this outcome safe.

no output No output was generated. In this case, the application crashed or was stopped before generating any output.

incorrect output Output was generated that differed from the golden run. This includes not only runs were some parts or the whole output differed, but also runs which first generated completely correct output and afterwards appended additional output.

The first three output types are *safe*, i. e., no erroneous output is generated. But the last type – incorrect output – represents an *unsafe* behavior. If a system

generates this type of arbitrary, incorrect output, it is not fail-stop.

Injection results

Figure 7.3 shows the obtained results for our error injection experiments on `md5`. While the encoded interpreter produced no unsafe (incorrect output) at all, 4% and 9% respectively of the unencoded interpreted and the native injection runs produced incorrect output. On the other hand, 31% of all native runs and 15% of the unencoded interpreted runs produce complete and correct output despite the injected errors. For the encoded interpreter, only 0.06% of the runs under error injection produced completely correct results. That shows that the interpreter detects errors even before they can propagate and become visible. Thus, encoding can be used to detect hardware that becomes unreliable due to aging. Note, however, that the parts of the encoded interpreter that are not yet encoded such as bitwise logical operations or unaligned memory accesses were excluded from the error injections.

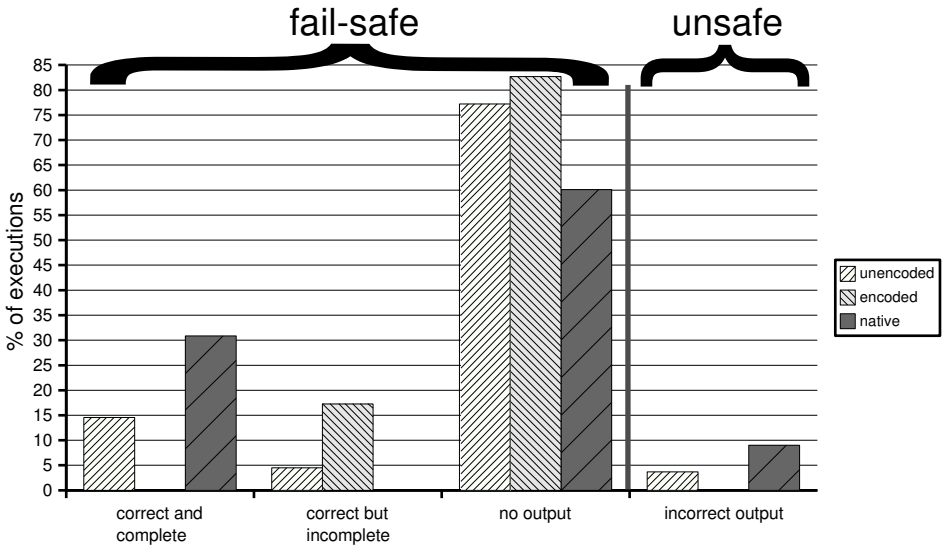


Figure 7.3.: Error injection results: Classification of output generated by runs that are subject to the injection of bitflips in memory and operation results and execution of different instructions.

Furthermore, we classified the error injection according to the kind of error detection. The results show which amount of detected errors is detected by the encoding itself and which amount is detected by other means. We distinguish the following kinds of error detection:

recognized by the operating system (OS) The operating system recognized an error and crashed the application. For example, a segmentation fault might have occurred.

killed by FITgrind The application ran much longer than the golden run and was killed by FITgrind. These runs might be caused by infinite loops caused by the injected errors.

no error recognized No error was recognized. Most of these runs either generated *correct and complete* or *incorrect output*

fail-safe inconsistency Here the interpreter detected an inconsistency, e. g., an unaligned jump.

fail-safe invalid code These are errors recognized by the encoded interpreter because an invalid code word was found.

Figure 7.4 presents the distribution of kinds of error detection that occurred in our injection experiments. Most of the errors are detected by the operating system. Only a small part is detected by the interpreter. Obviously, the structure of the interpreter adds redundancy that leads to additional error detections by the operating system: For the encoded and the unencoded interpreter rate of errors detected by the operating system is significantly higher than for the native execution. The interpreter itself does consistency checks apart from code checking. This leads to further detections even for the unencoded interpreter. There are no timeouts (killed by FITgrind) for the unencoded and encoded interpreter. Obviously, the code and consistency checks already prevent deadlocks and endless-loops for our benchmark.

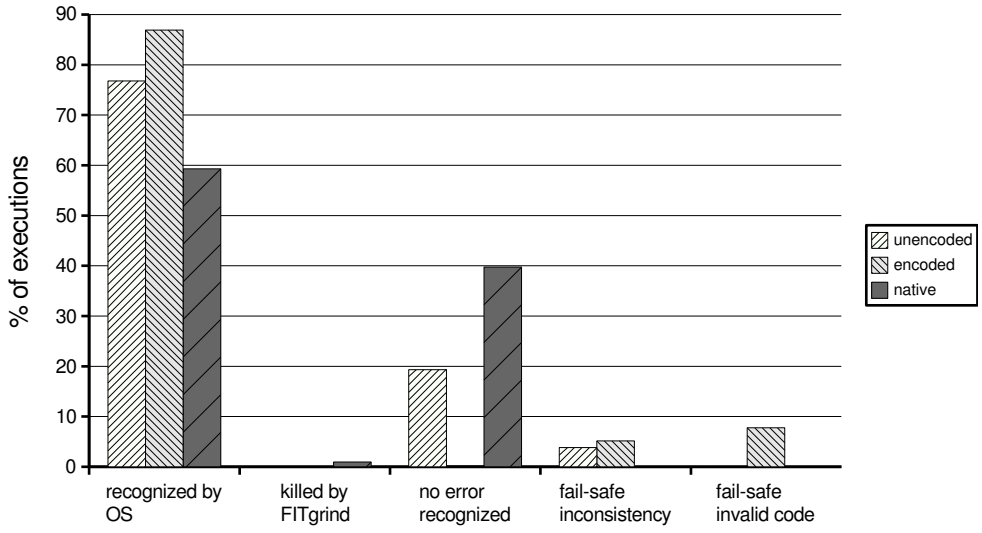


Figure 7.4.: Error injection results: detected kinds of errors.

7.4.2. Runtime Overhead

The results of our performance measurements are presented in Table 7.1. We measured the runtimes for the following set of programs which were chosen to have a broad set of different data usage patterns:

Experiment setup

`matrix-50`: Multiplies two matrices of the size of 50. This benchmark is computation intensive and uses a lot of multiplications whose encoding is

quite expensive in terms of runtime[1].

`prime-5000:` Computes the prime numbers up to 5,000 using the Sieve of Eratosthenes. This is less computation intensive, but has many memory accesses and input/output.

`md5-10000:` Computes the MD5 hash of a string that has a length of 10,000 characters. This represents a real world problem. It is a mixture of loops, branches and computations.
Note that `md5-10000` contains a large amount of bitwise logical operations, which are executed unencodedly in our SEP interpreter implementation. Thus, for a completely encoded version of `md5-10000`, higher slowdowns have to be expected.

`quicksort-1000:` A quicksort, sorting 1,000 numbers. Quicksort is the least computation intensive of our benchmarks.

The runtime measurements we performed on an AMD Athlon 64 running with a clock rate of 2200 MHz under SuSE Linux (2.4-based kernel). For measuring the runtime, GNU `time` was used. To measure the native executions thousands of runs were executed and the resulting span of time divided by the number of executions. The executions with the unencoded and the encoded interpreter were done in the same fashion. However, only five executions were used. The compilation was done with optimization (level O3).

program	**native**	**unencoded interpreter**	**encoded interpreter**
`matrix-50`	0.00125s	4.63s	19.74s
`prime-5000`	0.00029s	0.27s	6.77s
`md5-10000`	0.00001s	0.74s	2.49s
`quicksort-1000`	0.00112s	1.04s	2.30s

Table 7.1.: Runtime comparison [Kna06]

The native versions were compiled using gcc-3.3 and optimized for the Intel (CISC) architecture while the DLX-binaries were generated using the DLX-compiler, which is based on gcc-2.7 that does not optimize for the DLX (RISC) architecture.

Slowdowns observed

The slowdown generated by the interpretation (native vs. unencoded interpretation) ranges from around 900 times slower for `quicksort` and `prime` up to 74,000 times slower for the `md5` program.

The slowdown induced by the encoding (unencoded vs. encoded interpretation) ranges from around 2 times for `quicksort` up to 25 times for `prime`. Programs which are more computation intensive as `prime` or `matrix` result in a higher slowdown than less computational intensive ones such as `quicksort` that mostly compares and swaps.

[1] Note that the performance impact of multiplications for SEP is even worse than the measurements in Section 4.2.1 suggest because SEP still uses software implementations of the 128-bit operations.

We have not tuned the performance of SEP. Obviously, the interpretation itself adds much to the runtime overhead. This overhead could be reduced to a range that is similar to other machine level interpreters, e. g., by using just-in-time compilation. For example, the overhead factor of QEMU is in the range of 4 to 10 [Bel05].

7.5. Summary of SEP

To summarize, the advantage of SEP is that every binary can be executed in an encoded fashion without requiring its source code. Furthermore, any kind of control flow as well as dynamically allocated memory are supported. No explicit encoding for function calls – direct or via function pointer – for if-statements or loops is required. Thus, also nested control flow statements are easily encoded. On the other hand, the sphere of protection compared to VCP is reduced. Protection only starts when the process image either is encoded or an encoded signature of it is computed that later facilitates encoded checking for modifications. However, its greatest disadvantage is the overhead generated. Encoded operations are already significantly slower than unencoded ones. Interpretation and signature management further worsen performance.

Advantages

Disadvantages

To improve the performance and to extend the sphere of protection, we decided to develop an encoded compiler that is presented in Chapter 8. Binaries that were already encoded at compile time require no additional interpretation at runtime. Furthermore, their protection starts with the encoding at compile time and also facilitates the detection of errors introduced by processing tools applied to the program after encoding, for example, the linker.

However, to use an encoding compiler to provide detection of execution errors, the source code of the safety-critical applications has to be available. But we expect that the source code of critical components is anyhow required because when building a safety-critical system we have to be able to fix bugs or modify the behavior of the components used.

8. Compiler Encoded Processing (CEP)

In this chapter we will introduce our encoding compiler that we developed because neither the *Vital Coded Processor (VCP)* (introduced in Chapter 6) nor *Software Encoded Processing (SEP)* (introduced in Chapter 7) are practically usable for encoding applications that support dynamically allocated and accessed memory and data and control flow that cannot be predicted statically. While the VCP does not support dynamically allocated and accessed memory and data and control flow that cannot be predicted statically, SEP supports both but its performance impact is too high for usage in safety critical systems.

Disadvantages of VCP

Forin's VCP [For89] ANBD-encodes an application on source code level. As we point out in [WM08a] and in Section 6.4, VCP requires knowledge of the complete data and control flow of the encoded program to precompute the signatures of all output variables for code checking. This prohibits the usage of dynamically allocated memory. Furthermore, encoding loops and nested control flow structures at source code level is cumbersome and not described by Forin. We assume it has to be done by hand and is not automated. Thus, the VCP is only applicable to small applications. Forin presents neither an evaluation of the error detection capability of VCP nor any runtime measurements.

Disadvantages of SEP

Our first encoding solution, SEP, implements ANBD-encoding on assembler level at runtime. Therefore, we developed an interpreter for programs given as binary that itself is encoded using the principles of the VCP [For89]. Thus, we can encode arbitrary programs with arbitrary control flow. To encode dynamically allocated memory, we introduced the already mentioned *dynamic signatures* (see Section 4.6) that are determined at runtime. The error injection results presented in [WF07b] and Section 7.4 show that SEP successfully prevents erroneous output. However, the observed slowdowns make SEP unusable in practice and, thus, we stopped its development. Hence, SEP's current encoding remained incomplete because replacement operations as described in Section 4.2.2 were not yet used for SEP. Thus, for example, bitwise logical operations are executed unencodedly and were not targeted in the error injections presented in Section 7.4.1.

CEP vs VCP and SEP

In contrast to VCP and SEP, CEP that we introduce in this chapter encodes programs at the intermediate code level. In our case, we encode programs by instrumenting LLVM bitcode [LA04]. In contrast to previous encoding approaches (VCP and SEP), adding the encoding at intermediate code level at compile time needs new concepts to encode the control flow. These newly developed concepts we published in [SSSF10a] and we will present them in more

detail in Section 8.3.3. However, encoding the intermediate code makes encoding control flow easier compared to encoding at source code level because it is not necessary to handle nested control structures explicitly. In contrast to the VCP, CEP provides automatic encoding of programs with arbitrarily nested control structures and dynamically allocated memory.

In contrast to SEP, CEP provides a more complete protection because:

1. CEP uses the replacement operations, which are not yet used by SEP. Thereby, for example, encoded execution of bitwise logical operations and shift operations is supported.
2. CEP also protects against bugs in the compiler back-end that generates code for a specific machine. SEP cannot detect compiler bugs because encoding is only applied at runtime. However, using additional checksums, SEP can at least safely detect modifications of the still unencoded binary.

At the same time, CEP introduces much less overhead than SEP because no expensive interpretation is required. Furthermore, CEP restricts usage of expensive dynamic signatures to dynamically allocated memory. CEP uses *static* signatures (i.e., signatures computed at compile time) for all statically allocated memory, that is, memory for which we know at compile time the amount that is allocated and when which memory location is read or written. In contrast, in SEP, every data item has a dynamic signature because all signatures are assigned at runtime due to the interpreter-based implementation which prevents any predicatability of variable and memory accesses.

Chapter overview

In the following, we will first give an overview of the execution environment of binaries encoded by our encoding compiler. Next, we will describe the workflow used to obtain an encoded binary. This is followed by a detailed description of the encoding applied by the encoding compiler. We focus especially on the encoding of control and data flow which is the main contribution of this chapter. The encoding of simple data transforming operations such as arithmetic or logical operations is already described in Chapter 4. Last, we will present our evaluation of CEP that is comprised by measurements of the error detection capabilities, the generated additional runtime, and a comparison to other error detection approaches such as replication.

8.1. System Overview

Figure 8.1 summarizes the execution environment of a binary that was encoded using our encoding compiler. This binary is directly executed by the CPU without the need for an additional interpreter or virtual machine.

Input/Output

The encoded binary takes input variables and produces output variables. In our current implementation of CEP, input of an application is encoded at runtime, and its output is decoded at runtime. In the future, the system could be extended to support direct communication of different encoded applications. Therefore, encoded applications could directly exchange encoded data. Thereby,

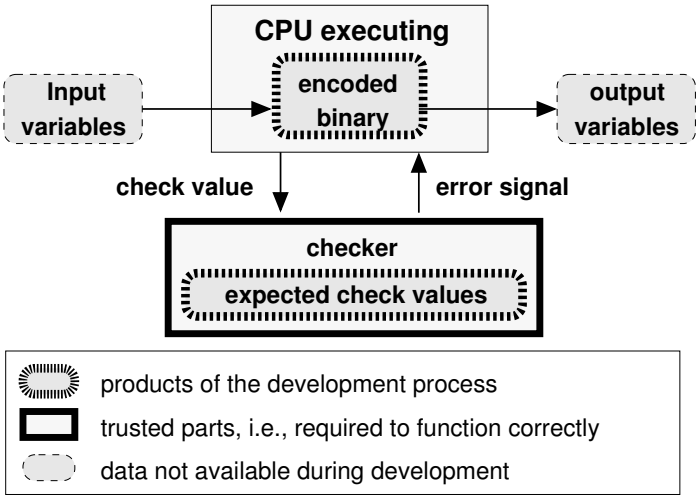


Figure 8.1.: Execution environment of an encoded binary.

the protection provided by encoding can be extended to the communication occurring between these applications.

In contrast to VCP, for CEP, we do not known at compile time when exactly input and output takes place. The reason is that for CEP, control and data flow are not required to be predictable at compile time. Nevertheless, we are able to assign static signatures to input variables and to precompute static signatures for output variables. The reason is that within basic blocks[1], which may contain input or output functions, the control flow is predictable statically, i. e., at compile time. Only for checking the control flow between basic blocks a dynamic approach is used. Furthermore, also the data flow between the variables of one basic block is statically predictable. Only data flow in dynamically accessed memory is not predictable statically. However, in LLVM, the parameters and results of input and output functions are always statically allocated variables. Thus, we know at compile time when an input or output function will be called within a basic block and which variables it uses. Hence, we can assign static signatures to the parameters and results of input and output functions.

Execution checking

During its execution the encoded binary continuously produces check values. These values and their order are determined during encoding, i. e., at compile time. The list of *expected check values* is stored in the checker. Note that the encoded binary itself does not contain this list directly. If an error disturbs the execution, the encoded binary will produce unexpected check values with high probability. That will be detected by the checker, which has the list of the expected check values. The checker is a watchdog that checks that the encoded binary continuously sends check values and that the values received by the checker match the expected values. If a mismatch is detected or no value is received in time, the execution of the encoded application is stopped because it is assumed to be erroneous.

[1] *"A basic block is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed)."[All70]*

Trusted parts

As for VCP and SEP, CEP requires that the checker is not disturbed by execution errors. The CEP-checker executes a simple algorithm that consists of a watchdog and the comparison of the check values received with the check values expected and stored in the checker. Thus, several mechanisms such as duplication or VCP-based encoding can be used to make the checker safe, i. e., insusceptible to execution errors. Furthermore, several checkers can be used independently in parallel. If any of them detects an error, it will halt the application.

8.2. Workflow

Our encoding compiler takes a C program as source code as input and produces an encoded binary and the list of expected check values. The encoding compiler supports different encodings. The user can decide which of the following encodings is applied to the program using a compiler flag:

- an AN-code,
- an ANB-code, or
- an *ANBDmem-code*, that is, an ANBD-Code is used for all dynamically allocated and accessed memory and an ANB-Code is used for the remaining statically allocated memory, for example, the variables used by a basic block.

Restrictions of the source program

Nevertheless, the encoding compiler poses some restrictions on the input source code. The binary produced by the encoding compiler is a 64-bit binary. However, the C program is not allowed to process integer types larger than 32-bit because we use the additional 32 bit for the redundancy introduced by the multiplication with A and the addition of the signature. Furthermore, all floating point operations have to be implemented using an encodable software implementation of floating point operations such as [sof09]. During compilation and encoding, we modify the program so that it is ensured at runtime that stack and heap addresses will never become larger than $2^{32} - 1$. This is required because we encode address computations and memory accesses also.

Translation to bitcode

Figure 8.2 shows the structure of the encoding compiler and the intermediate results produced. First, the *compiler frontend* translates the C source code into a lower-level intermediate code. By exchanging the compiler frontend other source languages then C code could be supported in the future. In our case, the generated intermediate code is the LLVM intermediate code (*LLVM bitcode*) because our encoding compiler is based on the LLVM compiler framework [LA04], which also provides the compiler frontend that we use. The advantage of LLVM bitcode, in comparison to any native assembler, is its manageable amount of instructions for which we have to provide encoded versions and the available LLVM framework for analyzing and modifying LLVM bitcode. We will provide a short introduction into the LLVM bitcode in Section 8.3.1.

Encoding of bitcode

In the next step, several *encoding passes* provided by us transform the unencoded bitcode, i. e., the unencoded intermediate code that represents the program, into encoded bitcode. This requires nearly a complete rewrite of the code because we have to:

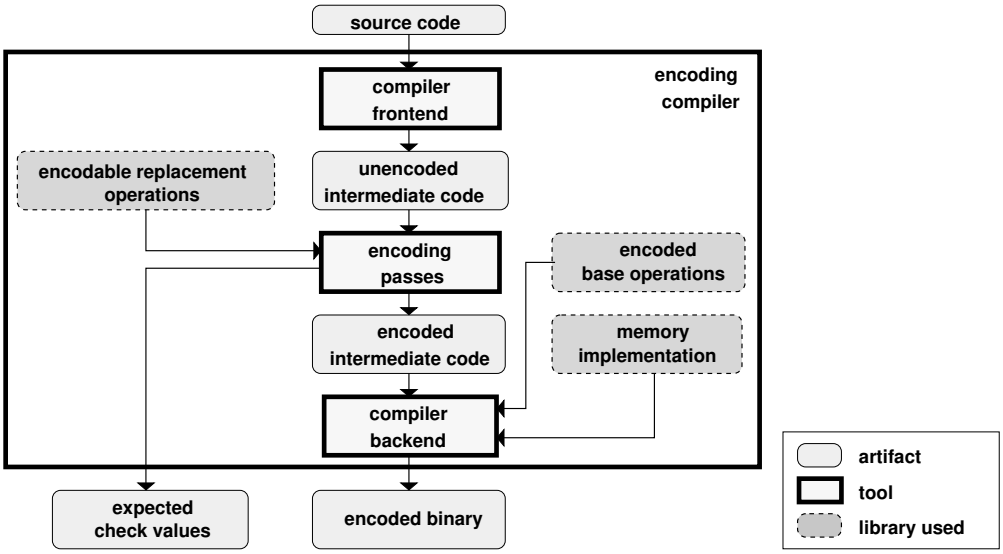


Figure 8.2.: Structure of the encoding compiler.

- ensure that all memory accesses – both to stack and heap – use only addresses that have at most 32 bits,
- replace all processed data items with their encoded versions, which require a larger data type,
- replace all operations either
  - with their encodable replacement operation before the actual encoding is done, or
  - with calls to their encoded counterparts during the encoding process,
- add the code that sends the check values to the checker,
- for the ANB- and the ANBDmem-encoding, we have to add the encoding of data and control flow, and
- we have to add code for handling input and output, which is realized by calling external functions, i. e., functions whose source code is not available to us and, thus, is not encodable.

Lowering

Last, the bitcode encoded in this way is transformed into an executable binary using the *compiler backend*. The compiler backend is also provided by the LLVM compiler framework. Generating the executable binary requires to lower the bitcode to native machine code, and to link the resulting native machine code with the libraries implementing the encoded base operations and the implementation of the encoded memory backend. We describe the implementation of the encoded base operations in Section 4.2.1. The memory backend contains all functions required to encodedly access dynamically allocated memory. We describe its implementation in Section 8.3.3 on page 150.

Note that it is not required to link in the replacement operations, which are used to substitute specific operations with an encodable implementation of the operation's functionality (see Section 4.2.2). The replacement operations are

already inserted by one of the encoding passes before the actual encoding is done. This is required because their implementation is automatically encoded by subsequent encoding passes that follow the one that applies the replacement operations. Therefore, naturally, their implementation has to be already part of the bitcode.

8.3. Program Encoding

This section describes how we encode LLVM bitcode. Thereby, we focus on the description of the encoding of the control and data flow because for these parts of the encoding CEP uses different approaches than VCP and SEP.

First, we will introduce LLVM bitcode. Afterwards, we first discuss the preparations we apply to the unencoded bitcode that the compiler frontend generates. These preparations include, for example, the application of the replacement operations. This is followed by the discussion of the actual encoding.

8.3.1. LLVM Bitcode

This section introduces the LLVM bitcode. A basic understanding of LLVM bitcode is required for understanding the following sections. An extensive reference can be found on the LLVM website [LLV].

Programs in LLVM bitcode are composed of modules. Each module represents one compilation unit. We link all modules of the program that we encode into one module using the LLVM linker before passing the program to the encoding passes.

Module structure

Modules contain global variables, functions, and symbol table entries. Global variables and functions are also denoted as global values and their names always start with an `@`.

Function

Each function definition starts with the function declaration using the `define` keyword. The declaration determines the name and type of the function. The type of a function comprises its return type and its parameters with their types.

A function's implementation consists of basic blocks. One of the basic blocks is marked as the `entry block` with which the execution of the function starts. A basic block is a list of instructions, for example additions or loads from memory. The first instruction is called the entry point and execution of the block always starts with this instruction. Furthermore, the basic block is always closed with one terminator instruction, for example, a conditional branch instruction, that exits the block. The variables – so-called *registers* – local to the function have names starting with a `%`.

Type system

Global and local variables in LLVM bitcode are typed, that is, each variable – often also called a register – has a type such as `i32` representing a 32-bit integer or `float` representing a floating point value. LLVM bitcode provides cast operations that allow to assign the content of a variable of one type to a variable with another type.

Not only basic types are supported, but also so-called *derived types* such as pointers, structures, unions, vectors, and arrays. The pointer type is used to access values stored in variables of an aggregate type such as structures or arrays. Variables of such aggregate types are always located in memory that is explicitly accessed using `load` and `store` operations. For accessing their content, a pointer is explicitly dereferenced using the `getelementptr` instruction that implements the required pointer arithmetic. A `getelementptr` instruction receives always a pointer and several indices as parameters. The pointer is the base address and the indices identify the element of the aggregate data structure that shall be accessed. The actual reading or writing access to the address computed by `getelementptr` is executed by an explicit `load` or `store` instruction.

Control flow

LLVM bitcode provides several control flow instructions. These are used as terminator instructions and are always the last instruction in a basic block. Supported control flow instructions are:

br is a conditional or unconditional branch to statically known basic blocks.

indirectbr is a branch whose destination is chosen from a list of possible destinations at runtime.

switch chooses the destination block among a set of possible destinations based on an additionally given integer value.

invoke and unwind are function calls with a specific semantics that enables the implementation of exception handling.

Note that calls to functions using the `call` instruction do not terminate a basic block in LLVM bitcode. Nevertheless, they are control flow instructions that we have to encode.

Memory access

In LLVM bitcode memory can only be accessed using explicit `load` and `store` instructions. Other instructions such as additions or control flow instructions always take registers as parameters. They cannot access parameters in memory or store results to memory.

Single assignment form

LLVM bitcode is a *single assignment architecture*, that is, for every variable only one assignment in the whole module exists. Thus, for realizing loops and joins after a conditional execution, an additional instruction is necessary – the so-called `phi` instruction. Phi instructions are always placed before the non-`phi` instructions of a basic block. A `phi` instruction assigns a value to a variable. The value that is assigned depends on which basic block was executed previously. Therefore, a `phi` instruction has a list of (value, basic block)-pairs. This list maps a value to each possible predecessor basic block. Depending on the basic block executed before the entered block, the matching value is chosen from this list for assignment.

Listing 8.1 depicts a simple LLVM bitcode module that demonstrates the concepts introduced in this section. In the example, the function `@main` is defined, which executes an endless loop that increments a counter `%indvar`. The variable `%indvar` of the integer type `i32` is either assigned `0` if the basic block `%Loop` was entered from `%entry`, i. e., for the first time, or it is assigned the value of `%nextindvar` if the block was entered from `loop`.

```
1 define i32 @main(i32 %argc, i8** %argv) {
2 entry:
3   br label %Loop
4
5 loop: ; Infinite loop that counts from 0 on up
6   %indvar = phi i32 [ 0, %entry ], [ %nextindvar, %loop ]
7   %nextindvar = add i32 %indvar, 1
8   br label %loop
9 }
```

Listing 8.1: Example program in LLVM bitcode.

After introducing LLVM, we can now present how we encode LLVM bitcode. We start with the preparations required in the next section.

8.3.2. Preparations for Encoding

Before the actual encoding is done, we transform LLVM instructions which we cannot directly encode, into instructions that we can encode. Note that, therefore, an instruction might also be replaced by several instructions. Furthermore, we increase the size of allocated memory regions to 4-byte boundaries. This is required because encodable unaligned loads and stores are implemented using loads of 4-byte chunks. If a memory region has a size that is not divisible by 4 bytes, we might otherwise access unallocated and uninitialized memory. This process of replacing instructions with their encodable versions and increasing the size of the memory regions allocated we call *preparation* for encoding.

Our passes that do the actual encoding are able to handle all base operations as presented in Section 4.2.1, conditional and unconditional branches, memory accesses using `load` and `store` instructions, and function calls. However, they are not able to encode long jumps, `indirectbr` instructions, calls to `invoke` and `unwind`, `switch` and `select` statements, `alloca` instructions that allocate memory on the stack, and all the instructions for which we provide replacement operations (see Section 4.2.2) such as casts and bitwise logical operations.

For some of the required replacements of not directly encodable instructions, we use passes that are already provided by the LLVM-framework. For others, we provide our own passes to transform these instructions into encodable implementations. See Table 8.1 for a summary of the instructions that we replace using existing LLVM passes and Table 8.2 for a description of our own LLVM passes for replacing not directly encodable instructions.

instruction	**pass name**	**description**
longjmp	lowersetjump	replace longjmp with invoke and unwind
invoke and unwind	lowerinvoke	remove invoke and unwind using calls
switch	lowerswitch	replace switch using conditional branches

Table 8.1.: Descriptions of preparations that we realize using existing LLVM passes.

Order of preparations

Most of these preparations can be executed in any order. However, for some we have to ensure a specific order. For example, long jumps have to be handled before invokes and unwinds are removed because long jumps are replaced using invokes and unwinds which also have to be removed by replacing them with normal calls. Another example, is the getelementptr instruction. First, getelementptr instructions with several indices have to be transformed into getelementptr instructions that have at most two indices. Afterwards, these are transformed into an explicit address computation using arithmetic operations. Our framework of encoding passes enforces the order required.

Currently unsupported LLVM instructions

Currently, our preparation and encoding passes support large parts of the LLVM instruction set but not the complete instruction set. None of our preparation and encoding passes supports floating point operations. Currently, these operations have to be replaced by hand with an integer-based software implementation such as [sof09]. In the future, an automated replacement before the actual encoding can be implemented. Currently, we do also not support the vector operations provided by LLVM bitcode. These operations allow to process vectors of elements of the same type with the same operation. These could be also handled by automatically replacing these vector operations by single operations for each entry of the vector. Furthermore, the `indirectbr` instruction is currently unsupported. However, an additional preparations pass can be implemented that replaces indirectbr instructions using conditional branches, which are encodable by our encoding passes.

8.3.3. Encoding

After we prepared the LLVM bitcode for encoding we can apply the actual encoding passes. These passes replace every instruction and every variable with its appropriate encoded version. In detail the following is done:

1. All variables are replaced with their encoded version. Therefore, we have to be able to encode variables of different types.
2. All data transforming instructions are replaced with their encoded version.
3. All constants and their initializers are replaced with their encoded values.
4. For ANB- and ANBDmem-encoding, signatures are assigned and additional instructions for encoding the data flow within a basic block are added. Thereby, the control flow within a basic block is also encoded.
5. Conditional and unconditional branches are replaced with their encoded versions. Thereby, the control flow between basic blocks is encoded.
6. Calls to internal functions are encoded.
7. Calls to external functions are replaced with calls to their encoding/decoding wrappers.
8. Memory accesses are replaced by their encoded versions.

In the following sections we will describe each of these steps in more detail.

instruction	**applied transformation**
select (chooses one value based on a condition)	replace select using conditional branches, assignments and phi instruction
getelementptr instruction with more than two indices	transformation into several successive getelementptr calls with two indices
getelementptr instruction with one or two indices	transformation into explicit address computation using additions and multiplication; therefore, the size of the accessed data structures is determined using a native helper binary that writes out the size's that data structures used have on the target platform
seteq and setne	replace with an implementation based on greater and less comparisons connected by a boolean `and` for ANB- and ANBDmem-encoding; for AN-encoding equality and inequality are directly encodable and, thus, are not replaced during the preparations
alloca instruction (allocates memory on the stack, this memory can only be accessed using `load` and `store` instructions)	replace with our own encodable stack implementation; however, note that currently this implementation is not encoded due to an unresolved error occurring during its encoding; instead it is treated as external library
shift operation	replace with appropriate replacement operations; see Section 4.2.2
cast operation	
unaligned load or store	
urem and srem	
bitwise logical operation	
memory allocation	increase size of allocated memory region so that it is a multiple of 4-bytes

Table 8.2.: Descriptions of preparations for which we implemented our own LLVM passes that apply these preparations.

Step 1: Encoding Variables of Different Types

Primitive data types

We are able to encode the following primitive data types:

- integer types of the sizes 8, 16, and 32 bits
- boolean variables, i. e., an integer with the size of 1 bit, and
- pointers.

The encoded version for all these different types is always a 64-bit integer, that is, all variables of the program are replaced with 64-bit integer variables.

Derived data types

Furthermore, our encoding passes are able to handle variables with a derived data type such as arrays, strings (which are a special case of arrays), and structures. Remember that variables of these types are only accessable using pointers and `getelementptr`, `load`, and `store` instructions. Thus, the encoding of variables with derived data types is facilitated by the ability of our encoding passes to encode pointers, `getelementptr` instructions, and `load` and `store` instructions. Additionally, we enabled the encoding passes to initialize the encoded versions of variables with a derived data type with appropriately encoded data values. However, vectors and vector operations are not supported by our current implementation.

In spite of all encoded variables with a primitive data type being 64-bit integers, operations whose semantics depends on the data type of the processed value observe the restrictions posed by the data type of the unencoded operands. For example, the unencoded addition %res = **add i8** %param1, %param2, which adds two 8-bit integer values, in its ANB-encoded version becomes:
%res_c = **call i64** @add_anb_8(**i64** %param1_c, **i32** %sig1, **i64** %param2_c, **i32** %sig2, **i32** %A).
Thereby, the function @add_anb_8 implements the overflow behavior of an addition of 8-bit integer values, that is, the addition of the functional values is realized as an addition modulo 2^8. We explained in Section 4.2.1 how we implement this addition in an encoded fashion. Note that the parameter `A` contains the encoding parameter A. `A` is constant that later is inlined by the compiler. `A` is part of the function declaration because this eases unit testing of the encoded base operations with different As.

Encoding

At compile time, we have to encode the values of constants (see step 3), and at runtime, we have to encode the values returned by calls to external functions (see step 7). All other operations produce encoded values. When encoding values whose type is not already a 32-bit integer, we cast it to a 32-bit integer. This cast always will be an upcast, that is, a cast from a smaller to a larger type. Note that we use a cast operation that does no sign extension. Otherwise, we would modify the value of the functional value encoded. After the upcast, 32 bits for possible redundancy remain because encoded values are 64-bit integers. These we use for our encoding.

Note that our preparations, our decoding/encoding wrappers for memory allocation, and our encoded memory backend ensure that pointers do not contain addresses that require more than 32-bits for storage. Thereby, we reduce the address space supported by our encoded applications. We do not support the encoding of any other type that is larger than 32-bits.

Decoding

Some operations require decoding of values. Examples are calls to external functions that perform output or accesses to memory that use the decoded address. After decoding, we first obtain a 32-bit integer. This integer has to be casted to the intended original type that is used at its place in the unencoded version of the program. Afterwards, the decoded value can be used. If this decoded value is used by an external function, the value is completely unprotected

after this decoding. If this decoded value is used as address in an memory access, the correctness of the decoded address is checked by other means, which we present later in this section on page 150 when describing our encoded memory access operations.

Step 2: Replacement of Data Transforming Instructions

We define *data transforming instructions* as all instructions that take input values and produce an output value. Examples are arithmetic operations, comparisons, and logical operations. We explicitly exclude instructions for managing and accessing memory and for changing control flow.

After we executed all our preparations, the LLVM bitcode, which we are about to encode, contains only data transforming instructions for which we provide a hand-encoded version in our encoded base operations. These encoded base operations were already introduced in Section 4.2.1.

During encoding, each such data transforming operation is replaced with a call to its encoded version. This encoded operation uses the encoded variables instead of the unencoded ones and produces an encoded result. For example, the unencoded instruction %res = **mul i16** $param1, %param2
is ANB-encoded to
%res_c = **call i64** @mult_anb_16(**i64** %param1_c, **i32** %sig1, **i64** %param2_c, **i32** %sig2, **i32** %A)
or AN-encoded to
%res_c = **call i64** @mult_an_16(**i64** %param1_c, **i64** %param2_c, **i32** %A).
In both cases, our encoded 16-bit implementation of the multiplication is called, instead of executing the native unencoded multiplication instruction of LLVM.

Note that the type of the unencoded variables is not anymore recognizable on the type of the encoded variables, but on their interpretation through the operations processing the variables. While in the unencoded version of the multiplication the type of the parameters indicates that the multiplication in our example is a multiplication module 2^{16}, we have to call the 16-bit variant – `@mult_anb_16` or `@mult_an_16` – explicitly in the encoded version.

For ANB- and ANBDmem-encoded applications, we provide the signatures of the parameters when calling one of our hand-encoded base operations. The values of the signatures expected for the parameters is required for corrections such as the overflow correction. However, in contrast to our example above, in our implementation, the signatures `%sig1` and `%sig2` are no variables but hard-coded constants. Their values are chosen at encoding time. We here used variables to clarify the meaning of these parameters. We describe the assignment of the signatures in step 4.

Special Case: phi Instruction The phi instruction is a special case of a data transforming instruction. It moves the content of one variable to another. The source variable is chosen based on the predecessor basic block. For encoding `phi` instructions, we replace each value in a `phi` instruction with its encoded version.

Remember that, for ANB- and ANBDmem-encoded applications, we provide the signatures expected for the encoded parameters when calling one of our hand-encoded base operations. Theses expected signatures are constant at runtime. The encoded variables used as parameters have to match these expected signatures. However, encoded values generated by different basic blocks will and should have different signatures with high probability. Thus, we have to ensure that encoded values assigned by `phi` instructions have one signature independent of the basic block that computed the value. Therefore, we add for every `phi` instruction another one that determines an adaptation value for each predecessor basic block that might compute the value assigned. This adaptation value is chosen in a way that its addition changes the signature of the encoded value to the one intended for this basic block. After executing all `phi` instructions, we add to each variable assigned by a `phi` instruction its adaptation value determined by the matching `phi` instruction. Thereby, we ensure that these variables will have always the same signature in this basic block, independent of the basic block that computed the value assigned to them.

See Listing 8.2 for an unencoded usage of a `phi` instruction. The variable `%var` at the start of `bb3` is either assigned `%var1` or `%var2` depending on which basic block was executed before entering `bb3`.

```
1  bb1:
2    %var1 = ...
3    br label %bb3
4
5  bb2:
6    %var2 = ...
7    br label %bb3
8
9  bb3:
10   %var = phi i32 [ %var1, %bb1 ], [ %var2, %bb2 ]
11   ...
```

Listing 8.2: Unencoded usage of `phi` instructions.

We present the encoded version of the `phi` instruction depicted Listing 8.2 in Listing 8.3. Note that all original variables (`var1`, `var2`, and `var`) are replaced with their encoded versions (`var1_c`, `var2_c`, and `var_c`), which are 64-bit integers. In the comments we show the signatures the values assigned will have in an error-free execution. Line 11 contains the modified original `phi` instruction that now uses the encoded variables instead of the unencoded ones. In line 12 we choose the appropriate value for adapting the signature of `var_c`. Note that the adaptation values are also completely determined during encoding. Thus, they are constants at runtime and we can hard-code them into the program as can be seen in line 12. The adaptation value is added to the chosen encoded value in line 13.

Note that an AN-encoded `phi` instruction will not contain lines 12 and 13 because no signatures have to be adapted.

Errors that lead to the selection of the wrong encoded value and the matching wrong adaptation value are not detectable. These errors could happen if the

```
1  bb1:
2    %var1_c = ... ; B_var1 = 22
3    br label %bb3
4
5  bb2:
6    %var2_c = ... ; B_var2 = 33
7    br label %bb3
8
9  bb3:
10   ; intended signature for var_c: B_var = 77
11   %var_c_tmp = phi i64 [ %var1_c, %bb1 ], [ %var2_c, %bb2 ]
12   %var_adapt = phi i64 [ 55, %bb1 ], [ 44, %bb2 ]
13   %var_c = add i64 %var_c_tmp, %var_adapt ; B_var = 77
14   ...
```

Listing 8.3: ANB-encoded usage of `phi` instructions.

control flow is disturbed by errors. For detecting them, we added additional redundancy that checks that the adaptation value expected and the one assigned are equal. The details are presented in the diploma thesis of André Schmitt [Sch09].

Step 3: Encoding Constants and Initializers

LLVM gives us access to all constants and initializers. We choose A and the static signatures of all variables at encoding time, i. e., compile time. Like the static signatures of variables we also randomly choose the signatures of the encoded constants at encoding time. Thus, we can replace the unencoded constants and initializers with their encoded versions at compile time for all the supported AN-codes, that is, for the AN-, the ANB-, and the ANBDmem-code likewise. Therefore, we change the type of the constant and encode its value using the chosen encoding parameters.

Step 4: Encoding of Data Flow and Intra-Basic Block Control Flow

While an AN-code only detects operation and modified operand errors, an ANB-code and an ANBDmem-code can be applied in a way that also ensures the detection of exchanged operands and operators and arbitrary combinations of these errors. Furthermore, the ANBDmem-code enables us to detect lost updates of memory.

This and the following steps show how we encode LLVM bitcode with an ANB-code in a way that facilitates the detection of data and control flow errors for programs. The same mechanisms are applied to ANBDmem-encoded programs because they are only different with respect to their encoding of memory accesses. This section focuses on the data flow and the control flow within one basic block, that is, that the instructions within one basic block are executed in the correct order using the intended operands. In constrast, the following steps discuss the encoding of control flow between basic block and the encoding of function calls.

Other approaches

Remember that the VCP and SEP also provide these comprehensive error

detection capabilities. However, the VCP requires statically predictable data flow and allows output only at one specific point in the program execution. Only at this point execution errors are detectable because only there the code of the output is checked. In contrast to VCP, we implement for CEP a continuous checking of the program execution. This enables CEP to

1. allow output at arbitrary positions,
2. support programs for which we do not know the data flow statically, and
3. provide *fail-fast* behavior, that is, CEP detects errors as fast as possible, thereby, allowing for an earlier reaction to them.

SEP, like CEP, provides encoding for applications with statically not predictable control and data flow, and SEP also allows output at arbitrary positions in the program executed. However, SEP has a high runtime overhead.

Check values for detecting errors

For detecting any execution errors, CEP checks the encoding of data that is about to be externalized for validity. Furthermore, for ensuring the fast detection of errors and the detection of control flow errors, our application encoded by the encoding compiler continuously produces check values, which it sends to the checker. The checker tests if the check value received from the encoded application equals the check value expected. The goal of the encoding is that if an execution error happens, the encoded application will not send the expected check value to the checker. Thus, the generation of these check values is implemented in a way that ensures the detection of all execution error symptoms defined in Section 2.5 and supported by the arithmetic code used.

We statically determine the expected check values and give them to the checker as an ordered list s. We encode the application in a way that it will not produce the check values expected with a high probability if an error disturbed the application. The list s of expected check values is indexed by a counter i by the checker. The counter i counts the check messages received by the checker. The encoded application also has a counter i for sent check messages. This counter allows the application to provide the expected check value in an error-free run. Therefore, the application contains a list *delta*, which has the same size as the checker's list s. However, *delta* contains the differences of consecutive elements of s, i. e., $delta[i] = s[i+1] - s[i]$. The application does not contain s directly.

Signature and correctional value assignment

To all variables used we randomly assign signatures at encoding time. Therefore, we use a uniform distribution. Furthermore, we compute correctional values for each usage of a variable for adapting the signatures of the encoded variables before the usage. These correctional values are also constant at runtime. They are added to the encoded variable before its usage. This addition adapts the signature of the encoded variable to a value that allows the encoded instruction using the variable to produce a correctly encoded result. If no adaptation is required, the adaptation value is zero and the compiler later removes the useless addition of this zero. For determining the correctional values, the signature correction functions of our library of encoded base operations are used. We described signature correction in Section 4.2.1 on page 51.

Block signature BBx

Using the signatures assigned and the correctional values, we can precompute a signature for each variable computed by a basic block. Using the signatures of

the computed variables, we compute – also at encoding time – for every basic block `x` a *block signature BBx*. The block signature BBx of the basic block `x` is the sum of the signatures of all results produced in this block.

Accumulator

Furthermore, we add an *accumulator acc* to the application. This accumulator *acc* is a global variable that summarizes the complete execution of the encoded application. It is used to provide the check values that are sent to the external checker.

The accumulator *acc* is initialized for each basic block `x` so that it contains the next $s[i]$ expected by the checker minus the basic block's signature *BBx*, which is the sum of the signatures of the values computed by this basic block. While the basic block is executed, the signatures of all results produced are added to *acc*. At the end of the block, *acc* should equal $s[i]$ and is sent (using the `send` function) to the checker. The accumulator *acc* will not contain the expected value if any error modified the data flow, order of the instructions executed, the computations, or the data values.

In contrast to existing control flow checking solutions, our control flow checking implemented using *acc* provides more than inter-basic block checking. We also check that every instruction was executed in the correct order, with the right operands, and its execution itself was error-free.

We will demonstrate the approach discussed in this section on a simple example in the next section, after introducing the encoding of unconditional branches.

Step 5(a): Encoding of Inter-Basic Block Control Flow: Unconditional Branch

In this section we will first present our encoding of unconditional branches. Then we demonstrate the control and data flow checking within a basic block and between basic blocks connected by an unconditional branch using a simple example.

Before terminating a basic block by branching to the next basic block, we have to adapt the accumulator *acc* for the next basic block. By this adaptation, we can provide control flow checking. We ensure that if the control flow is transfered to the wrong basic block after adapting *acc*, the basic block signature contained in *acc* will not be the expected one with a high probability. In this case, the next check value sent to the checker will we wrong.

To adapt *acc* for the execution of the next basic block, we subtract the next block's signature from *acc* and update the contained value $s[i]$ to the next expected check value $s[i+1]$ by adding $delta[i]$. Afterwards, i is incremented. For an unconditional branch, the next basic block is known at compile time. We can precompute its the signature at encoding time because at this time also the signatures of all variables used in a basic block are assigned. Thus, we precompute the signature of the next basic block statically and add an instruction that subtracts this signature from *acc* to our source basic block that contains the unconditional branch that we are encoding.

Note that it is not necessary to encode the counter i that counts the check values already sent. If i is erroneously modified, e. g., not incremented, modified by a bitflip or a wrong increment, then the wrong *delta* will be applied to *acc*. Thus, *acc* at the end of the next basic block will evaluate to a check value not expected by the checker.

However, with that approach the following error scenario might still occur: It is possible to jump from any call to `send`(*acc*), which sends the check value to the checker, to any other call to `send`(*acc*) because in this moment *acc* contains the current $s[i]$ and is only adapted to the next basic block after the call to `send`. To prevent this, we assign to each basic block `x` an ID *BBx_id*. The ID *BBx_id* is subtracted from *acc* before a block is executed. However, it is not removed from *acc* before sending *acc* to the checker. Instead, the basic block ID is also sent to the checker. The checker checks if $acc + BBx\_id == s[i]$. If not, the checker shuts down the application.

Basic block ID

The ID's for the basic blocks are chosen randomly using a uniform distribution. Note that the ID *BBx_id* of a basic block x and its signature *BBx* are different values. In contrast to the ID *BBx_id*, the signature *BBx* is completely removed from *acc* before transmitting *acc* to the checker. Thereby, the computations executed are checked because the signatures of their results are added to *acc*. These additions step by step remove the signature *BBx*, which is the sum of the signatures of the results computed in the basic block `x` and was removed from *acc* before executing the current basic block. However, the ID *BBx_id* remains in *acc* until *acc* was sent to the checker together with the expected value for *BBx_id*. Only if the ID stays in *acc*, it is possible to check if the `send` call of the correct basic block was executed. The reason is that if control flow is transfered to any other `send` call, the following corrections will assume another basic block ID and, thus, make *acc*'s content invalid.

After sending *acc* to the checker, the basic block ID is removed together with subtracting the signature of the next basic block. Both values, basic block ID and signature, are constant at encoding time and, thus, can be summarized into one value. This ensures that both values are either applied to *acc* or none of them is applied. If they are not applied, *acc* will not contain the expected basic block signature. Thus, if this update of *acc* is lost, wrong check values will be sent to the checker with high probability.

We will now demonstrate the encoding of control and data flow within a basic block and of an unconditional branch using an example. See Listing 8.4 for the unencoded version of our example. Note that we use for our examples a simplified pseudocode that is similar to LLVM-bitcode. Furthermore, our examples assume that no signature corrections are required to improve readability.

Our ANB/ANBDmem-encoding compiler transforms the example presented in Listing 8.4 into the code presented in Listing 8.5. The comments (denoted by ';') show the expected value of the accumulator *acc*. Note that `xc` means the encoded version of `x` where `x` can be either a variable or a function/instruction. Line 1 shows which value *acc* has at the beginning of the basic block `bb1` assuming that the execution up to now was not disturbed by errors. This value of *acc* is

```
1  bb1:
2    x = a+b
3    y = x-d
4    br bb2
```

Listing 8.4: Unencoded example that is used in the following for demonstrating the encoding of a basic block and encoding of an unconditional branch instruction. Therefore, the content of basic block `bb1` and the branch to block `bb2` will be encoded.

```
1  bb1: ;acc=s[i]-BB1-BB1_id   BB1=Bx+By=(Ba+Bb)+(Ba+Bb-Bd)
2    xc = addc(ac,Ba,bc,Bb) ;Bx=Ba+Bb
3    acc += xc mod A        ;acc=s[i]-BB1-BB1_id+Ba+Bb
4    yc = subc(xc,Bx,dc,Bd) ;By=Bx-Bd=Ba+Bb-Bd
5    acc += yc mod A        ;acc=s[i]-BB1-BB1_id+2*Ba+2*Bb-Bd
6                           ;    =s[i]-BB1-BB1_id+BB1 =s[i]-BB1_id
7    send(acc, BB1_id)
8    acc += delta(i)            ;acc=s[i+1]-BB1_id
9    i++                        ;acc=s[i]-BB1_id
10   acc += BB1_id-BB2-BB2_id   ;acc=s[i]-BB2-BB2_id
11   br bb2
```

Listing 8.5: Encoding of data and control flow within a the basic block `bb1` and of the unconditional branch to basic block `bb2`.

ensured by the previously executed block. Lines 2 and 4 contain the encoded versions of the original instructions. The signatures of the results produced by these instructions are computed and added to *acc* directly after executing the instructions. Thereby, the execution of these instructions and data and control flow within the basic block as well are checked. Finally, in line 5, *acc* has the value $s[i] - BB1\_id$. In the next line, *acc* and the constant basic block ID *BB1_id* are sent to the checker. The checker checks if the sum of both values equals the expected $s[i]$. The following lines adapt *acc* for the next basic block. Line 8 ensures that *acc* will contain the next check value $s[i+1]$ and line 10 adds $BB1\_id - BB2 - BB2\_id$. This addition removes this block's ID $BB1\_id$ and instead introduces the next block's ID $BB2\_id$ and signature $BB2$. Note that the value added in line 10 is a constant known at encoding time. We only display it as a sum for demonstrating our encoding. That the different correctional values are summarized into one constant ensures that either all corrections are added or none at all. Thus, the loss of their addition is detectable and cannot lead to undetectable errors.

Step 5(b): Encoding of Inter-Basic Block Control Flow: Conditional Branch

For encoding conditional branches, we additionally have to to check that the reached jump destination matches the actual branching condition. We explain our encoding of conditional branches using the example depicted in Listing 8.6. In the example, *cond* is the branch condition, and the control flow goes either to *bb_true* if *cond* is one, i. e., `true`, or to *bb_false* if *cond* is zero, i. e., `false`.

```
1  bb1:
2    cond = ...
3    br cond bb_true, bb_false
```

Listing 8.6: Unencoded example that is used in the following for demonstrating the encoding of a conditional branch instruction.

```
1  bb1: ; acc=s[i]-BB1-BB1_id and  BB1=Bcond
2    condc = ...          ;condc=A*0+Bcond if cond is false
3                         ;    or A*1+Bcond if cond is true
4    acc += condc mod A   ;acc=s[i]-BB1-BB1_id+Bcond
5    send(acc, BB1_id)    ;acc=s[i]-BB1_id
6    acc += delta(i)      ;acc=s[i+1]-BB1_id
7    i++                  ;acc=s[i]-BB1_id
8    acc += BB1_id-BBtrue-BBtrue_id-(A*1+Bcond)
9                         ;acc=s[i]-BBtrue-BBtrue_id-(A*1+Bcond)
10   cond = condc / A     ;get functional value of condc
11   acc += condc         ;acc=s[i]-BBtrue-BBtrue_id-(A*1+Bcond)+condc
12   br cond bb_true, bb_false_correction
13
14 bb_true: ; acc=s[i]-BBtrue-BBtrue_id-(A*1+Bcond)+condc
15   ; condc = A*1+Bcond
16   ; => acc=s[i]-BBtrue-BBtrue_id
17   ...
18
19 bb_false_correction: ; acc=s[i]-BBtrue-BBtrue_id-(A*1+Bcond)+condc
20   ; condc=A*0+Bcond=Bcond
21   ; => acc=s[i]-BBtrue-BBtrue_id-A*1
22   acc += BBtrue+BBtrue_id-BBfalse-BBfalse_id+A
23   ; => acc=s[i]-BBfalse-BBfalse_id
24   br bb_false
25
26 bb_false: ;acc=s[i]-BBfalse-BBfalse_id
27   ...
```

Listing 8.7: Encoding of data and control flow within a basic block and of an unconditional branch instruction.

Listing 8.7 shows the encoded version of our example for the conditional branch instruction. In line 4 *acc* is used to check the computation of the encoded condition *condc* with the already introduced approach. After sending *acc*, we adapt it in line 8 for the basic block *bb_true* and for checking if the executed branch matches *condc*. For the latter, we subtract $A * 1 + Bcond$, the value *condc* has if it represents true. The value added in line 8 is a constant known at encoding time. We only display it as a sum for demonstrating our encoding. That the different correctional values are summarized into one constant ensures that either all corrections are added or none at all. Thus, the loss of the addition in line 8 is detectable and cannot lead to undetectable errors.

In line 11, we add *condc* to *acc*. In contrast to the value added in line 8, *condc* is dynamically computed. If the condition is true, *acc* now contains the correct basic block signature and ID at the start of *bb_true*. If the condition *condc* represents false, we have to do additional corrections which are executed in the basic block *bb_false_correction* before jumping to the actual destination *bb_false*. These corrections ensure that when *bb_false* is entered, *acc* contains *bb_false*'s

signature and ID. Therefore, in line 22 we add $BBtrue + BBtrue\_id$ to *acc* to remove the term $-BBtrue - BBtrue\_id$ from *acc*. Furthermore, we subtract $BBfalse$ and $BBfalse\_id$ from *acc*. These steps prepare *acc* for the execution of the block *bb_false* instead of *bb_true*. Furthermore, we have to remove A from *acc*. Otherwise, the addition of *condc* that is done in line 11 is not neutralized correctly because we added – assuming that we will branch to *bb_true* – already $A * 1 + Bcond$ in line 8. However, if we (correctly) branch to *bb_false*, *condc* contains the value *Bcond* representing false. Note that the corrections that are applied in line 22 are also summarized in one constant. Thus, they are either applied or we will detect if they are not applied.

If no other blocks than *bb1* branch to *bb_false*, we can add the code of the block *bb_false_correction* to the start of *bb_false*. However, if different blocks branch to *bb_false*, different or no corrections at all will be required for *bb_false*. Thus, in this case, intermediate blocks – such as *bb_false_correction* – for executing the corrections are required.

If the branch executed in line 12 does not match *condc* then *acc* will not contain the expected block signature and ID because the expected values added to *acc* as constants do not match the dynamically computed values that are used to neutralize these expected values. Thus, a wrong check value will be sent to the checker. Therefore, it is required that $BBfalse + BBfalse\_id \neq BBtrue + BBtrue\_id$. Currently, we do not explicitly enforce this probability. For small functions that contain only a few basic blocks, it is improbable that the property is violated because basic block signatures and IDs are random values depending on a uniform distribution. However, in the future, it should be checked if the property is fulfilled and maybe constraint solving should be used to ensure its fulfillment.

If the execution of the branch in line 24 is faulty, this is also detectable with a high probability because *acc* most likely will not contain the expected value for executing another basic block. Therefore, the sum of the signature and ID of different blocks should be different. As stated before currently we are not enforcing this property but use randomly assigned values.

Step 6: Encoding of Internal Function Calls

After demonstrating the encoding of control flow within functions, we now introduce the encoding of function calls, that is, the encoding of control flow between functions.

Possible errors

When a function is called we have to ensure that

1. the correct function is called,
2. with correct and unmodified parameters, and
3. the function is executed correctly.

Furthermore, we have to

- provide a predictable signature for the return value of non-void functions,
- to adapt *acc* for the entry-basic block of the called function, and,

- before returning from the function, we have to adapt *acc* for the remainder of the basic block that called the function.

Correct function called?

To ensure **1.**, i. e., that the correct function is called, we assign to every function a *function signature* by which it has to modify *acc*. Before the function returns, it adapts *acc* for the remainder of the calling basic block minus this function signature. For non-void functions, an additional signature is assigned to the return value. This guarantees a predictable signature for the return value.

Correct and unmodified parameters used?

To ensure **2.**, i. e., that the correct and unmodified parameters are used, we add the expected signatures of the parameters (known at encoding time and, thus, constant) to *acc* before entering the function. In the function, we subtract the signatures of the actual used parameters (computed at runtime) from *acc*. If added and subtracted signatures do not match, *acc* will become invalid, that is, at the end of the entry block of the function a wrong check value will be sent to the checker.

After subtracting the signatures of the parameters from *acc*, the signatures of the parameters are corrected to function-specific ones that are independent of the call-site, i. e., from where the function call originated. Therefore, statically computed correction values are used that depend on the call-site. These are given to the function call as parameters whose values are constant but different for each call-site of this function.

Function executed correctly?

Before starting executing the function with the entry block of the function, *acc* is adapted. The remaining signature and the ID of the calling basic block are removed (by adding them), and the signature and ID of the first basic block – the entry block – of the function are subtracted from *acc*. The correction value used therefore is determined at encoding time and also added to the call-site dependent adaptation value. Thereafter, the execution continues as described before – now executing and checking the basic blocks of the function called. These last measures ensure **3.**, i. e., that we detect if the function is not executed correctly.

Step 7: Encoding of External Function Calls

In contrast to SEP, the static instrumentation of CEP does not allow for the protection of external libraries whose source code is not available at compilation time. For calls to these libraries, we currently provide hand-coded decoding/encoding wrappers, which decode (*including code check*) parameters and, after executing the unencoded original, encode the obtained results. For implementing these wrappers, we rely on the specifications of the external functions.

Special Case: Memory Allocation and Deallocation The functions for allocating and freeing memory are also realized as decoding/encoding wrappers. However, the allocation also takes into account that encoded values require more space than unencoded values. Thus, larger memory areas are allocated than the unencoded program would allocate.

Furthermore, our memory backend when accessing memory ensures that the addresses are mapped to the correct addresses in these larger memory areas. Note that, in principle, the functional values represented by our encoded addresses are equivalent to the addresses of an unencoded execution. Only the base addresses vary due to different allocation patterns. However, due to the larger size of encoded values these unencoded addresses do not point to the correct encoded value. Thus, the mapping is required and currently implemented using a page-table-based approach. However, encoded memory access is explained in the next section.

Step 8: Encoding of Memory Access

All the example listings that we presented so far used only variables – registers – for which static signatures are used whose values are assigned at encoding time. This is possible because for registers we know at encoding time in which order they are accessed within a basic block.

However, we cannot predict when addresses in dynamically allocated memory are written or read, that is, the access pattern of dynamically allocated memory is not known at encoding time. Thus, we need to use dynamic signatures that are calculated at runtime for values stored in memory. Dynamic signatures we introduced in Section 4.6. Remember that a dynamic signature depends on the address the value is stored to and, in case of the ANBDmem-code, also on the current version counter. For ANB-encoding, we directly use the address at which a value is stored as dynamic signature. For ANBDmem-encoding we use the sum of address and version as dynamic signature. In the context of CEP and these two codes, it is not required to ensure that the dynamic signatue is smaller than A, because dynamic signatures are only applied to values stored in memory and not to values that are decoded for externalization or used in computations.

When storing a value, that is, transfering this value from a register to memory, we convert its static signature into a dynamic signature. When loading a value from memory, that is, transfering the value from memory into a register, we convert the dynamic signature back into a static signature. The static signature is assigned to the load instruction accessing the dynamic memory. Thus, it depends on the location the load instruction has in the code. All these changes – from the static to the dynamic signature and vice versa – are also encoded.

Memory without versions

When executing a load instruction, we first have to decode the address and to load the value referenced. Afterwards, the signature of the value is adapted, that is, the dynamic signature is replaced by the static signature assigned to the load instruction executed. This adaptation is implemented in a way that errors during the load and the adaptation will result in the encoded load returning an invalid code word with high probability. Listing 8.8 demonstrates our ANB-encoded load instruction.

Note that calls to the functions `load` and `store` in our following listings do not directly execute the `load` and `store` instructions of LLVM. These functions are

calls to our memory backend implementations that before loading or storing the value map the address used to the correct value as described before.

```
1  /* ANB-encoded load instruction.
2   *
3   * @param ptrc encoded pointer, ptrc = A*ptr+Bptr
4   * @param Bptr signature of the pointer
5   * @param corr statically determined correctional value: corr = A*Br+Bptr
6   * @result encoded loaded value
7   */
8  uint64_t loadc(uint64_t ptrc, uint32_t Bptr, int64_t corr){
9     uint32_t ptr = ptrc / A;       // decode address
10    uint64_t vc = load(ptr);       // load value => vc = A*r+ptr
11
12    // adapt signature of loaded value
13    uint32_t tmp = (ptrc-corr)/A; // tmp = ((A*ptr+Bptr)-(A*Br+Bptr))/A
14                                  //     = ptr-Br
15    uint64_t rc = vc-tmp;         // rc = (A*r+ptr)-(ptr-Br)
16
17    return rc;                    // rc = A*r+Br
18 }
```

Listing 8.8: ANB-encoded load instruction.

ANB-encoded load

The encoded load presented in Listing 8.8 takes an encoded pointer $ptr_c = A * ptr + B_{ptr}$, the expected signature B_{ptr} of ptr_c, and a correctional value $corr$. During encoding we choose a value $B_r < A$ as signature of the return value of the encoded load. For each load instruction contained in a program a new signature B_r is chosen for the value returned. The signature of the encoded pointer B_{ptr} and A are also chosen at encoding time. Thus, for each call to load, the correctional value $corr = A * B_r + B_{ptr}$ is a constant whose value can be determined at encoding time. This ANB-encoded correctional value is used to encodedly compute an adaptation value $tmp = ptr - B_r$ in line 13 of Listing 8.8.

In line 15 the adaptation value tmp is subtracted from the loaded value. Remember that the dynamic signature of an ANB-encoded value is the address ptr at which it is stored. Thus, the functional value of the pointer ptr_c from which we load data should equal the signature of the encoded value read from memory. Hence, subtraction of $tmp = ptr - B_r$ from the loaded value removes the dynamic signature ptr and adds the new static signature B_r in one indivisible step. If a wrong address is read or ptr_c is an invalid code word, the return value will not have the expected signature B_r in line 17 with high probability.

ANB-encoded store

Listing 8.9 demonstrates the ANB-encoded store operation, which is very similar to the load operation. Before storing the encoded value its static signature is adapted to the dynamic signature required. This adaption is ANB-encoded and uses an at encoding time determined correctional value $corr$ which is constant for each store operation. After the adaptation, the address stored in ptr_c is decoded and the adapted value r_c is transfered to memory.

Memory with versions

For ANBDmem-encoded programs, values stored in memory are ANBD-encoded (see Section 3.5). Thus, the dynamic signature used for values stored in memory depends additionally on a *version*. This version is determined by the number of executed stores because updates of ANBD-encoded memory occur when a

```
1  /* ANB-encoded store instruction.
2   *
3   * @param vc encoded value that shall be stored, vc=A*v+Bv
4   * @param Bv signature of vc
5   * @param ptrc encoded pointer, ptrc = A*ptr+Bptr
6   * @param Bptr signature of the pointer
7   * @param corr statically determined correctional value: corr = A*Bv+Bptr
8   */
9  void storec(uint64_t vc, uint32_t Bv, uint64_t ptrc, uint32_t Bptr,
10       int64_t corr){
11
12    // adapt signature of stored value
13    uint32_t tmp = (corr-ptrc)/A; // tmp = ((A*Bv+Bptr)-(A*ptr+Bptr))/A
14                                  //     = Bv-ptr
15    uint64_t rc = vc-tmp;         // rc = (A*v+Bv)-(Bv-ptr)
16                                  //    = A*v+ptr
17
18    uint32_t ptr = ptrc / A;      // decode address
19    store(rc, ptr);               // store rc at ptr
20 }
```

Listing 8.9: ANB-encoded store instruction.

value is stored and for each ANBD-encoded value that is updated a new version has to be used. Hence, we use the number of stores previously executed by the application as version for a value stored.

A global version counter is incremented with every store executed by an ANBDmem-encoded program. This global version counter is required to compute the correct expected dynamic signature of a loaded memory value using an additional version management data structure as presented in Section 4.7. Any lost or faulty update of the global version counter will lead to inconsistencies between global version counter, signatures of the stored values, and the version management data structure. This we explained already in Section 4.7.

Furthermore, the success of the incrementation of the version counter can be checked using *acc*. For every basic block , we know at encoding time how many stores and, thus, incrementations of the global version counter it contains. Thus, we ensure for every basic block `x` that contains stores that *acc* at the beginning of the block `x` has the following value: $acc = s[i] - BB_x - BBx\_id - noOfStores$. This adaptation can be summarized with the other adaptations required for the basic block and, thus, cannot be left out unnoticedly. At the end of the block `x`, the difference of the global version counter at the beginning and at the end of basic block `x` is subtracted from *acc*. Only if all stores were executed, *acc* will contain the expected value at the end of basic block `x`. Of course, this approach does not guarantee that we detect if a store instruction adapts only the global version counter and the version management data structures and looses all other instructions. However, in that case a later load of the affected address will result in an erroneous modification of *acc* making this error scenario detectable.

ANBDmem-encoded load

For an ANBDmem-encoded load, we have to remove the expected dynamic signature *and* the version, and to replace them with the static signature of the destination register. Listing 8.10 demonstrates the ANBDmem-encoded load. The only difference compared to Listing 8.8 is the subtraction of the expected

version in line 13 and the content of the loaded value, which now additionally contains the version. The `getVersion` function used in line 13 returns the expected version for a given address. It is part of the interface of the encoded version management data structures that we described in Section 4.7.

```
1  /* ANBDmem-encoded load instruction.
2   *
3   * @param ptrc encoded pointer, ptrc = A*ptr+Bptr
4   * @param Bptr signature of the pointer
5   * @param corr statically determined correctional value: corr = A*Br+Bptr
6   * @result encoded loaded value
7   */
8  uint64_t loadc(uint64_t ptrc, uint32_t Bptr, int64_t corr){
9     uint32_t ptr = ptrc / A;       // decode address
10    uint64_t vc = load(ptr);       // load value => vc = A*r+ptr+version
11    uint32_t tmp = (ptrc-corr)/A; // tmp = ((A*ptr+Bptr)-(A*Br+Bptr))/A
12                                  //     = ptr-Br
13    uint64_t rc = vc-tmp-getVersion(ptr);// rc=(A*r+ptr)-(ptr-Br)-version
14    rc += (ptrc-corr) % A;        // additional check: (ptrc-corr)%A == 0
15    return rc;                    // rc = A*r+Br
16 }
```

Listing 8.10: ANBDmem-encoded load instruction.

ANBDmem-encoded store

Listing 8.11 contains the pseudocode implementation of our ANBDmem-encoded store operation. For a store, we need to adapt the signature of the stored value from the static signature used for values stored in registers to the dynamic signature (consisting of address and version) used for values stored in memory. First, this adaptation removes the old signature and adds the address part of the new dynamic signature. This is done in line 18 of Listing 8.11. Note that this is done in one indivisable step. If the instruction in line 18 is lost, we will detect this error whenever the value is loaded again. Next, in lines 24 to 26 the version of the stored value is adapted and the version management data structures are updated. Last, in line 29 the adapted value is stored. If this store is lost or goes to a different address, this will turn *acc* invalid whenever either the address erroneously not written or the one erroneously written are read.

8.4. Checking the Correctness of the Execution

As discussed in Section 8.1, the checker checks the correct execution of the encoded program during its runtime. The checker is not part of the encoded program and needs to be executed reliably outside of the encoded program.

Remember that the encoded application periodically sends check values to the checker (see Section 8.1). As described in Section 8.3.3 the application always sends two values at the same time:

- the current value of the accumulator *acc* and
- the ID of the current basic block.

To check the execution, the checker tests if it regularly receives these values from the application, and if the sum of the *acc* value received and the basic

```
1  /* ANBDmem-encoded store instruction.
2   *
3   * @param ptrc encoded pointer, ptrc = A*ptr+Bptr
4   * @param Bptr signature of the pointer
5   * @param vc encoded value to store, vc = A*v+Bv
6   * @param Bv signature of stored value
7   * @param corr statically determined correctional value: corr = A*Bv+Bptr
8   */
9  void storec(uint64_t ptrc, uint32_t Bptr,
10       uint64_t vc, uint32_t Bv,
11       uint64_t corr){
12
13     // Adapt signature and version of the stored value
14     // from the static to the dynamic signature required
15     int64_t tmp1 = ptrc - corr;// tmp1 = (A*ptr+Bptr) - (A*Bv+Bptr)
16                               //       = A*(ptr-Bv)
17     int64_t tmp2 = tmp1 / A;   // tmp2 = ptr-Bv
18     vc = vc + tmp2;            // vc = A*v + ptr
19
20     // decode address
21     uint32_t ptr = ptrc / A;
22
23     // adapt version and update version management data structures
24     globalVersionCounter++;
25     vc = vc + globalVersionCounter;
26     updateVersionInformation(ptr);
27
28     // store the adapted and encoded value
29     store(ptr, vc);
30 }
```

Listing 8.11: ANBDmem-encoded store instruction. Note that this implementation does not use checkpointing of the version management data structure (see Section 4.7 page 86). Therefore, additional code is required that triggers a checkpointing mechanism after a specific number of store operations were executed.

block ID received equals the $s[i]$ expected. After a check, it resets the timeout and increments i and thereby goes to the next $s[i]$ expected.

If the checker encounters an unexpected $s[i]$ or the application stops sending values (detected using a timeout), the checker terminates the application. If the end of s is reached, both application and checker start again at the beginning of s by setting i to zero. In improbable scenarios, this might lead to undetected errors due to the repeated usage of the same values. Yet, the more entries s has, the smaller is the probability of such undetected errors.

The checker has to iterate over s, do periodic comparisons with the check values received, and has to test if the application is still alive. The checker's easy implementation supports the application of various mechanisms to make its execution safe, e. g., redundant execution on different hardware such as onboard FPGAs or the graphics unit, or hand-encoding according to VCP [For89]. Additionally, we can use multiple checkers in parallel to further reduce the risk of an erroneous checker.

An efficient, asynchronous communication between checker and application is required to reduce the overhead induced by sending the check values. Currently,

we use shared memory and wait-free queues. To ensure an acceptable runtime overhead, we can adjust the frequency of the `send` operation, that is, *acc* is not sent in every basic block but only every x-th block or before data is externalized. This does not reduce the provided safety but how fast an error is detected. Every error is detected eventually. Of course, if the check values are sent with a lower frequency, errors will be detected later on average.

8.5. Evaluation

Our evaluation in this chapter focuses on the error detection capabilities and the performance overhead of applications that are AN-, ANB-, or ANBDmem-encoded at compile time. Furthermore, we compare our encoding with two replication-based approaches: SWIFT [CRA06] and SWIFT with extended control flow checking (ECF) [RCV+05a]. For these different hardware error detection approaches, we compare their error detection capabilities and the slowdowns introduced by the approaches.

First, we will introduce the benchmarks we used for our evaluation. Second, we will shortly present SWIFT and SWIFT ECF. These sections are followed by our actual evaluations of the error detection capabilities and the slowdowns introduced. Last, we compare the costs introduced by the different detection approaches with the gains – the detection probability – provided by these approaches.

8.5.1. Benchmarks Used

Usually, some parts of an application are more safety-critical than others [PGZ06]. For example, for detecting a soft error that disturbs a message transmission, it is sufficient to protect the message with a safely computed end-to-end checksum that checks that the message was delivered to the intended receiver unmodified. It is not required to provide error detection for the complete network protocol stack. Other application examples are event processing frameworks that support the development of distributed applications. While the framework handles, for example, message distribution, the so-called operators implement the business logic. The framework itself can remain unprotected while the operators have to be sufficiently equipped to detect erroneous executions that may result in silent data corruptions (SDCs).

Thus, for our comparisons, we use several algorithms that could be expected in safety-critical and event processing systems and executed them within a networked environment. While the network stack is unprotected, the safety-critical algorithms are equipped with error detection. However, we assume that the execution of the complete application – including safety-critical algorithms and network protocol – can be modified by transient and permanent errors.

For our evaluation, we use the following benchmark algorithms that can be expected in safety-critical or event processing systems:

`md5` calculates the md5 hash of a string.
`tcas` is an open-source implementation of the traffic alert and collision avoidance system [Pap09] which is mandatory for airplanes.
`pid` is a Proportional-Integral-Derivative controller [Wes00].
`abs` implements an antilock braking system.
`topK` finds the k most frequent items in a data stream [CCFC04].
`primes` implements the Sieve of Eratosthenes.

Although, `primes` is not a typical example of our targeted application domain, we included it because it uses a large array and thus is vulnerable to data modifications.

8.5.2. Other Error Detection Approaches Evaluated

We compare our three different encoding approaches, which use different arithmetic codes, with two replication based detection approaches: SWIFT and SWIFT with control flow checking (SWIFT ECF), which we shortly introduce in the following.

For evaluating SWIFT we used Martin Süßkraut's implementation. This we extended further for SWIFT ECF. Both implementations use the LLVM compiler framework [LA04], which we already use for encoding.

SWIFT

Software implemented fault tolerance (*SWIFT*) [CRA06] duplicates all instructions and registers apart from memory accesses and control flow instructions.

SWIFT does not duplicate memory because it assumes that memory is protected by other means such as parities. This can lead to undetected lost stores because store instructions are not duplicated. Loads from memory are also not duplicated because they might be uncachable [CRA06]. Instead values loaded from memory are copied. This approach opens a window of vulnerability. For example, data modifications on the bus may remain undetected.

Function calls are not duplicated. For a function defined in an external library, we do not know if it is idempotent and, thus, can be called twice. Instructions within internal functions are already duplicated by SWIFT and, thus, executed two times without duplicating the calls. Parameters of an internal function are duplicated at the start of the function. The return parameter of any function is duplicated when the call returns to the callee. These duplications can lead to undetected data modifications.

The equality of duplicates is checked before their externalization, before values are stored to memory or are used as address in a load or store instruction, or before the values influence control flow. For SWIFT any failed check results in the abort of the application. Errors occurring in the vulnerable window between check and use might remain undetected. Checks in SWIFT are not easily protectable because they are strongly interleaved with the original program.

As already discussed SWIFT contains several windows of vulnerabilities, i. e., SWIFT does not provide end-to-end detection of hardware faults. Additionally, if both duplicates are affected by the same error, for example, a permanently faulty operation, we expect that SWIFT will not detect this error. Furthermore, SWIFT is susceptible to control flow errors such as taking the wrong branch of a conditional jump.

SWIFT ECF

The authors of [RCV+05a] added enhanced control flow checking (*ECF*) to SWIFT. This facilitates the detection of control flow errors. Therefore, a unique signature is assigned to every basic block. A dedicated register named `GSR` keeps track of the signature of the basic block that is supposed to execute currently. Before leaving a basic block, an adaptation value for `GSR` is written to a second dedicated register `RTS`. For unconditional jumps, the value assigned to `RTS` is a constant. For conditional jumps, it is one of two constants depending on the jump destination. Which of the two constants is assigned to `RTS` is chosen using the duplicate of the jump condition. The actual jump uses the original. Thus, if only duplicate or only original were modified, control flow errors are detectable. At the beginning of a basic block, `GSR` is adapted to its new expected value by xor-ing `RTS` to it. Afterwards, it is checked that `GSR` contains the expected signature for this basic block. The expected signature and the adaptation values are hard-coded into the binary because they are constant.

Every mismatch of the expected and the actual basic block signature results in the abort of the application. Furthermore, any failed check of the equality of duplicates results in the abort of the application.

8.5.3. Error Detection Capabilities

Error injection

We used our symptom-based error injector EIS [SSSF10b] for our evaluation of the error detection capabilities of our encodings and the two SWIFT approaches. For a detailed description of EIS see Chapter 9. This section here contains a short introduction only.

Instead of injecting errors directly into the hardware either physically or using fault injecting hardware, EIS injects the software-level symptoms of possible hardware failures. Directly injecting at software-level reduces masking and, thus, makes the error injection more efficient.

We injected the following kinds of errors:

Exchanged operand (EO1): A different but valid operand is used, that is, instead of the intended operand another register which is already defined and has the same type is used.

Exchanged operator (EO2): A different operator is used, e. g., an addition is executed instead of a subtraction. The operands remain the same.

Faulty operation (FO): The result of an operation is modified by bitflips. We inject multiple as well as single bitflips. The number of flipped bits is chosen using a Poisson distribution, that is, single bitflips are more probable than multiple bitflips.

Lost store (LS): A store operation is omitted.

Modified operand (MO): An operand used by an instruction is modified by a single or a multiple bitflip. While faulty operation influences every read of the result, modified operand only influences one read of a register.

Further errors can be represented by combinations of these symptoms. For example:

- The replacement of a complete instruction comprised of operator and operands with a different instruction can be emulated by applying the symptoms exchanged operator and exchanged operand to the same instruction.
- Control flow errors can be emulated by combinations of such instruction replacements.
- Modifications of memory content are emulated by modified operand errors that are applied to store operations.

The error model presented before is based on the assumption that every hardware error that is not masked influences the execution of a program in some way and that all possible influences can be emulated by these basic symptoms.

Experiment setup

We applied those errors in three different modes:

Deterministic (Det): In this mode exactly one error is triggered per run. We execute approximately 50,000 runs for each benchmark and protection mechanism: 10,000 for each symptom. In each run another error of the same symptom is triggered. For each symptom the injection points are distributed equally over the possible injection points available in the program execution used. This tests the ability of a detection mechanism to cope with seldom occurring errors.

Probabilistic (Prob): Here, all error symptoms are injected with the same probability and they all might be injected in one run. We use the same error probability for all error detection mechanisms evaluated. At each possible point where an error (of any symptom) could be triggered an error is injected with the given probability. Thus, one execution might be hit by several different errors. This mode allows to mirror the fact that for an error detection mechanism which increases code size, the protected program version is more probable to collect errors than the program version without error detection. With this mode we executed 6,000 runs per benchmark and per detection mechanism.

Permanent errors (Per): In this mode we inject permanent faulty operation errors simulating permanent logic errors in the processor. Depending on the input values of an instruction, its result is modified. If a specific bit within the input values of a specific operation is set, a bit of the result is flipped. For one injection run, the targeted operation, the bit which has to be set for triggering the error, and the bit flipped in the result remain the same. Permanent errors

are only applied to arithmetic integer operations, and loads and stores of integer values. We are injecting approximately 1,700 different permanent errors per benchmark, per detection mechanism – one error only per run.

All our example applications are of similar size. Hence, with our fixed number of fault injection runs we achieve similar coverages for all applications. We chose the number of fault injection runs so that the experiments complete in a feasible time.

We compared the results of injection runs to the results of an error-free run to determine if the error injected resulted in a silent data corruption (SDC). Figure 8.3 presents the results of the described error injection experiments. It depicts the share of injection runs that resulted in an SDC, i. e., a failure of the error detection mechanism. Note the logarithmic scale.

Injection results

We make the following observations:

- We clearly see the superiority of all the AN-codes compared to SWIFT and SWIFT ECF with respect to permanent errors. For transient errors (Det and Prob) the AN-code has for most benchmarks higher detection rates (i. e., lower SDC rates) than SWIFT and SWIFT ECF and for some not.
- The ANB- and ANBDmem-codes always have an order of magnitude better detection rates than AN-encoding, SWIFT, and SWIFT ECF. While AN-encoded versions still have a considerable amount of SDCs: on average 0.96%, ANB-encoding reduces the amount of undetected errors to on average 0.07%. ANBDmem-encoding again halves the rate to on average 0.03% SDCs.
- We see that the amount of SDCs in the *native*, i. e., unprotected, case depends on the benchmark. There are programs that are more robust, for example, `tcas`, than others, for example, `md5`.
- For the native cases, we see also large differences between transient and permanent errors. While `md5` is very susceptible to transient errors (Det and Prob), it is not to permanent ones (Per). For `topK`, for example, it is the other way round.
- Probabilistically (Prob) injected errors are more often detected. The reason is that for both injection modes programs are more often hit by several errors. This increases the probability of detection as we have shown in [WF07b].
- Furthermore, benchmarks that show already high silent data corruption rates for the native case, show also rather high rates for SWIFT, SWIFT ECF, or the AN-code.
- In some cases error detection mechanism with additional protection perform worse than their supposedly weaker counter parts. For example, in 50% of the cases SWIFT ECF detects less permanent errors than SWIFT (for `topK`, `pid`, `md5`).
  Another example are the deterministic injections (det) into `tcas`. In one case (`abs` for the deterministic injection mode), ANB-encoding detects more injected errors than ANBDmem-encoding. Increased complexity

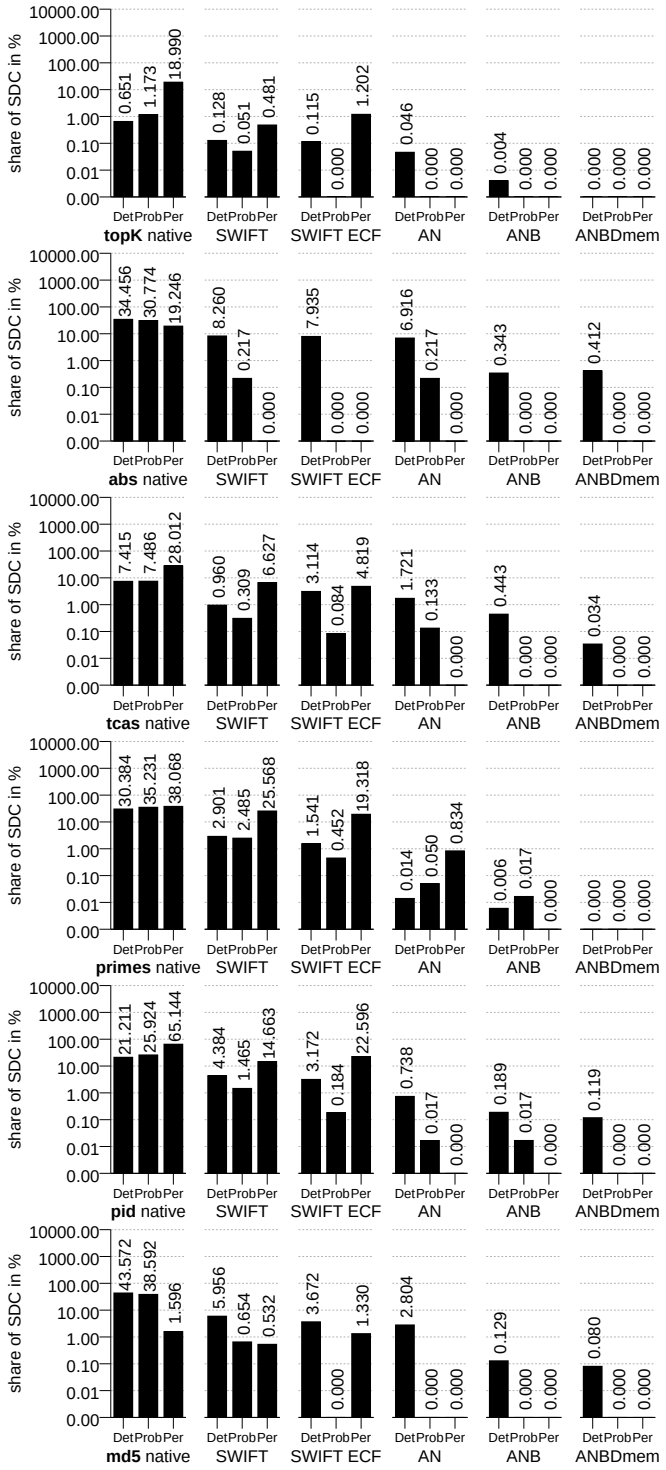


Figure 8.3.: Error injection results: Share of error injection runs resulting in silent data corruptions (SDC) in % for unprotected native applications and applications protected with SWIFT, SWIFT ECF, AN-, ANB-, and ANBDmem-codes AN-, ANB-, and ANBDmem-codes used A=65521.

seems to introduce new vulnerabilities that lead to undetected silent data corruptions. A more detailed analysis of ANBDmem-encoded runs indeed revealed such vulnerabilities for our ANBDmem-code implementation.

Figure 8.4 depicts the results of the same error injection experiments as Figure 8.3 in more detail. All the results for the deterministic injection mode are presented separately for each symptom: exchanged operand (EO1), exchanged operator (EO2), faulty operation (FO), lost store (LS), and modified operand (MO). Furthermore, not only SDCs but all possible outcomes are depicted. Apart from SDCs, we can observe the following outcomes:

correct output The error is masked and does not influence the outcome of the application execution.

failure detected The error is detected and prevented from becoming a failure, that, the application crashed without producing incorrect output. The application might crash because the error led to inconsistencies disturbing the execution, for example, by causing a segmentation fault. Or the application might crash because the error detection approach used detected inconsistencies and aborted the application.

detected performance failure In this case, a timeout occurred. For CEP-encoded applications, the checker times out. Currently we use a very large timeout, which is three times the time needed to execute the error-free golden run. For SWIFT (ECF)-ed applications, the error detection does not provide a liveness check. Here runs identifed as timed out were stopped by the error injector that also checks the execution time. Thus, for CEP-encoded applications, a timeout means the error was detected. However, for SWIFT (ECF)-ed applications, it means no error was detected.

Figure 8.4 shows that all protection mechanisms often reduce the number of runs that produce the correct output. The reason is that all the approaches also detect errors that might be masked in a further execution. However, they are not guaranteed to be masked. Thus, these detections are no false positives. If an error is detected, their definitely was an error. However, it is unclear if it really would have become a failure.

However, for some benchmarks and detection approaches the amount of failure-free runs even increases. The reason might be that the detection approaches increase the code base. Thereby, the probability of masking errors might also be increased. Consider, for example, the version management of the ANBDmem-encoded application. If an error is introduced there for an address that is never read again, then this will neither be detected nor lead to a failure.

For all detection approaches, the amount of detected errors is higher than for the native execution. Whereas, timeouts are a seldom event for native execution, SWIFT, and SWIFT ECF. For the encoded variants, the amount of detected timeouts for the permanent errors is surprisingly high. Obviously, the additional encoding introduces the possibility for endless loops in case of errors. We assume that this source is located in the memory implementation that maps the unencoded address space to the encoded space because this is one of the common code parts of all the encodings. Further common code parts exist in the decoding/encoding wrappers.

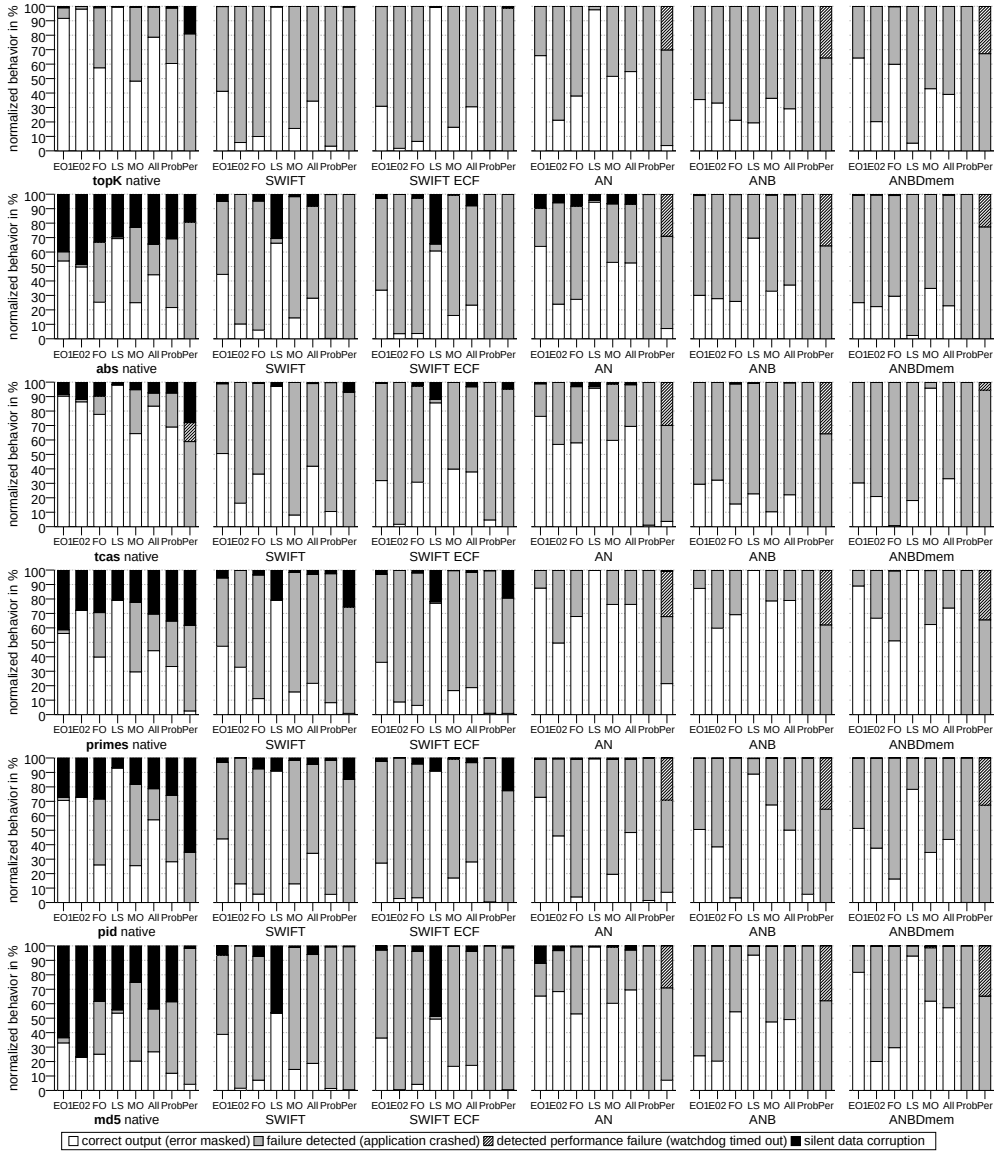


Figure 8.4.: Error injection results. AN-, ANB-, and ANBD-codes used A=65521.

Last, we are surprised to see that AN-encoding already reduces the amount of SDCs caused by lost stores significantly. We would have expected that AN- and ANB-encoding perform not well in that area.

8.5.4. Runtime Overhead

Experiment setup: networked environment

For our runtime measurements, we evaluate the benchmark algorithms within a networked environment. A client sends requests to a server application that

executes one of our benchmark algorithms. The benchmark algorithms represent the safety-critical part of the system. Hence, they are completely protected by one of our detection approaches evaluated. The client, the part of the server that receives the messages, and the network protocol stack are completely unprotected. For the encoding-based detection approaches, we transfer encoded data from client to server and vice versa to enable end-to-end detection of hardware errors.

To evaluate the performance, we measured the throughput of our client server scenario. The client sends a request to the server and waits for the reply. The server feeds the request as input to the benchmark implementation and sends back the output as reply to the client. After receiving the reply the client sends the next request to the server. We measured the throughput in requests per second. All results presented are the trimmed arithmetic mean of at least 8 measurements. The deviation is negligible. Hence, we omit it in our diagrams.

Our server test machine has two Intel Xeon processors (in total 8 cores) and runs a 64-Bit Fedora 10. The client machine has the same OS as the server, but runs on an AMD Opteron processor (in total 16 cores). Client and server are connected by a 10-MBit half-duplex network.

Results: networked environment

The throughput values shown in Figure 8.5 are relative to the throughput of the native and unprotected execution of the respective server benchmark. The throughput of SWIFT and SWIFT ECF is within the measurement error of the throughput of the native execution. As we expected, the arithmetic codes come at a higher cost. AN-encoding degrades the throughput much less than ANB-encoding. ANB-encoding degrades the throughput not as much as ANBDmem-encoding. However, the overhead varies with the application. For instance, `abs` has even for ANBDmem-encoding 79% of the throughput of the native execution, whereas `topK` protected by ANBDmem-encoding only reaches 4% of the throughput of the native execution. The reason is that some encoded operations are more expensive than others (see Section 4.2). For example, the `topK` benchmark uses more of the expensive operations than the `abs` benchmark.

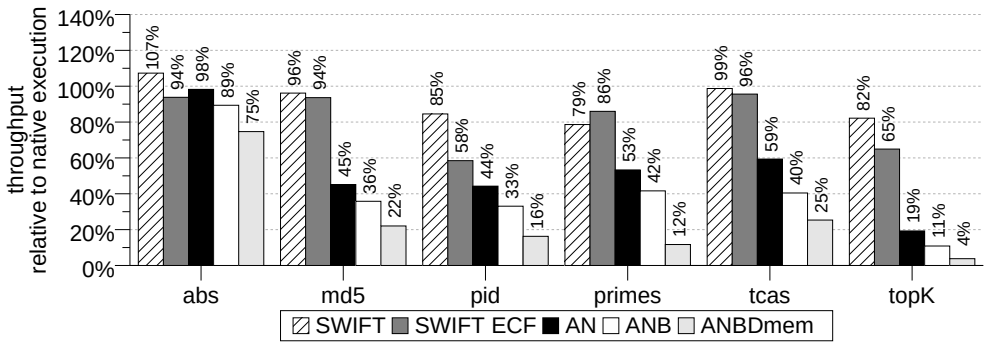


Figure 8.5.: Throughput of all detection techniques evaluated normalized against the native execution.

While in Figure 8.5 the overhead is CPU-bound, Figure 8.6 depicts the effects of a network-bound setting. Therefore, we ran eight servers and clients in parallel on the server and the client machine, respectively. Because both computers have at least eight cores, this did not introduce additional limits on the CPU. However, the network traffic increased by a factor of eight. For this setting, we also observe reduced throughput for all detection techniques. But the observed throughput degradation that comes with the increased safety is in general much lower than for the non-network bound setting. For `topK` with ANBDmem-encoding the throughput is still only 12% of the native execution. However, `abs` with ANBDmem-encoding degrades the throughput only down to 85% of the throughput of the native execution.

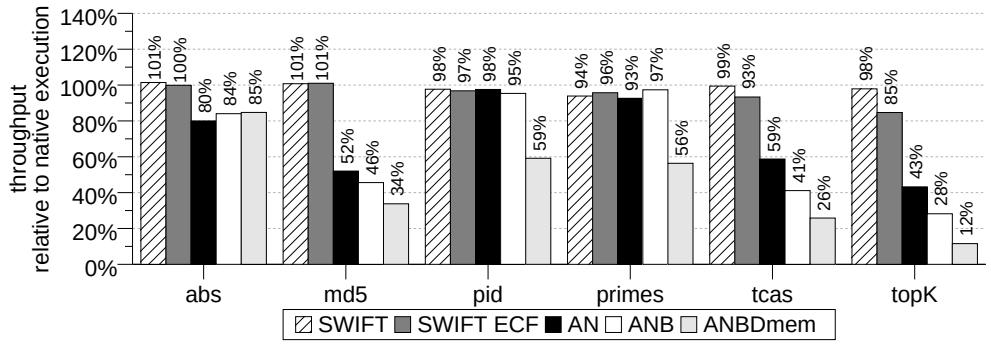


Figure 8.6.: Same setting as in Fig. 8.5 but network-bound because of eight parallel running servers and clients on each computer.

Experiment setup: completely encoded

Furthermore, we evaluated the absolute slowdowns of encoded application, that is, of applications that are completely encoded. These applications have no unencoded parts apart from data externalization realized with the decoding wrappers. Therefore, we directly executed some of our benchmarks without the above described client-server-setting and measure the times for the complete application including input-output operations. All results presented are the trimmed arithmetic mean of at least 5 measurements.

Results: completely encoded

Figure 8.7 depicts the slowdowns of the different encoded applications compared to their unencoded native versions.

For the AN-code, the slowdown ranges from 2 (`primes`) to 75 (`tcas`). As we already observed, the slowdowns vary strongly because some applications use more of the expensive encoded operations such as multiplications or floating point operations than other applications. For example, `md5` contains an above average number of bitwise logical operations, which, in their encodable version, make extensive use of expensive encoded multiplications. The encoded version of `tcas` is much slower because of the extensive use of floating point operations.

The ANB-code is on average 1.9 times slower than the AN-code because it provides encoded control and data flow and the encoded operations used have

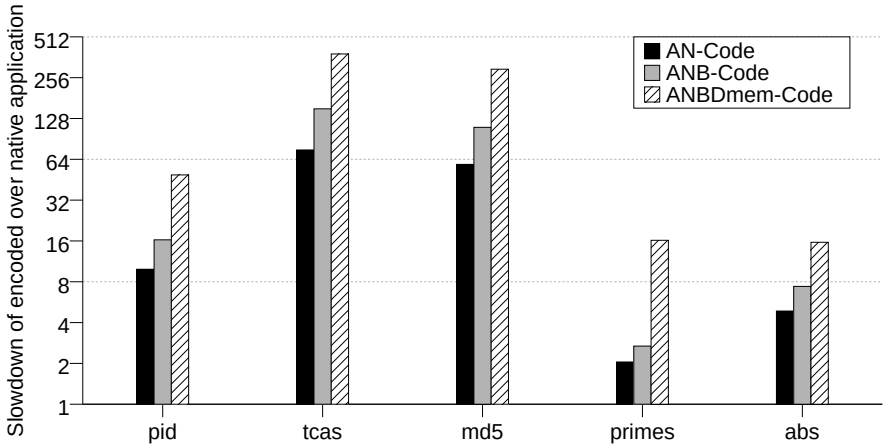


Figure 8.7.: Slowdowns of encoded application compared to their native versions.

to consider the signatures as well.

The slowdown of the ANBDmem-code compared to the ANB-code is on average 2.6. The main reason is the additional overhead needed to safely store and retrieve version information for dynamic memory. The version management is a significant part of the overall runtime overhead especially for programs that have low locality in their memory write accesses. Note that we use the list data structure (see Section 4.7) with checkpointing for version management for all our measurements because it showed the best results for most applications.

Comparison: SEP vs CEP

One objective for the development of CEP was to be faster than the interpreter-based SEP. For some applications, Figure 8.8 compares the speedup of the most expensive CEP-variant – ANBDmem-encoding – to the SEP-variant of that application. These measurements again were done for as completely encoded applications as possible, that is, the encoding used was applied to the whole application and no part that could be encoded with the technique used was left unencoded. Thus, for CEP, the application was completely encoded apart from the encoding/decoding wrappers. In contrast, for SEP, bitwise logical operations, shifts, casts and unaligned memory accesses are executed unencodedly. For the benchmarks `tcas` and `abs` no measurements are presented because they are not supported by SEP due to SEP's incomplete implementation.

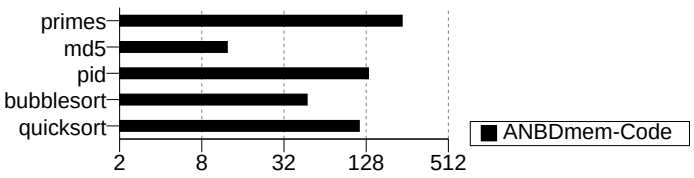


Figure 8.8.: Speedup of CEP compared to SEP.

CEP always clearly outperforms SEP. We observe that the obtained speedups depend on the executed program. Especially `md5` has smaller speedups. `md5`

contains an above average number of bitwise logical operations. However, SEP's encoding is incomplete. It especially does not support encoded versions of bitwise logical operations, shift operations, and casts. Those operations are just executed unencoded in SEP while they are encoded by CEP. Nevertheless, even for these applications that are incompletely encoded in SEP and completely encoded in CEP, CEP outperforms SEP.

8.5.5. Costs vs Gains

For comparing the different error detection approaches (CEP with its different encodings, SWIFT, and SWIFT ECF) compared in the previous section, we set their costs, i. e., additional runtime, and their gains, i. e., their error detection capabilities, into relation to each other in this section.

Figure 8.9 summarizes the engineering trade-offs when choosing one of these error detection techniques. The graph relates the costs of the presented detection techniques to the achieved gain when using them. The cost model used (y-axis) is the throughput of the protected execution relative to the throughput of the native execution, i. e., *the higher the throughput the smaller are the costs.* The used costs for the left graph are taken from the non-parallel client-server experiment depicted in Figure 8.5 and for the right graph from the parallel, i. e., network-bound, client-server experiment shown in Figure 8.6. For both graphs, the gain (x-axis) is the share of SDCs produced relative to the share of SDCs produced in the native execution (the values are taken from the measurements presented in Figure 8.3), i. e., *the smaller this remaining rate of undetected errors is the better.* We averaged the number of undetected errors for all three injection modes: deterministic, probabilistic, and permanent. Note that the x-axis is log-scale. Every point is the mean of the measurements of all our benchmark applications. The native execution as expected has no additional costs, but also no gain. Thus, the native execution is located in the upper right corner. An optimal error detection approach would induce no costs and prevent all SDC. Thus, it would be located in the upper left corner of the diagrams in Figure 8.9.

SWIFT's and SWIFT ECF's runtime overhead is negligible. But using them, about 18% of the undetected errors of the native execution remain undetected. ANB-encoding and ANBDmem-encoding have a negligible rate of undetected errors, but introduce high costs. AN-encoding has neither a negligible error detection rate nor a negligible performance overhead. Furthermore, AN-encoding has the highest variability in the error detection rate provided for different applications.

From comparing Figure 8.9 left and right, we conclude that limits of the environment (such as limited bandwidth) can hide some of the performance costs of the detection techniques used. Overall we conclude from Figure 8.9: When choosing one of the detection mechanism with higher detection rate, the performance degradation is *linear*. However, the gain, i. e., reduction of the rate of undetected silent data corruptions, grows *exponentially.* It is not surprising that the arithmetic codes lead to a higher reduction in performance in terms

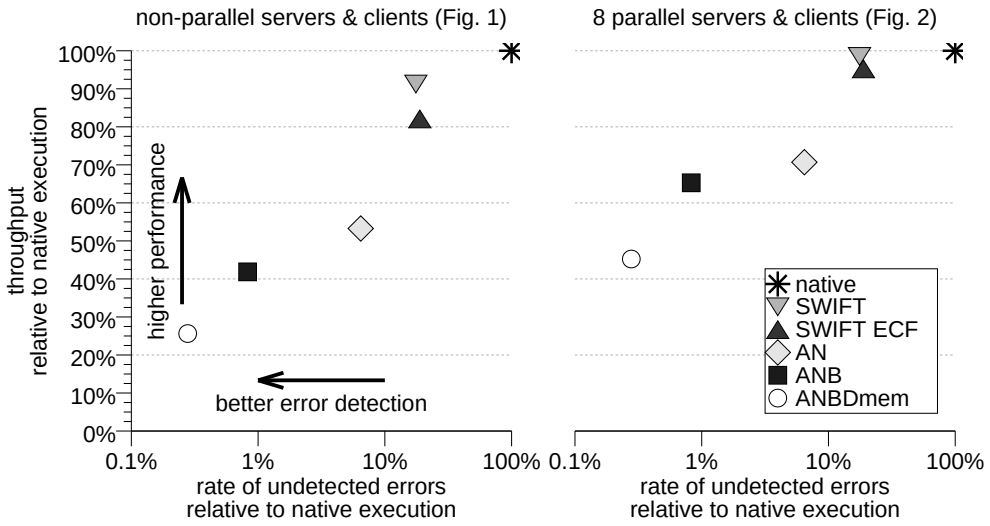


Figure 8.9.: Comparison of the cost and the gain of our five detection techniques.

of reduced throughput. Still, we are surprised by the cost-effectiveness of the ANB- and the ANBDmem-code.

8.6. Summary of CEP

In this chapter we presented Compiler Encoded Processing (CEP). We especially focused on the encoding of data and control flow and dynamic memory. Furthermore, we presented an evaluation of the runtime and the error detection capabilities of the different encodings that the encoding compiler can apply and compared them to the replication-based approaches SWIFT [CRA06] and SWIFT ECF [RCV+05a]. Additionally, we compared the slowdowns induced by CEP to the slowdowns induced by our previous approach SEP.

The encoded compiler that we described is able to apply different encodings (AN-, ANB-, and ANBDmem-code) to LLVM bitcode. Our current evaluations and tests encode C programs. Due to the flexibility of LLVM the support for other source languages could be provided in the future.

Our evaluations have shown that the different encodings and also SWIFT and SWIFT ECF reduce the amount of SDCs compared to an unprotected native execution. The different approaches provide a different reduction of SDCs. However, they also induce different runtime overhead in terms of additional execution time. We observed the following relation: When choosing one of the detection mechanism with higher detection rate, the performance degradation is *linear*. However, the gain, i. e., reduction of the rate of undetected silent data corruptions, grows *exponentially*. Thus, our encoding compiler together with our implementations of SWIFT and SWIFT ECF provides a toolset that enables a

user to trade safety and additionally required runtime.

In the future, further approaches can be implemented to provide more different possibilities of trading safety and overhead. For example, SWIFT can be extended with duplicated memory and with the direct use of duplicate values as function parameters. Thereby, several windows of vulnerability are closed and its detection capabilities are increased. However, surely the runtime overhead will increase also.

9. Symptom-based Error Injection Tools

For evaluating the error detection capabilities of our error detection approaches SEP (see Chapter 7) and CEP (see Chapter 8), we used error injection. Therefore, we implemented two symptom-based error injection tools: FITgrind and EIS (Error Injection Slicing). We developed FITgrind for evaluating SEP. For evaluating CEP, we developed EIS – a successor of FITgrind.

For testing fault tolerance mechanisms, error injection is a useful and accepted method. Tools for injecting errors exist for a wide range of fault and error types, e. g., for emulating programmer faults tools are used that apply software mutations. To test approaches that aim at the detection of hardware errors, we need to inject errors that failing hardware might cause.

Direct vs symptom-based injection

Existing tools for injecting hardware errors that we discuss in Section 9.1 either directly modify hardware state or inject symptoms of hardware failures into the running software. We use symptom-based error injection because, compared to approaches that inject errors directly into hardware state, symptom-based injection is easier to apply and the results obtained are more general because the injection is hardware independent.

Why another symptom-based injection tool?

We implemented new symptom-based injection tools and did not use any of the existing symptom-based tools for several reasons:

1. Most of the existing tools implement an *insufficient error model* that does not represent the whole range of possible errors. Usually just single bitflips are injected.
2. There are tools with an error model comparable to ours, but these tools are *processor dependent* and *difficult to use*. For example, to use the FERRARI injection tool [KKA95], which implements a sufficient error model, the user has to provide a processor model. Especially, if the error detection approach shall be evaluated independently of any specific hardware, this is a daunting and laborious task because in this case several different processor models should be used.
3. We were not able to obtain any of the symptom-based tools. We believe the reason is that most of them date back to the nineties. Thus, it is probable that they are no longer actively developed and supported. This claim is backed up by the fact that in all recent research papers presenting error detection mechanisms custom error injectors are used for evaluating the approaches presented. But as mentioned earlier, these custom injectors usually just inject single bitflips.

The results of error injection campaigns provide information about how well error detection mechanisms cope with errors. We are especially interested in the percentage of errors that cause a *silent data corruption (SDC)*, that is, the generated output is different from the output of the fault-free run and no error is reported. In particular, if the rate of SDCs is higher than expected, we want to identify the ways how errors circumvent the detection mechanisms. Therefore, we extended our error injector EIS with a dynamic forward slicing mechanism. For the errors which interest us – usually the ones which produced an SDC – we can generate a log of the data flow that was influenced by the error injected. The result is a *slice* of the trace of the complete execution of the tested application that shows only executed instructions whose operands or/and results were influenced by the error. This slice is easier to manually inspect for debugging the error detection mechanism than a complete log.

This chapter introduces our error injection tools FITgrind [WF06] and EIS [SSSF10b] that we developed and used for the injection experiments for evaluating SEP (see Chapter 7) and CEP (see Chapter 8). We presented the results of these experiments in sections 7.4.1 and 8.5.3.

Before describing our error injection tools, we discuss existing error injection and slicing mechanisms and tools. We use the results of this discussion to motivate our decision to implement our new injection tools, which are based on symptom-based error injection, instead of using one of the existing tools. Finally, we describe our tools FITgrind and EIS.

9.1. Related Work

This section describes the related work for the injection of hardware errors. First, we discuss error injection tools. However, the authors of research papers often seem to use their own specific injectors. Thus, next, we discuss the injection experiments that were used in recent research papers for evaluating approaches for detecting hardware errors. Furthermore, we shortly discuss related work for slicing to show that none of the existing approaches is sufficient for debugging hardware error detection approaches. Last, we summarize our findings by deriving design decisions for our own injectors FITgrind and EIS.

9.1.1. Error Injectors

In the following, we give an overview over error injection mechanisms that allow to inject or simulate hardware errors.

Physical Injection. The most realistic hardware errors can be injected by directly influencing the hardware, thereby simulating the real causes of hardware errors. For example, injectors exist that bombard the processor with heavy ions, radioactive particles or heat, or that inject errors into the pins of a circuit. One example system is the LFI-injector [JRSMF98] that uses heat generated by a laser.

All physical injection methods have in common that they are rather expensive because special hardware and an elaborate setup is required. Furthermore, the system under test might be damaged, and controllability and reproducibility are very low. Debugging error detection mechanisms would be very difficult under these conditions.

Simulation-based Tools. Simulation-based tools try to overcome the disadvantages of physical injections while at the same time keeping the advantage of being very realistic. These tools simulate a processor into which gate-level errors (stuck-at-0 or stuck-at-1), bridging errors, or timing errors are injected. Providing such a simulator is a daunting task. The controllability and reproducibility of this approach are bought with high runtime overhead generated by the processor simulator.

Both, simulation-based tools and physical injection are hardware-dependent. This makes it difficult to test hardware-independent detection mechanisms. Furthermore, the masking rate for errors injected at such a low level is rather high. Yount and Siewiorek [YS96] showed that for gate-level injection only 30% of the errors are activated. Wang et. al. [WQRP04] and Blome et. al. [BMBF05] observed similar high masking rates. Injecting errors directly as symptoms into running applications is more efficient because hardware masking is circumvented. According to Arlat et. al. [ACK+03] injection at software-level can be used to emulate hardware errors, as long as not only the data segment but also the code segment of an application is subject to injections.

Hardware-based Tools. Hardware-based tools use special functionality provided by the hardware to modify the current state of executed software. For example, errors can be injected using the debug and test interfaces of modern CPUs. Software tools can access these interfaces and use them to modify system state. One of the tools following this approach is Xception [CMS98].

Obviously, hardware-based tools are also hardware dependent. However, we prefer hardware-independent error injection for evaluating our hardware-independent error detection approaches. Furthermore, the setup of these hardware-based injectors is rather complex.

Error Injection Using Formal Methods. The error injection approaches described so far never cover all possible error scenarios because realizing a complete injection using these approaches is not possible due to the time required. In contrast, error injection using formal methods can provide the guarantee that all consequences of the errors injected are investigated. Therefore, possible errors are formally modeled. Then, these error models are applied to a model of an application or to the application directly. All the possible outcomes resulting from the application of the general error models are analyzed. The analysis either shows that the error modeled was detected or not. In the latter case, a weakness in the error detection is found.

For example, the projects KeY [LH07] and SymPLFIED [PNKI08] use symbolic execution to implement a symbolic injection of errors. KeY and SymPLFIED aim at verifying the completeness of error detection approaches, that is, they try to prove that all errors defined by an error model are detected. Therefore,

during the symbolic execution all variables (one after another) are marked as erroneous, and the outcome of the symbolic execution is checked for failures, i. e., undetected SDCs. SymPLFIED [PNKI08] uses model checking to detect these failures.

Another example for formal error injection is presented in [NGY+05]. This paper presents the verification of the completeness of a signature-based control flow checking approach using model checking. Therefore, a general program using the signature-based control flow checking approach and all possible control flow errors are modeled and checked using the SPIN model checker[1].

However, symbolic error injection and formal error injection based on model checking exhibit major problems that prevent their usage for large systems so far:

- An error model and, for some approaches, a model of the application have to be provided. The completeness of this model determines the quality of the results obtained. If the model is too abstract or misses important error classes, possible failures of the error detection approach can remain undetected.
- For larger problem sizes the state explosion that is inherent especially to model checking makes the formal approaches very expensive if not unusable. All three of the formal error injection methods discussed here ([NGY+05, LH07, PNKI08]) were only applied to relatively small examples. The largest program analyzed with one of the approaches (SymPLFIED) requires only 1550 lines of assembly code and, thus, is relatively small.
- Last, adaptations are required for checking probabilistic error detection approaches such as approaches using arithmetic codes. The reason is, that as long as the probability for an error is not zero, formal approaches will find error scenarios.

Software-based Tools. A wide variety of software-based tools for injecting symptoms of hardware errors exists. Most of them (see the next section) just insert single bitflips, an error model we deem insufficient. The tools Ferrari [KKA95] and FINE [KIT93] do not only inject such data errors but also control and data flow errors, e. g., by using different input registers or leaving out instructions. But Ferrari requires the specification of a processor model and FINE is tailored for injection into the Linux kernel only. Furthermore, both tools are not freely available or are not available anymore.

None of the error injection tools known to us supports debugging of error detection mechanisms as our tool EIS does.

9.1.2. Error Injectors Used in Recent Research Papers

Most of the software-based error injection tools are rather old. In recent research papers that present methods for detecting execution errors, mostly custom injectors are used. Of the 14 papers that we reviewed 13 used their own specific

[1] SPIN model checker: http://spinroot.com.

error injector and one did no evaluation of the provided reliability using error injection. In the following, we investigate the capabilities of these custom injection tools.

For recently published error detection mechanisms aiming at general hardware errors, mostly single bitflips into data processed by an appliction were injected. While the injectors used in [YGS09] and [WsKWY07] are implemented using dynamic binary instrumentation for injecting single bitflips into data, the injectors in [CRA06, PKI07] are realized using static instrumentation. The authors of [RCA+06, WP06], and [RCV+05a] solely describe the error model used to evaluate their detection approaches as single bitflip. However, the authors do not discuss the implementation of the error injection used. For testing the approaches presented in [RLC+08] and [RR07] simulation-based injection of single bitflips was used.

However, recent studies (e. g., [DHW09] and [BMBF05]) show that decreasing feature sizes lead to an increasing number of multiple bitflips. Thus, solely injecting single bitflips is insufficient to test hardware error detection mechanisms. Furthermore, solely injecting into the data segment of an application is not sufficient to emulate all symptoms possibly caused by hardware errors [ACK+03]. Injections in the code segment, that is, changing the executed instructions, are also required. For example, the simulation-based injector used to evaluate the techniques presented in [NPI07] injects errors into the data processed and the code executed as well. Our error injectors also modify the data processed and the instructions executed. However, we inject these errors as symptoms into the software executed while [NPI07] uses a hardware simulation to inject the errors. This simulation is more time consuming because it requires to simulate a hardware architecture. Furthermore, it obviously is hardware-dependent.

For techniques that check the validity of the control flow of an application, we see a wide variety of techniques for evaluating the ability of these checks to detect control flow errors. The evaluations executed range from no evaluation with error injection for the technique presented in [BWWA06], over simple modifications of the program counter in [VFM06] to quite elaborate error models implemented by the error injector described in [VA06] and [VHM03]. Especially the gdb-based error injection SFIG used in [VHM03] applies a very comprehensive error model including:

- replacement of one or two instructions,
- usage of wrong operands and wrong result locations,
- exchanging operators,
- modification of operands and results, and
- modification of condition flags.

Unfortunately, SFIG is platform-dependent and only supports Sun Sparc machines. However, currently, we do not aim at supporting Sun Sparc as a platform for our encoding approaches.

9.1.3. Slicing

Our error injector EIS uses slicing to facilitate debugging of error detection approaches. Originally, slicing is a technique used for automatic debugging. To compute a slice means to reduce a program or a trace of a program execution to these parts that are of interest, for example, to the parts that influence a specific variable.

Several tools implementing slicing exist. For example, Triage [TLH+07] generates slices as bug reports for production systems. Therefore, Triage repeats the last seconds of an application's execution after a (crash) failure with slicing enabled. Thereby, Triage computes a backward slice that contains any dataflow leading to the failure. Dimitrov et. al. [DZ09] use slicing similarly to Triage for filtering anomalies in the dataflow to a given failure. Our tool EIS could also be extended to compute a backward slice from corrupted output to an injected error. Therefore, the user would have to identify the corrupted output generated by the application that is subject to error injection by EIS.

However, our injector EIS uses slicing in another way. It follows the dataflow from a given error to an undetected failure – an SDC. We do not know of any error injection tool that provides the possibility to compute a dynamic forward slice based on the injected error, i. e., enables the user to observe how the error propagates. Several papers, for example [HJS01] and [ANS+04], address static analysis of systems to assess the susceptibility of their components to propagate errors. But these approaches are not suitable for debugging error detection mechanisms.

9.1.4. Design Decisions Derived

After reviewing existing error injectors and error injection methods used in recent research papers, we decided to implement a software-implemented symptom-based error injector. Instead of injecting errors directly into the hardware either physically or by using special hardware interfaces or by simulating the hardware, we abstract from the actual hardware error and inject the software-level symptoms of possible hardware failures. The advantages of this approach are manifold:

Less masking: Yount and Siewiorek [YS96] showed that around 70% of the gate-level errors are already masked at the level of architected state, that is, they never reach application registers. Directly injecting at software-level reduces masking and, thus, makes the error injection more efficient.

Hardware independence: We want to use the error injector to test error detection mechanisms which are hardware-independent. Thus, we also want to test as general as possible, that is, hardware-independent. This is provided by injecting symptoms at software-level.

Good controllability and reproducibility: To facilitate debugging of undetected errors, for example, by using forward slicing, it is essential that the error injection is deterministic and easy to control. Since we are injecting errors

into an application at the level of machine code, we can exactly determine where we inject which error and when. This facilitates reproducibility of the injection runs and detailed debugging by determining which further instructions are influenced by the error.

Costs and ease of use: Symptom-based error injection requires no additional hardware or software, e. g., processor simulators, processor models or computers for controlling the injection process. This reduces setup costs. Furthermore, compared to simulation-based injections the runtime overhead is much lower.

On the other hand, the abstraction of the error model used by symptom-based error injection bears the risk of using an error model that does not mirror real errors exactly. Furthermore, in contrast to injections using formal methods, symptom-based error injection cannot provide guarantees that all errors are detected with a specific probability.

For implementing our symptom-based injectors, we assume that any non-masked hardware failure results in a software-level symptom. For example, timing issues might lead to a register being written too late. An instruction using this register will use the register's previous value. We can simulate this with the software-level symptom that exchanges an operation's operand with another value.

Furthermore, we consider the often used single bitflip error model insufficient because:

1. Arlat et. al. [ACK+03] showed that errors modifying only the data segment of an application, that is, the values processed, cannot emulate the whole set of possible hardware errors. Only additional modifications of the code segment, that is, which instructions are executed with which data, result in a sufficient coverage of possible hardware errors by the symptoms injected.
2. Reduced feature sizes in todays and future integrated circuits lead to a growing amount of multiple bitflips [DHW09, BMBF05].

Considering this, we decided to insert error symptoms that modify data and data and control flow. Thereby, we oriented ourselves on the error model described in Section 2.5.

9.2. FITgrind

We developed the error injection tool FITgrind for evaluating SEP (see Chapter 7). For implementing FITgrind, we used dynamic binary instrumentation with the help Valgrind [Net04]. Thus, FITgrind adds the code required for injection at runtime.

FITgrind abstracts from the underlying hardware architecture. Errors are injected as error symptoms into the artificial architecture provided by Valgrind. FITgrind can inject the following types of errors:

- Modification of operands and results of instructions to simulate bitflips or stuck-at faults in memory, registers or on buses.

- Exchange of operands with other operands to simulate address line faults during data access.
- Replacement of instructions with other valid instructions or groups of instructions to simulate address line faults during instruction loading.

It would be also possible to simulate bitflips in instructions or operand addresses, but Valgrind will reject most of these injections because they generate invalid code.

9.2.1. Design and Implementation

Error injection using dynamic instrumentation

For instrumenting a binary using Valgrind, we have to execute this binary with Valgrind. Valgrind translates every basic block encountered during the execution of a binary into UCode which forms the hardware model on which FITgrind is injecting faults. *UCode* is a single-assignment load-store architecture, that is, every register is assigned at most once and memory is accessed using explicit load and store instructions. Valgrind facilitates the implementation of tools that can instrument the translated basic blocks.

We register FITgrind as a tool with Valgrind. Thus, Valgrind hands every translated basic block to FITgrind for instrumentation. After FITgrind instrumented the basic block, Valgrind transfers the instrumented basic block from UCode into native binary code. The result is stored into a cache for later reuse and then executed. Thereafter, the next basic block, which is the destination of the jump leaving the previous block, is translated and instrumented in the same fashion.

FITgrind randomly chooses locations within a basic block for error injection and extends the UCode accordingly. Valgrind tools (as FITgrind is one) can modify existing UCode instructions, add new UCode instructions, or add calls to so-called dirty helpers. *Dirty helpers* are C functions to which calls can be added to a basic block. Using these possible instrumentations, we can add code that might inject errors when it is executed. For example, if we want to inject an error that modifies the result of an instruction, we redirect the result of the original instruction into another register by modifying the UCode. Then, we call a dirty helper that randomly decides between the following two options:

- it either stores the original result into the original destination register, that is, it does not inject an error, or
- it stores a modified version, that is, it injects an error. For example, to modify an operand, Valgrind randomly chooses how many bit shall be flipped. Therefore, it uses a Poisson distribution because single bitflips are more probable than multiple bitflips [LSHC07]. Then, FITgrind randomly chooses the bits using a uniform distribution and inverts their values.

All other error types are implemented in a similar fashion. Currently, the user of FITgrind can configure which of the supported error types he wants to inject with which probability. Of course, FITgrind could also be extended to support a more detailed choice of error injection points, e. g., by time or instruction type as provided by Xception and FERRARI.

Experiment execution

FITgrind allows the user to define *error injection campaigns* with given probabilities for errors, error types, and target applications. One campaign consists of several error injection runs for one target application with one configuration for the error types and their probability. The different runs of one campaign differ in the seed used for the random number generator controlling the error injection process. Furthermore, for each campaign, we execute one *golden run* in which we do not inject errors. Each error injection run is repeatable by choosing the same configuration of probabilities, error types, and seed.

The output of an error injection run and the golden run are compared byte wise. The following results are possible:

correct but incomplete output In this case, we first observe correct output. However, at some point, the output stops. The reason is that either the operating system detected the error, for example, if an unallocated memory area was accessed due to the error, or that another error detection mechanism detected the error and prevented erroneous output.

no output No output at all was generated by the erroneous run. The application crashed before any erroneous data could be externalized. Thus, *no output* is a special case of *correct but incomplete output*. In this case, the error is detected before any output is generated.

correct and complete output We observe the correct output, that is, the output of the run with an injected error and the error-free golden run are equal. There are no deviations. Thus, the error was masked. For example, a later unused value was modified or a value was modified that was used for a comparison. This comparison might still have evaluated to the correct result despite the erroneous comparison parameter.

incorrect output In this case, a silent data corruption occurred, that is, output was produced that differs from the error-free run. Obviously, the reason is that undetected errors became failures. When testing our error detection approaches, this is a failure of the error detection. In contrast to crashes, arbitrarily erroneous output is difficult to detect as such.

Furthermore, we parse the output of the program to check if an error was detected and which error was detected. Thus, we will see if an error was detected by the operating system or a specific error detection mechanism. All results, the error injection configuration and the types and number of the errors injected are stored in a database for later analysis.

9.2.2. Results

We presented the results of our error injection experiments for evaluating SEP in Section 7.4.1. FITgrind clearly demonstrated the differences between unprotected applications and applications protected by SEP. For the latter, the rate of SDCs observed was much lower and we could also see that about 8% of the errors detected were detected by SEP exclusively.

To determine the runtime overhead of FITgrind, we tested it with a recursive grep as target application for which we executed 1,300 injection runs. The time

measurements were taken with adding the instrumentation for triggering errors but without actually triggering them. One error injection run was on average 21.5 times slower than an uninstrumented natively compiled run. This is the same slowdown Valgrind (version 3.2.0) generates when no instrumentation is done at all. When triggering errors, the runs are almost always faster, because of crash failures.

9.3. EIS

FITgrind supports our intended error model (see Section 2.5) only partly. Furthermore, the dynamic binary instrumentation used by FITgrind leads to large slowdowns – at least for these injection runs that do not crash immediately. Thus, we decided to develop the new injection tool named EIS[2] that is based on static instrumentation instead of dynamic instrumentation that was used by FITgrind.

EIS stands for *Error Injection Slicing*. This name captures that EIS does not only inject errors, but also provides a debugging mechanism for error detection approaches. This debugging mechanism applies slicing to allow the user to determine where an error circumvented error detection mechanisms.

In the following, we first introduce the error injection provided by EIS. Afterwards, we describe the slicing for debugging error detection mechanisms.

9.3.1. Error Injection

In EIS, we insert the error symptoms at the level of the LLVM bitcode [LA04] of the application's source code. For implementing EIS, we use the LLVM compiler framework that allows us to modify LLVM bitcode. We give a short introduction to the LLVM compiler framework and LLVM bitcode in Section 8.3.1. If you are not familiar with LLVM, you should read this section before going on with the current section. However, note that the error model implemented by EIS is not tied to LLVM. It can be easily mapped to other assembler languages and with more difficulty to programming languages.

LLVM's platform-independence allows simulation of hardware errors independent of any hardware. In contrast to error injection at source code level, insertion in LLVM-bitcode is reasonably easy and allows to realize a wide variety of symptoms of hardware errors. Furthermore, LLVM-bitcode is easier to read than assembler and easier to map to the original source code.

On the other hand, the LLVM framework restricts us in the kind of errors we can insert because it performs a restrictive code consistency check. For example, it is not possible to use an undefined register.

[2]Note that EIS is a joint work with Martin Süßkraut and André Schmitt. While the author of this thesis mainly implemented the error injection part of EIS, Martin Süßkraut and André Schmitt provided the slicing part that facilitates the debugging of error detection mechanisms.

The Error Model

Error symptoms

The injector inserts the following error symptoms that were first introduced by [For89] and that we already describe in Section 2.5:

- **Exchange operand**: A different but valid operand is used, that is, instead of the intended operand another register which is already defined and has the same type is used.
- **Exchange operator**: A different operator is used, e. g., an addition is executed instead of a subtraction. The operands remain the same.
- **Faulty operation**: The result of an operation is modified by bitflips. This can be multiple as well as single bitflips. Every read of the result is influenced by the injected error.
- **Lost store**: A store operation is omitted.
- **Modify operand**: An operand used by an instruction is modified by a single or a multiple bitflip. In contrast to a faulty operation, a modified operand only influences one read of a register.

These symptoms represent data and data flow errors.

Currently, errors modifying the control flow of an application are supported implicitly because any conditional jump can be subject to the existing error model. For example, a modified or exchanged condition operand could result in erroneous control flow. In the future, EIS could be extended with explicit support for control flow errors. Therefore, error symptoms have to be added that

- modify existing jump instructions or
- insert new jump instructions.

This facilitates the simulation of errors that arbitrarily modify the program counter.

Injection modes

In contrast to FITgrind that supported only probabilistic injection, EIS supports different *injection modes*:

- **Deterministic:** In this mode exactly one error is triggered per run. Usually several thousands of such runs are executed where in each run another error of the same type is triggered. This tests the ability of a detection mechanism to cope with rarely occurring errors. Furthermore, we can determine if an error detection mechanism is especially susceptible to some error types.
- **Probabilistic:** This mode combines all error types. The user has to provide the probability that an error will occur. At each possible point where an error could be triggered a random number is generated. The random number and the error probability provided by the user determine if an error is injected or not. Thus, one execution might be hit by several different errors. This mode allows to mirror the fact that for an error detection mechanism which increases code size, the protected program

version is more probable to collect errors than the program version without error detection.

- **Permanent errors:** In this mode we inject permanent faulty operation errors simulating permanent logic errors in the processor. Depending on the input values of an instruction, its result is modified. If a specific bit within the input values of a specific operation is set, a bit of the result is flipped. For one injection run, the targeted operation, the bit which has to be set for triggering the error, and the bit flipped in the result remain the same. Permanent errors are only applied to arithmetic integer operations, and loads and stores of integer values.

Trigger Points and Error Triggering

Injecting errors into an application, which might or might not contain error detection mechanisms, is a two-stage process: First, EIS statically, i. e., at compile time, injects the code which might trigger an error. We call the code injected *trigger points*. Second, when executing a trigger point at runtime, we decide if an error is injected or not. The decision algorithm used depends on the injection mode chosen by the user.

Inserting Trigger Points When inserting the trigger points, the user has to decide if he wants do a probabilistic, deterministic, or permanent injection. For a deterministic injection, he has to provide the desired error type. For a deterministic and a probabilistic injection, he also has to provide the trigger frequency that determines how many of the possible trigger points are inserted. If the user chooses a trigger frequency of `x`, every `x`-th possible trigger is inserted. Thus, a frequency of one inserts all possible triggers. For example, for the modify operand error type that means that at runtime every operand used in the application could be replaced with a modified one. For large applications, inserting all possible trigger points results in too high memory requirements for linking the result obtained by the injection process to the libraries required for the error injection. The reason is that every trigger point inserted contains a call to a function implemented in an additional library. To circumvent these out-of-memory problems of the linker process, we could directly insert the code for triggering an error. However, this surely would lead to a code explosion that makes lowering of the resulting LLVM code to machine code difficult.

Note that the trigger frequency for the injection of permanent errors is always one. This ensures that a permanent error is always triggered if its trigger conditions are true. If in the future more types of permanent error should be supported, this might lead to compilation and linking problems due to resource constraints.

Example: trigger point

In the following, we present the insertion of a trigger point. Take, for example, the following LLVM bitcode extract which represents the multiplication of two 32-bit integers:

```
1 %c = mul i32 %a, %b
```

After inserting trigger points for the modified operand error type with a trigger frequency of one the code looks as follows:

```
1 %a_bf = call i32 @bitflip_i32(i32 %a)
2 %b_bf = call i32 @bitflip_i32(i32 %b)
3 %c = mul i32 %a_bf, %b_bf
```

The `@bitflip_i32` function gets one argument: the register which might be modified at runtime. At runtime, `@bitflip_i32` decides if an error is injected or not. If an error is injected, `@bitflip_i32` returns a modified version of its argument. Otherwise, it returns the unmodified argument. Furthermore, we replace the original operands of `mul` with the possibly modified versions `%a_bf` and `%b_bf`. However, the decision if an error is injected that is implemented in `@bitflip_i32` depends on the chosen injection mode.

Probabilistic trigger point

For the probabilistic injection, a random number between 0 and 1 is generated and compared to the trigger probability provided by the user of EIS. If the number is smaller than the probability, the error is triggered and `@bitflip_i32` returns a modified version of the original operand. Otherwise, no error is triggered and `@bitflip_i32` returns the original, unmodified operand.

Deterministic trigger point

For the deterministic one-error-per-run injection, a counter is incremented in `@bitflip_i32` at each passed trigger point. The user gives an *error ID*, i. e., the counter value identifying the passed trigger point at which an error is to be injected. At runtime, when the trigger point with the given counter value is reached, an error is injected. No other passed trigger point will inject an error for the deterministic injection.

Permanent trigger point

For a permanent error, the decision if an error is injected depends on the instruction executed and the values of the parameters given to this instruction. If the targeted instruction is executed and a specific bit of the input parameters is set, a bit of the output value is flipped. For each different injection run, a different operation or a different bit of the input parameters is automatically chosen as a trigger condition.

Implementation of the Error Symptoms LLVM is a typed language. To simplify the presentation, we only present the instrumentation for the LLVM type `i32` (32-bit integer). Operands and operators of other types are instrumented accordingly. We have implemented the five error symptoms as follows:

- **Exchange operand**: The instrumentation adds a call to `@select_operand_i32 (i8 %no_alt_ops, i32 %orig_o, ...)` for each operand that shall be replaceable at runtime. The second argument `%orig_op` is the original operand. Other operands with which the original operand could be exchanged are passed as vararg. Matching operands are selected by type statically at compile time. The first argument `%no_alt_ops` gives the number of passed alternative operands. At runtime, when an error is triggered the function returns the value of one of the other operands. If no error is triggered the original operand's value is returned.

- **Exchange operator**: If an operator shall be exchanged, its result is passed through the function `@exchange_operator_i32 (i32 %result, i32 %op1, i32 %op2, i8 %op)` The argument `%result` is the original result of the operation. The constant `%op` identifies the original operator (such as `add`, `sub`, `or`, and so on). The original arguments to operator `%op` are passed as `%op1` and `%op2`. If no error is injected `%result` is returned. Otherwise, a random operator (different from `%op`) is applied to `%op1` and `%op2` and its result is returned.
- **Faulty operation**: The result of any operation whose result we want to modify is passed through `@bitflip_i32()` as introduced above.
- **Lost store**: All `store` operations that are to be left out are wrapped by the function `@lose_store(i32* %ptr, i32 %val)`. If no error is injected `%val` is stored at `%ptr`. Otherwise, `@lose_store` returns without doing a `store`.
- **Modify operand**: The instrumentation for modify operands uses `@bitflip_i32()` as introduced above.

Triggering Injections Before executing injection runs, an error-free *golden run* is executed. Its results are used to determine the effect of the error injection. For example, by comparing the output of the golden run with the output of an error injection one can identify silently corrupted output.

The golden run is also used to measure the time that is required for an error-free execution. For error injection runs, we wait three times the execution time of the error-free run before we kill the application and assume that the injected error resulted in a deadlock or and endless loop.

Furthermore, the golden run provides the number of passed trigger points. This number is normally higher than the number of inserted trigger points because injected trigger points located within loops mostly are passed multiple times. In the deterministic injection mode, the number of passed trigger points is used to determine the trigger point that has to inject an error in the current run. If the number of the trigger point passed equals the ID of the error that shall be injected in the currently executed run, an error is injected. The error IDs used are uniformly distributed over all trigger points passed at runtime.

To execute the actual injections the user of EIS has to provide the chosen injection mode (deterministic, probabilistic, or permanent). For the deterministic mode, the number of injections has to be provided. For the probabilistic mode, additionally the probability with which a trigger point turns into an error is required. For the permanent mode, always all possible errors are injected. For every possible permanent error, one injection run is executed. The information provided by the user is used in the trigger points, e. g., the `@bitflip_i32` function, to decide if an error is inserted.

Determining the Injection Results As for FITgrind an error injection campaign consists of several injection runs applying the same symptom in the same

mode at different points of the application. The results of these injection runs are compared to the error-free golden run. We described the possible outcomes already in Section 9.2.1.

Evaluation

We used EIS for evaluating our Compiler Encoded Processing presented in Chapter 8. Therefore, we parallelized its execution. This is easily possible because the different injection runs are independent of each other. The results of these injections are discussed in Section 8.5.3.

9.3.2. Debugging with Forward Slicing

Most hardware error detection tools only detect a certain amount of silently corrupted output failures. However, the question arises if the remaining undetected failures are caused by the incomplete coverage of the detection approach or by bugs in the error detection. EIS' debugging support helps to analyze and debug undetected failures. Therefore, EIS identifies the complete data flow of an injected error through the application. It provides this information to the user as a so-called *forward slice*. The forward slice produced by EIS represents the LLVM instructions that were influenced by the error. This enables the user of EIS to detect bugs in his error detection mechanisms because he can read LLVM-bitcode and map it to the original source code. By inspecting the data flow the developer of an error detection mechanism can find missing checks and missing redundancy in the detection mechanism or just plain bugs in the implementation of the detection mechanism. Hence, EIS's debugging support is two-fold:

- It helps to improve the coverage of the error detection approach and
- it helps to debug the error detection implementation.

Approach

To facilitate the debugging of error detection mechanisms, EIS computes a *slice* S_e for every injected error e.

Definition 1. *The* slice S_e *of an injected error* e *is the set of all instructions operating on values directly or indirectly influenced by error* e*.*

Together with the data flow the slice contains also the control flow influenced by the injected error e. However, deviations from the control flow of an error-free execution are hard to detect because only the control flow of the erroneous run is part of the slice. Furthermore, the slice contains all output instruction that may generate erroneous output because of the injected error.

For convenience, EIS generates an XML representation of S_e. The XML representation can be automatically post-processed for further analysis or more

```
1  [+] @abs:
2   [+] bb1:
3    %reg1 = call i32 @bitflip_i32 (i32 %tmp0)
4    store i32 %reg1, i32* %addr1
5
6   [+] bb2:
7    %reg2 = load i32* %addr1
8    store i32 %reg2, i32* %addr2
9
10  [+] return:
11   %reg3 = load i32* %addr2
12   ret %reg3
13
14 [+] @main:
15  [+] bb4:
16   %reg1 = call i32 @abs (i32 %tmp0)
17   %reg2 = add i32 %reg1, %tmp1
18   store i32 %reg2, i32* %addr3
19   ;; loop start (repeated 350 times)
20   %reg3 = load i32* %addr3
21   %reg4 = add i32 %tmp2, %reg3
22   store i32 %reg4, i32* %addr3
23   ;; loop end
24
25  [+] bb15:
26   %reg5 = load i32* %addr3
27   call void @printf(i8* %msg, i32 %tmp3, i32 %reg5, i32 %tmp4)
```

Figure 9.1.: An example slice produced by EIS. The data flow that is of interest, that is, depends on the error injected, is underlined. A bitflip is injected in function `@abs`. It propagates into `@main`, through a loop and into the output function `@printf` where the bitflip becomes visible as corrupted output.

human-accessible presentation. Currently, we only support the generation of a browsable HTML version of S_e. The user can manually inspect S_e to analyze and debug why the error e is not detected. Note that a slice is more than the difference between the log of a run with error injection and the log of the error-free golden run. As long as the injected error does not influence the control flow the slice and the difference between the two logs are similar. However, if the injected error changes the control flow, e. g., the error changes the conditional value of a conditional branch, a slice follows the injected error. Whereas the difference between the two logs would contain a mixture of both control flows and not be very helpful anymore.

Listing 9.1 shows a slice of a modified operand error injected into an unprotected application, i. e., an application without any additional mechanisms for detecting hardware errors. The slice consists of a sequence of LLVM instructions. To simplify the presentation, we highlighted the data flow of the error by underlining the registers that were influenced by the injected error. The modified operand is injected into the first operand of the `store` in line 4 via a call to `@bitflip_i32` in the previous line. The error propagates via `load` and `store` through the memory (addresses `%addr1` and `%addr2`) until it is returned from function `@abs` (line 12). Hence, the function call in line 16 returns an erroneous value. The

erroneous return value influences an addition in line 17 and then is used by the loop that starts at line 20. For readability, we omitted all but one loop iteration. In line 27 the erroneous value is finally output by the external function `@printf`.

Implementation of the Slicing

EIS assigns a unique *error ID* to each error injected. Each execution of an arbitrary trigger point is assigned a different error ID. Thus, the error ID identifies one specific injected error. Together with the runtime configuration of an injection, the error ID allows us to deterministically repeat error injection runs. The runtime configuration consists of the seed that controls the possibly used random numbers, the symptoms injected, and the injection mode used. To generate a slice for a given error with error ID e, EIS does a *single* error injection run with debugging support enabled. In this run only the error e is triggered. The output of this run is a binary log containing all instructions of the forward data flow derived from e. EIS then converts this log into an XML representation for further processing.

EIS' debugging support consists of a dynamic data flow analysis. The implementation of the data flow analysis consists of two parts:

Shadow instructions trace the data flow of the injected error through the application.

Logging code logs unique IDs of instructions that take part in the traced data flow.

The shadow instructions and logging instructions are added *after* the error injection instrumentation and only if debugging support is enabled.

Shadow Instructions At runtime, each LLVM register r is shadowed by a register r_S tracing the data flow of the injected error. Depending on whether the current value of r is part of the traced data flow or not, r_S contains one of two values: either `INTERESTING` or `BORING`. `INTERESTING` means the value of r was influenced by the injected error. `BORING` means it was not influenced.

EIS uses shadow instructions to update the shadow values after each instruction that was executed. Therefore, after each instruction a shadow instruction is added. EIS adds shadow instructions and registers as follows:

- Constants are shadowed as `BORING`.
- The shadow value of the result of an unary instruction gets the same shadow value as the operand of the instruction.
- Instructions with an arity of two and higher are shadowed by the disjunction of its operands, that is, if any of the operands is `INTERESTING` the result is also.
- The injected error is shadowed by `INTERESTING`. All other error injection points pass-through the shadow values of the original registers. Thus, they are treated as instructions with multiple operands.

- Loads and stores are shadowed by accesses to a shadow memory. Each byte in memory is shadowed by a shadow value. Initially, all bytes are shadowed by `BORING`. A shadow store writes the shadow value of the stored register into the shadow memory at the address of the original store. Similarly, a shadow load reads the shadow value from the shadow memory at the address of the original load. Our shadow memory implementation is derived from [NS07].
- Functions get one additional argument per original argument carrying the shadow value of the original argument. They also get a second return value representing the shadow value of the original return value. Function calls are instrumented accordingly.

Logging Code EIS adds to every original instruction a log instruction. The log instruction logs the unique ID of the original instruction, if and only if, at least one operand (or argument in case of function calls) of the original instruction has a shadow value of `INTERESTING`.

After running the application, the logged instruction IDs form the slice. In a post processing step, EIS converts the logged IDs with the help of the original LLVM code into the XML representation introduced above.

Evaluation

It is difficult to objectively evaluate the usefulness of slicing in our context. However, we performed a case study with four of the applications introduced in Section 8.5.1 to give an intuition about the usefulness of our approach. For each application, we executed 10 error injection runs in two configurations. For the first configuration, we logged all executed instructions as a *full log*. The second configuration only logged the *sliced log* of the data flow of the injected error.

Table 9.1 shows the results of these experiments. All values for runtime and size are the average of all 10 runs. Logging a slice only is in average 14.48x faster than doing the full log. Data flow analysis needed for slicing introduces some runtime overhead. On the one hand, for the two faster executions (`primes` and `md5`) the slicing is actually slower than writing out the full log. However, the absolute runtime with slicing is still shorter than 0.1s. On the other hand, for long running applications the overhead is more than compensated by the IO overhead of logging all instructions for a full log. The slices are in average 1316.1x smaller than the full log. We believe the space reduction makes it much more feasible to manually inspect the slice than the full log. Automated inspection will also profit from the reduced size.

We inspected slices of the error injection results of ANBDmem-compiler-encoded benchmarks (see Section 8.5.3). We were able to identify several issues that lead to silent data corruptions.

One example is the overflow correction for an ANB-code that we described in [WF07a] and Section 4.2.1. The correction is required to ensure correct

Application	Runtime for		
	full log	sliced log	Speedup
primes	0.017s	0.027s	0.63x
md5	0.020s	0.029s	0.7x
pid	1.691s	0.106s	15.95x
tcas	3.649s	0.092s	39.66x
Average			14.48x

Application	Size of		
	full log	sliced log	Reduction
primes	458kb	4kb	114.5x
md5	616kb	20kb	30.8x
pid	79,000kb	1,879kb	42.0x
tcas	132,000kb	26kb	5076.9x
Average			1316.1x

Table 9.1.: Runtime and space required for a full log compared to a sliced log for four different applications.

integer overflow behavior for computations with encoded values. But for small As it has a vulnerability to exchanged operand errors. We found this issue because the number of silent data corruptions for exchanged operand errors was unexpectedly high. Slices of runs where operands were exchanged revealed the cause of this vulnerability.

For ANBDmem-encoding version management is required (see [WF07b] and Section 4.7). Using the slicing of EIS we were also able to identify a safety-relevant bug in our list-based version management.

9.4. Conclusion

This Chapter introduced our two error injectors FITgrind and EIS that we used to evaluate the error detection capabilities of SEP and CEP, respectively. Both error injectors are generally applicable and can be used to evaluate arbitrary error detection approaches or the resilience of unprotected applications with respect to execution errors. Furthermore, both injectors are hardware-independent and do not require a hardware model.

EIS in addition to error injection also provides forward slicing. Thereby, it facilitates the debugging of error detection approaches. This already enabled us to identify several safety issues in our Compiler Encoded Processing presented in Chapter 8.

Both tools helped in evaluating our encoding approaches and in further improving their error detection capabilities. Results obtained using them where published in several papers [WF07b, SSSF10c, SSSF10a].

10. Related Work

In this chapter, we relate our work to other hardware error handling approaches. Thereby, we focus on techniques for detecting hardware errors, but also discuss approaches for avoiding the occurrence of hardware errors. Detection facilitates the correction of errors. However, we will not present techniques for handling the errors detected by the mechanisms presented.

For determining the capability of error detection mechanisms to detect errors, often error injection is used. We already described the state of the art for error injection tools in Chapter 9.

Chapter overview

Before describing the error detection approaches in the following, we introduce different ways of ordering and classifying approaches to handle hardware errors in Section 10.1. Then, we describe the different error detection approaches ordered by their kind of implementation. The detection of hardware errors can be implemented in

- hardware (see Section 10.2),
- software (see Section 10.3), or
- a combination of both (see Section 10.4).

10.1. Classifying Error Handling Approaches

This section introduces several schemes to order and classify error handling approaches. Therewith, we will order the related work presented in the following sections and our own work.

To implement a fault tolerant system two general strategies exist:

Avoidance: The occurrence of errors is avoided. For hardware errors, this mostly means that hardware is designed in a way that prevents hardware errors. This hardware must not be susceptible to soft errors, which might be caused by radiation, heat, etc., and must not contain permanent errors – or at least the error rates caused by soft and permanent errors should be as low as required for building a reliable system. Also software approaches exist that try to avoid the occurrence of hardware errors by using the hardware in a way that makes the occurrence of hardware errors improbable.

Detection & correction: The occurrence of errors is accepted and not prevented. Instead, approaches are applied that detect errors and cope with them. Furthermore, approaches exist that correct errors without detecting them. We call these *self-correcting*.

This thesis introduces a mechanism that facilitates the detection of hardware errors. We do not provide correction. Thus, our related work presented in the following will focus on the detection of hardware errors – implemented in hardware or software. However, we will also present some approaches that avoid the occurrence of hardware errors or are self-correcting. We will show that these approaches have an insufficient coverage or are not generally applicable.

Every error detection approach is based on some kind of redundancy or a combination of different kinds of redundancy. There are three different types of redundancy used for error detection:

Hardware redundancy: Additional hardware is used to implement the detection of errors. For example, for detecting soft errors, an additional processor could be used. This processor executes a program, function, or instruction a second time. An additional voter compares the results of both processors and detects an error if the results are not equal. Of course, the additional voting step may increase the runtime.

Time redundancy: In contrast to space redundancy, time redundancy uses the same hardware to realize error detection. However, additional runtime is used to detect errors by executing additional code. For example, for detecting soft errors, a program can be executed two times – one replica after the other – and the results of these two executions are compared. A mismatch indicates an error.

Information redundancy: Data is supplemented with additional data that facilitates the detection of errors. For example, data can be stored several times or a parity is added to each data item stored. Any mismatch between the redundant information and the data identifies an error. Checking the consistency of data may increase the runtime.

Encoding is a combination of all three forms of redundancy. Encoding supplements data with additional data whose consistency property facilitates error detection. Thus, information redundancy is used. The signatures expected for encoded values are precomputed at compile time. Therefore, other more safer hardware can be used. Thus, the signature precomputation adds time and possibly hardware redundancy.

Classification by the type of redundancy is too general. Thus, we will order the detection approaches first by their kind of implementation: in hardware, software, or a combination of both. Within these categories we will use the following classification that describes the technique used in general:

Replicated execution: Approaches that use replicated execution use hardware or time redundancy to re-execute the program or parts of it. One re-execution facilitates detection of one erroneous execution. Using more replicas even facilitates the correction of detected errors.

Consistency Checking: For consistency checking, also additional hardware resources or runtime are required. Furthermore, the program executed needs to contain some invariants that can be checked. For example, assertions are an example for consistency checking. Another example is a processor that checks the consistency of the control signals generated during the

execution of an instruction.

Control and Data Flow Checking: Control flow checking ensures that the control flow, that is, the order in which instructions are executed, is valid for the application executed. To realize control flow checking, most approaches extend the program executed with a model of the control flow expected and check if the control flow observed at runtime matches this model. Most approaches check only the correct ordering of basic blocks, but not the correctness of the order of the instructions within the basic blocks.
Similar to control flow checking, also data flow checking exists. Therefore, at runtime, the data flow observed is compared to the data flow expected for the application. A mismatch indicates an error.
Control and data flow checking cannot check for general computation errors. Control flow checking detects only errors that turn control flow into invalid control flow, and data flow checking can only detect errors that disturb the choice and loading of operands.

Arithmetic Encoding: Arithmetic encoding uses information redundancy to detect erroneous modifications of data and also erroneous computation of data. This also requires additional hardware resources because encoded data and programs require a larger amount of memory for storage. Furthermore, also additional runtime is required because arithmetic encoding slows down the protected program. However, no replicated execution (either in parallel on redundant hardware or sequentially on the same hardware) is required.

The AN-/ANB-/ANBDmem-encoding proposed in this thesis is an arithmetic encoding approach that is implemented in software.

10.2. Reliable Hardware

An alternative for handling hardware errors in software – as proposed in this thesis – is to handle them already in hardware. This section aims at showing that building reliable hardware is a complex and expensive task. Therefore, we first present hardware solutions for avoiding the occurrence of hardware errors. Second, we describe hardware-implemented solutions for detecting hardware errors.

Most solutions presented in the following protect only parts of the system, for example, the memory, or the control flow, or a computation. However, every part of the hardware is susceptible to errors and, thus, has to be protected. Hardware approaches that aim at a comprehensive error detection for the whole system usually combine several techniques which makes them complex and expensive.

10.2.1. Error Avoidance

Of course, it would be much more effective to completely prevent the occurrence of hardware errors – soft and permanent ones. Calhoun et. al. in [CCL+08] give an overview of techniques that are nowadays used to produce usable hardware

despite the problems posed by downscaling. For example, the usage of *design-for-manufacturing rules* and *standard cell libraries* shall avoid that circuits contain permanent errors caused by downscaling and are tolerant to soft errors induced by variation, heat, radiation, etc. However, the smaller the feature sizes become the less successful are these techniques. For example, the design-for-manufacturing rules have become so complex that several rules contradict each other. Already today, the resulting hardware is not sufficient for safety-critical applications.

Several other papers present approaches for realizing hardware that is not susceptible to soft errors. In the following, we will present some examples. However, this list of examples is far from being complete because the focus of our work is the detection of errors and not their prevention because it is our opinion that complete prevention is not possible.

Fault-tolerance by design

Avirneni et. al. in [ASS09] present registers that are designed in a way that enables them to tolerate single event transients. However, this approach does not protect logical circuits and busses. Therefore, additional measures are required.

Gate resizing

One approach to make memory and logical circuits tolerant to soft errors is gate resizing. In this approach again larger feature sizes are used to produce the circuits. These circuits are than used with a higher voltage. Using larger feature sizes and higher voltages makes circuits more tolerant to soft errors because the negative impacts of downscaling are reversed. However, it makes no sense to resize whole circuits. In contrast, the approaches described in [WM08b, ZM06] determine gates that are susceptible to soft errors. Only these gates are than resized, that is, produced using larger feature sizes and used with higher voltages than the smaller-sized gates.

Flipflop selection

A transient is said to be latched if it erroneously modifies the state of a flipflop. If a transient induced into a logical circuit by radiation or heat is latched, depends on the timing behavior of the transient induced and the flipflop at which the transient arrives. Joshi et. al. in [JRBS06] present different flipflop designs with different temporal masking capabilities. Thus, depending on the signal properties on the input lines of the flipflop, a flipflop implementation can be selected that increases temporal masking of transients on the input lines as much as possible.

Rao et. al. in [RBS06] propose to combine gate resizing with flipflop selection. Additionally to increasing the feature size of susceptible gates, Rao et. al. determine the timing behavior of the signals at the input lines of flipflops, and choose a flipflop implementation that makes latching an error the most improbable.

Reduced exposure

Weaver et. al. in [WEMR04a, WEMR04b] reduce the impact of radiation by reducing the time that instructions forming a program are stored in vulnerable storage structures. The authors assume that the pipeline is vulnerable and the memory is not. Thus, instructions are removed from the pipeline when long delays are encountered.

All these approaches have in common that they increase the complexity of the hardware design process and make the hardware produced more complex and

expensive. Furthermore, all these approaches only aim at soft errors and protect only parts of the system. None of the approaches targets permanent hardware errors and all need to be combined with an approach to handle errors occurring in memory or on busses.

10.2.2. Error Detection

In the following, we will first present hardware-implemented approaches that detect errors in one single part of the hardware. This is followed by the presentation of several systems that combine different technologies to provide comprehensive error detection for the whole computing system.

Replicated Execution

Replication of functional units

Replicated execution is widely used to detect hardware errors during program execution. Several systems replicate large functional units of the hardware and use voters to detect and often even correct errors. Examples are:

- The *Boeing 777* uses triple modular redundancy for the computing system, the electrical power supply, the hydraulic power, and the communication path [Yeh96]. Thus, one erroneous replica can also be corrected.
- Complete function blocks of *IBM's S/390 G5 microprocessor* are duplicated for error detection [SAC+99].
- The *NonStop Advanced Architecture by HP* at least duplicates all parts of the computing system. If error correction is needed, also TMR, that is, triplication, is supported [BBV+05].

The DIVA checker [Aus99, WA01] replicates a whole processor. However, the DIVA checker that checks the original is a simpler implementation of the original processor. For example, no speculation and out-of-order execution are provided and a smaller frequency is used to operate the replica. For ensuring that the checker nevertheless is fast enough, it reuses results of the original processor, for example, the results of the original's pipeline instruction ordering logic are used to speed up the checker. The DIVA checker is more robust and easier to verify than the CPU checked. Thus, the DIVA checker can be assumed to be less susceptible to errors.

Bower et. al. extended the DIVA checker approach in [BSO05] with a reconfiguration procedure that disables permanently erroneous units in the complex original processor.

Replication using multi-threading

Several papers present CPU designs that use simultaneous multi-threading to execute several copies of a thread and compare the results to detect errors occurring in the CPU. Examples are:

- Active-stream/Redundant-stream Simultaneous Multithreading (AR-SMT) [Rot99],
- the Simultaneous and Redundantly Threaded (SRT) processor [RM00],
- the Simultaneously, and Redundantly Threaded processor with Recovery (SRTR) [VPC02],

- the Slipstream Processor presented in [PSR00],
- Redundant Execution using Critical Value Forwarding (RECVF) [SSSL10], and
- the Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR) [GSVP03].

All these processors execute programs using duplicated redundant threads: one leading and one trailing thread. In contrast to the approaches presented above, the redundant threads are dynamically scheduled and only synchronized with respect to data externalization at the boundaries of the sphere of protection, that is, at the boundaries of the duplicated part of the execution. For all the examples, the sphere of protection comprises the processor including the register file. Thus, all approaches check the consistency of the duplicates before externalizing data to the memory.

The Slipstream Processor [PSR00] reduces one of the threads executed to these instructions that are required for making process. Thereby, the whole system is sped up because, for example, branch predictions from this faster thread can be used to speed up the trailing thread that checks the execution. RECVF [SSSL10] also forwards results from the leading thread to the trailing thread. This speeds up the trailing thread. In RECVF this speed up is used to reduce the overall energy consumption by using a slower and lower-voltage core for executing the now faster trailing thread.

SRTR [RM00] additionally provides a rollback mechanism for recovering from detected errors.

Adaptable replication

The Mixed-Mode Multicore [WCS09] and the Reconfigurable Generic Dual-Core Architecture [KS06] also dynamically duplicate threads using the multi-threading capabilities of modern CPUs. In contrast to the above discussed approaches, these solution provide more flexibility because the user can choose which threads are critical and shall be duplicated and which are not critical and can run unduplicated and faster using only one core.

Yao et. al. in [YWZ04] presents the design of a quad-core processor that dynamically can be configured to execute applications in parallel two, three, or four times. Thus, even error correction and the adaptation onto different levels of criticality are possible.

Instruction replication

Furthermore, a more fine-grained replication can be realized by replicating single instructions as presented in [PF82, NPI07, RLC+08, TMBS10]. All these approaches only protect the execution of the instruction. Additional measures are required, for example, to detect data modifications in memory, the registers, or on the bus. Also errors disturbing the instruction selection are not detectable using solely these approaches.

The approach presented in [TMBS10] additionally duplicates the instruction decoding. However, instruction loading still is unduplicated and thus unprotected.

The approach presented in [NPI07] provides a hardware-implemented selective duplicated execution of critical instructions. This approach only duplicates the execution of instructions that influence variables that the user marked as

critical. While the duplication is done at runtime in hardware, it is determined at compile time which instructions have to be duplicated.

The approach presented in [PF82] not only duplicates the instructions but diversifies the processed data by, for example, using for the second execution operands that are shifted one bit. Thereby, permanent errors and even some design errors become detectable. However, for supporting this, special implementations of the instructions are required.

Replicated execution detects soft errors. However, it is susceptible to permanent errors, for example, in the instruction execution, loading, and decoding. If a permanent error influences all replicas, the error is not detectable. Furthermore, all the replication approaches described above have to be combined with error detection for memory and busses. Some even do not provide error detection within registers.

Consistency Checking

Under consistency checking we summarize all approaches that do not use replicated execution, but instead check consistency properties of an execution. These consistency properties could be already available in an architecture or are explicitly introduced by the checking approach. Of course, replicated execution is a special case of consistency checking because it checks the consistency of the replicas.

Offline selftesting

Offline selftesting of hardware checks the correctness of the hardware itself. Therefore, test patterns are applied to the hardware and the results generated by the hardware are compared to the expected results. Example implementations are presented in [BHP+71, BI86, LMM08, AB09].

Offline selftesting can detect permanent errors and the susceptibility of hardware to soft errors if a soft error disturbs the test execution. However, if a program execution is disturbed by soft errors or newly occurring permanent errors, is not detectable using offline selftesting of the hardware.

Checksums

One widely used consistency checking approach are checksums that detect data modifications occurring during storage and transport of data. The reason for their widespread use in reliable systems is that memory was the first part of systems in which soft errors were observed. Furthermore, due to its regular structure, memory is easier to protect than logical circuits.

To implement checksums, additional data is added to each data word. This information redundancy summarizes the original data word. An example is a single-bit parity that is zero if the original data word contains an even number of bits set to one, and one otherwise. This single-bit parity can detect single bitflips. If enough redundant information is added to each data word, not only detection of multiple bitflips but also correction can be realized. Examples, are error correcting codes (ECC) and IBM's Chipkill [IBM99].

Most checksum approaches provide no detection of computation errors. Thus, they are often combined with other approaches for detecting these errors. For

example, the Argus project [MBS07] combines checksums to detect data modifications with control and data flow checking and additional consistency checks of the functional units of the processor such as the adder circuit. However, for example, arithmetic codes can be used to detect data modifications and computation errors as well. We present hardware implementations of arithmetic codes in the following on page 200.

Latch replication

Nowadays processors are often designed for testability, that is, additional circuits are produced on a chip that are only used for testing purposes. During normal execution these parts of a processor remain unused. The approach presented in [MSZ+05] shows how these testing circuitry can be reused for the detection of soft errors during the execution of programs. Therefore, the existing test circuits are used to shadow latches with a replicated latch. Thereby, modifications occurring to one of the two latches become detectable to an additional checker.

The approach for reusing testing circuits presented in [MSZ+05] is extended with adaptability in [ZMM+06]. Only critical applications use the shadow latches for error detection. For non-critical applications, the shadow/testing latches remain unused. Thereby, the energy consumption of the circuit is reduced for non-critical applications.

Execution checking

While the previous approaches detected permanently faulty hardware or errors modifying stored data, the following approaches check consistency properties of the execution of a program. These approaches especially aim at detecting errors in the logical circuits that control the processing of data.

Ganesh et. al. in [GSS06] present a hardware-implemented checking of the control signals that are generated within the CPU during program execution. If these do not match the instruction scheduled, an error is detected.

In [YMOS07] checker implementations for adders and multipliers are presented. These checkers do not reimplement the circuits but use faster consistency checks. For example, the multiplication $a * b$ is checked by computing $((a \mod C) * (b \mod C)) \mod C$ and comparing it to $(a * b) \mod C$ where C is a small hard-coded integer constant that is larger than one. Thus, the implementation of the check is much simpler and faster than the operation checked.

[NDMF97] presents a similar approach that extends every logical circuit such as an adder or multiplier with a parity prediction. Thus, values stored with a parity for detecting data modifications can be processed by these circuits and the processing is checked by the parity prediction. [GEJL10] extends this approach with bit interleaving. The reason is that multiple bit errors have become more probable and thus might disturb the parity prediction as presented, for example, in [NDMF97]. Thus, the parity prediction of different computations is interleaved so that if multiple bits flip, only one of each operation is affected. However, [GEJL10] presents this approach only for an 8-bit adder.

Reddy et. al. in [RAZR06] present an approach that checks the correctness of an execution by *asserting microarchitectural "truths"*. They present two example functionalities for which they implemented such architectural assertions:

- Register Name Authentication (RNA) for the rename unit and

- Timestamp-Based Assertion Checking (TAC) for the issue unit of the processor.

In [RR07], Reddy et. al. describe how redundant execution traces within the execution of an application can be used for consistency checking. On the first execution of a trace, a checksum is computed which hashes non-data-dependent parts of the trace, for example, the order of the instructions and which registers are used. For further executions of the trace, the checksum is recomputed and compared to the value expected which was computed when executing the trace for the first time.

To increase the coverage provided by execution checking, Reddy et. al. in [RR08] combine several of their execution checking approaches. Thereby, the error detection capabilities of the whole system are improved considerable. However, still only a coverage of 83% of the non-masked hardware errors is achieved.

Value checking

Several approaches check the consistency of the values computed by a program. For example, [RCMM07] dynamically determines a model for result values expected for instructions. If these expectations are not met, the pipeline is flushed assuming that a soft error led to the mismatch and a reexecution will correct the error. [DZ07] presents a similar approach. However, it is checked that the variance of an instruction's result matches the limits within which it stays most of the time.

[PSC+06] describes the automated derivation of hardware-implemented error detectors. The mechanism described derives error detectors for a program and generates a hardware implementation executing these. A detector checks if a variable at a certain point during the program execution has an expected value, for example, is element of a certain interval or set, or is bigger than its predecessor etc. To derive these detectors, the program is executed with a representative set of inputs and certain previously chosen variables are monitored. Then for each monitored variable an appropriate detector class fitting the data seen is chosen and appropriately parameterized. To summarize, the authors present an approach that automatically derives assertions and provides a hardware-implementation checking these assertions.

If errors do not result in values that violate the checked properties, errors are undetectable for these value checking approaches.

Symptom detection

ReStore [WP06] checks an execution for behavior that indicates that an error disturbed the execution. For example, in the following cases the authors assume that a hardware error occurred:

- if memory access and alignment exceptions occur,
- if a mismatch between a branch prediction that has a high confidence level and the branch taken is detected, or
- if a cache miss occurs.

If such a symptom is detected, the processor is rolled back to a checkpoint. If the symptom occurs again in the following reexecution, the authors assume that no error was the cause and continue the execution or in case of an exception let the exception propagate.

This approach is susceptible to permanent errors. Furthermore, errors that do not cause the symptoms checked, but just silently corrupt output are undetectable using this approach.

Control and Data Flow Checking

Control flow checking

Control flow checking provides means to recognize invalid control flow for the program executed, that is, execution of sequences of instructions which are not permitted for the binary executed. This can be implemented in hardware and software. The following approaches are implemented in hardware: [LG04, BUEA06]. All of them are signature-based and protect the control flow between basic blocks but not within basic blocks. Therefore, for each program a model is generated that describes the allowed control flow. At runtime, the control flow is monitored and if it contradicts the model, it is assumed that an error occurred.

Control flow checking does not detect data modifications or computation errors that do not result in an invalid control flow.

Data flow checking

The already mentioned Argus project [MBS07] also applies this kind of control flow checking. Furthermore, Argus also determines for each application a data flow model and checks the data flows occurring at runtime for consistency with this model. This techniques is more detailed described in [MS07].

Control and data flow checking cannot detect errors that result in erroneous program state but not in invalid control or data flow.

Hardware-implemented Arithmetic Codes

Arithmetic codes can be implemented in hardware and software. For some codes, such as the Berger codes, only a hardware implementation makes sense as we explicated in Section 3.1. Indeed most of the hardware implementations of arithmetic codes use Berger codes. Examples are described in [LTRN92, MR98, DT99, LOBR09]. While [LTRN92] and [LOBR09] use plain Berger codes, [MR98] uses Dong's code and [DT99] uses a Bose-Lin-code. Both are special Berger codes that can be implemented more efficiently.

The only hardware-implemented approach that does not use a Berger code that we know of is the STAR computer [AGM+71] that uses a Residue code with residue 15 for detecting execution errors.

Frameworks & Systems

The approaches presented so far protect only single parts of a system. For detecting as much hardware errors as possible, it is necessary to combine different approaches. In the following we present approaches that strive to protect as much of a computing system as possible

The Reliability and Security Engine (RSE) The Reliability and Security Engine (RSE) [INKM05, IKP+07] is a common processor-level framework that provides application-aware reliability and security. This framework allows to build computing systems that provide detection, masking, and recovery for errors caused by accidental failures and malicious attacks as well. Therefore, RSE integrates the following hardware-implemented detection and masking mechanisms:

- hardware-implemented error detectors that like assertions check if variable contents are reasonable [PSC+06],
- algorithms for efficiently placing these detectors [PKI05], and
- hardware-implemented selective duplicated execution of critical instructions [NPI07].

Implementations of RSE exist for the pipelines of the DLX and the Leon3 processor.

High-end mainframes/servers Several high-end mainframes and servers apply various mechanisms for detecting hardware errors. For example, Hewlett Packard's NonStop computer systems [BBV+05] uses the following techniques:

- redundant execution using non-lock-stepped multicores that execute two or more copies of one thread,
- triple or double modular redundancy for important parts such as network and IO-adapters,
- dynamic reconfiguration, that is, failed components are not used anymore, and
- checksums and duplication to detect memory errors.

According to [BBV+05], no single error stops the execution of a NonStop computer system.

Further similar systems that strive to provide a comprehensive error detection are IBM's G4 [SG98], G5 [SAC+99], and z10 [Web08].

Argus The Argus project [MBS07] combines checksums to detect data modifications with control and data flow checking and additional consistency checks of the functional units of the processor such as the adder circuit.

10.2.3. Summary

Most of the hardware-implemented approaches that we presented in this section protect only parts of a computing system. Providing comprehensive error detection for a computing system completely in hardware is a daunting task that requires to combine several approaches. The reason is that errors can occur in every part of a system, that is, during storage and transport of data and also during the processing of data. While it is relatively easy to protect large memories and also busses, protecting logical circuits and all transfers and storing of data within those circuits is a complex task.

Developing such custom hardware is an expensive process. Once developed, such hardware would most certainly be exclusively used for safety-critical tasks. Furthermore, because of its high initial costs, it would for a long time not be replaced with newer and, thus, faster and less power-consuming hardware.

10.3. Handling of Hardware Errors in Software

The most important argument against hardware solutions are the high development costs and the restricted market. Due to economic pressure, the trend for critical and safety-critical systems is to use commodity systems instead of custom hardware. Thus, software approaches for handling hardware errors are required.

Software approaches provide much more flexibility. They are easier to apply, cheaper and faster to develop, and allow to use most recent, i. e., more powerful, hardware. Furthermore, software-based solutions facilitate mixed-mode systems. These systems execute both safety-critical and non-critical applications on the same (commodity) hardware. Thereby, they improve hardware utilization.

Software-implemented error detection mechanism allow to use up-to-date hardware in safety-critical systems which require certification. The precondition is that the error detection probability of the mechanism is independent of the hardware used. This can, for example, be provided by using arithmetic codes such as the AN-code.

In the following, this section describes approaches that avoid or detect hardware errors and are implemented completely in software.

10.3.1. Error Avoidance

Avoidance of faulty hardware

Detouring [MS08] compiles software in a way that the usage of faulty parts of the CPU is circumvented. For example, faulty data bypasses in the pipeline are avoided using instruction reordering or the insertion of nops. However, in an example implementation for a simulated RISC CPU, for only 42% of the hardware such a detour could be provided. Especially, for often used operations such as additions no (efficient) detour is available. Furthermore, this approach requires that the faulty parts of the hardware used are known at compile time.

Regular heat distribution

Yang and Orailoglu [YO09] observe that asymmetric register usage leads to an irregular heat distribution on the chip. This reduces the reliability of circuits because heat – especially having different temperatures at different parts of a chip – modifies the electrical properties of gates [Bor05]. Thus, Yang and Orailoglu [YO09] present a compiler that determines a register mapping that equally distributes heat generation over the registers. Therefore, it uses application-dependent execution profiles.

If safety-critical and non-safety-critical applications run on the same processor, the compilation for regular heat distribution has to be applied to all applications.

Otherwise, non-safety-critical applications could heat up parts of the register set.

All these avoidance techniques do not cover the complete processor. Thus, they are not sufficient for safety-critical systems. However, they could be combined with the error detection approaches presented in the following. Using these avoidance techniques additionally to detection techniques can increase the availability of systems.

10.3.2. Error Detection

In the following, we present approaches that as the encoding approaches proposed in this thesis aim at detecting hardware errors in software. For the correction of the errors detected, other methods have to be used.

Replicated Execution

Several papers present replication that is implemented in software. We can distinguish these approaches by the point in time when the replication is applied:

- source code is transformed into replicated source code before the actual compilation is executed, or
- the replication is applied to an intermediate, assembler-like code during compilation, or
- dynamic binary instrumentation is used, that is, the replication is done at runtime.

Source-to-source transformation

Rebaudengo et. al. [RRTV99, RRVT01] apply duplication to C source code. They replicate every instruction – including statements controlling the control flow such as if statements. Furthermore, they insert instructions that check that the two versions of a variable are equal after each use of the variable. The intention is to stop the propagation of errors as early as possible. These checks are part of the resulting modified program. They are executed by the same hardware that executes the program.

Benso et. al. [BCPT00] analyze for each variable of a C or C++ program which impact it has on the reliability of the program. Therefore, the lifetime of each variable and the number of variables depending on this variable are determined. The longer a variable is alive and the more other variables depend on it, the more impact does this variable have. Benso et. al. try to reduce this impact, for example, by reducing the lifetime of variables with high impact. Therefore, they use instruction reordering. Furthermore, they duplicate the computation of safety-critical variables. Which variables are safety-critical is either directly chosen by the user or he specifies the percentage of variables whose computation shall be duplicated. In this approach also, the consistency checks are added to the program and are executed on the same hardware.

Nicolescu and Velazco [NV03] also perform a source-to-source transformation of C code for duplicating all instructions. Furthermore, they apply a data

flow analysis with the objective that consistency checks are not performed on intermediate variables. In this approach also, the consistency checks are added to the program and are executed on the same hardware.

In contrast to the software replication approaches presented above, Wang et. al. [WsKWY07] parallelize the execution of the replicas. While, the approaches presented above execute original instructions and duplicates in one thread, Wang et. al. generate two threads and provide means to synchronize these threads with respect to memory access and externalization of data. Synchronization always includes to check the consistency of both replicas. However, as for the previous approaches, the consistency check is part of the duplicated software and executed on the same hardware.

Transformation of intermediate code

While the approaches presented so far apply replication to the high-level source code of programs, the approaches presented in the following transform some kind of simpler, assembler-like intermediate code.

SWIFT [RCV+05a] is a compiler extension working on intermediate code. It duplicates the instructions of a program and adds comparisons of the duplicates to detect execution errors. However, memory is not duplicated because the authors assume that memory is protected by other means such as ECC. This leads to several windows of vulnerability. For example, when the program writes a variable to memory, both replicas of the variable are compared. If they are equal, one of them is written to memory. However, the variable's content could be modified undetectable between the check and its storage in the ECC-protected memory.

To provide also error correction, the SWIFT compiler was extended with SWIFT-R [CRA06]. SWIFT-R uses triple modular redundancy instead of duplication. Having three replicas facilitates the detection of errors that disturb only one of the replicas. In SWIFT-R, comparing the three replicas and choosing the correct value in case of an error is added to the program and executed on the same hardware as the protected program.

Pattabiraman et. al. in [PKI07] use the LLVM compiler framework to implement a compiler that duplicates the computation of critical variables. A variable is considered critical by the authors if it exhibits high sensitivity to random data errors. The authors present an approach for determining which variables are critical in [PKI05]. For every critical variable, a backward slice is determined by static analysis. All such backward slices are duplicated and the duplicates are compared to the originals for detecting errors.

Lyle et. al. in [LCP+09] instead of duplicating the backward slices in software, generates a hardware-implemented version of the backward slice. Thereby, the overhead at runtime is reduced. However, expensive special hardware is required.

ESoftCheck [YGS09] uses the SWIFT-approach, that is, redundantly executes the instructions of a program within one thread. However, additionally a data flow analysis is done and based on its result unnecessary checks between copy and original are removed. Furthermore, unnecessary copies are removed. Thereby, the register pressure is reduced.

The system of Dimitrov et. al. presented in [DMZ09] has the objective of detecting hard and soft errors in graphics processors. It provides three variants of replication:

time redundancy two threads execute the same functionality and are executed sequentially

hardware redundancy two threads execute the same functionality and are executed parallelly on different cores

interleaved one GPU-thread contains for each instruction a duplicate.

All three mechanisms are combined with memory protection mechanisms. However, no automation is provided by Dimitrov et. al.. The replication has to be done by hand.

Shoestring [FGAM10] combines a symptom-based hardware error detection similar to ReStore [WP06] and compiler-implemented duplication of instructions as proposed by SWIFT [RCV+05a]. All instructions that despite symptom-based detection might produce user-perceptible failures are duplicated.

Duplication at runtime

Spot [RCA+06] executes each instruction two times. Therefore, it uses dynamic binary translation provided by PIN [LCM+05], that is, the duplication is done at runtime when executing the application. The application of redundancy is adaptable depending either on the register used or on the part of the code executed. This design is based on the observation that different registers and parts of code are differently susceptible to silent data corruption.

The Chameleon/ARMOR project [BSW+00, KIBW99] provides replication to detect execution errors in distributed systems. Therefore, the functionality of a critical component of a system is executed multiple times on different hardware nodes. The results of these replicas are checked for consistency. Due to the high level at which Chameleon/ARMOR replicates the execution, no modification of the software executed is necessary.

Note that for all those approaches which are based on redundant execution of instructions or whole programs, it is not possible to provide guarantees with respect to permanent hardware errors. Furthermore, all approaches presented here execute the consistency checks that compare the results of the redundant execution on the same unreliable hardware. Thus, the checks themselves are susceptible to execution errors.

Diversified versions

To overcome the problem that permanent errors might corrupt all replicas equally, the system presented in [Joc02] generates two different versions of a program with the aim that hardware errors do not result in the same error. For example, additional tests are introduced modifying the code layout, the register allocation is changed, basic blocks are split etc. The resulting two versions run in parallel and their results are compared. However, still the check for errors itself is susceptible to errors. Furthermore, permanent errors might remain undetected if they influence both replicas equally. This could, for example, happen for a permanently faulty addition instruction because the instructions themselves are not diversified.

Orchestra [SJGF09] similarly to the approach presented in [Joc02] provides different versions of a program. However, the motivation is not safety but security. Attackers are not able to successfully run buffer overflow attacks on Software that is diversified by Orchestra. Orchestra uses diversified redundancy. Thus, it is also able to detect transient hardware errors. However, Orchestra can also not detect hardware errors disturbing the consistency check and might not detect permanent errors in single instructions because the latter ones are not diversified.

Consistency Checking

ABFT: Algorithm-based fault tolerance

Algorithm-based fault tolerance (ABFT) [HA84, SM04] and self-checking software [WB97, BLR90] use invariants contained in the executed program to check the validity of the results generated. This requires that appropriate invariants exist. These invariants have to be designed to provide a good failure detection capability and are not easy – if not impossible – to find for most applications. Mostly, invariants are presented for mathematical problems. For example, [ZL09] presents an ABFT scheme for the elliptic curve cryptography algorithms.

Protocol models

The Chameleon/ARMOR project [BSW+00, KIBW99] additionally to replication uses consistency checking. Therefore, the developer of a system provides information which types of reply message are valid for a request. At runtime, the system checks if all request/reply pairs are valid. Chameleon/ARMOR does not provide consistency checks for computations. As long as a component generates a reply message that matches the message received, errors in the computations producing the reply message are not detectable by Chameleon's consistency checking.

Symptom-based error detection

SWAT [LRS+08] detects non-masked hardware errors by checking if one of the following symptoms that can be caused by hardware errors can be observed:

- occurrence of a fatal trap that would normally lead to a crash of the application or the operating system,
- the application exits with an abnormal exit code,
- the application or the operating system hangs, and
- abnormally excessive operating system activity.

If one or more of these symptoms occur, a diagnosis process is started to check if the cause was a hardware error. If a hardware error does not result in any of the symptoms checked, it is undetectable by SWAT.

In [SLR+08] SWAT is extended with error detection that uses automatically generated assertions. These assertions are generated for each store and check that the value stored is within a certain range. This increases the coverage provided by SWAT. However, still errors that neither cause a symptom nor lead to a value leaving its range of expected values are not detectable.

Shoestring [FGAM10] also uses symptom-based hardware error detection and combines it with the duplication of instructions to ensure error detection also for those parts of an application that are not probable to cause any of the symptoms

checked. Thus, Shoestring is vulnerable to permanent errors because these might not cause symptoms and might remain undetected by the duplication used.

To summarize, consistency checking is only as good as the invariants used or symptoms are. These have to cover as much of the executed software as possible. If good invariants or symptoms are available, depends on the application executed. Combining consistency checking with replication improves the coverage. However, permanent errors still might remain undetected.

Control Flow Checking

Control flow checking as for example described in [BCN+01, OSM02, GRSRV03, BKIL03, VHM03, BWWA06, VA06, SMF06] and also used in the Chameleon/ARMOR project [BSW+00, KIBW99] provides means to recognize invalid control flow, that is, the execution of sequences of instructions that are not permitted for the executed binary. Therefore, the compiler generates a model of the control flow expected for the compiled program. At runtime, the control flow is observed and compared to the expected control flow defined by the model.

All approaches presented in [BCN+01, OSM02, GRSRV03, BKIL03, VHM03, BWWA06, VA06, SMF06] provide only the checking of control flow between different basic blocks. Therefore, a signature is assigned to each basic block. The model describing the control flow of an application is, for example, an automaton that contains a node for each possible basic block. This node is marked with the signature of the basic block represented by this node. At runtime, a watchdog or the application itself checks that the order of the basic blocks executed matches this automaton.

None of the approaches cited above checks the control flow within a basic block. Using these approaches this can only be achieved by putting each instruction into its own basic block. To the best of our knowledge, only ANB- and ANBD-encoding approaches such as the Vital Coded Processor [For89], SES [MSVZ07, SMM10a], our Software Encoded Processing (see Chapter 7), and the ANB- and ANBDmem-encoding of our Encoding Compiler (see Chapter 8) provide direct checking of the control flow within basic blocks without splitting blocks and between basic blocks.

Control flow checking can only detect erroneous control flow that is, for example, caused by an erroneous modification of the instruction pointer. However, if a fault modifies a computation within a program or a value stored or transfered by the program, this is not detectable using control flow checking. The reason is that these faults and the resulting errors do not lead to control flow that is not valid. Even taking the wrong branch of a conditional jump is not detectable because the resulting control flow is consistent with the control flow model. In contrast, ANB- and ANBD-encoding provides detection of computation errors, control and data flow errors.

Software-implemented Arithmetic Codes

Instead of duplication, or additionally, arithmetic codes can be used to detect errors. We already introduced arithmetic codes in Chapter 3. Applying an arithmetic code requires to modify the program and also the data processed. This is called *encoding*. Software-implemented arithmetic encoding mostly uses AN-codes and enhancements of AN-codes such as the ANB-code.

AN-encoding

ED4I [OMM02], uses a source-to-source transformation to generate a second version of a program. This version does not process the original data but A-multiples of it. All results of duplicate instructions have to be A-multiples of the original results. The AN-encoded replica and the original are executed in parallel. For each output variable, ED4I checks that the result of the AN-encoded version is an A-multiple of the unencoded version's result.

However, whenever a program contains logical operations, the authors choose a factor A which is a power of two to make these operations encodable. Thereby, they reduce the detection capabilities immensely. The resulting code cannot detect bitflips in the higher order bits of data values. However, these bits contain the original functional value. See Chapter 5 for an evaluation of different As with respect to their error detection probability. In contrast to ED4I, our Encoding Compiler is able to encode also logical operations using an A that is not a power of two.

Furthermore, the developers of ED4I do not discuss the overflow problems occurring for AN-encoded arithmetic operations that we pointed out in [WF07a] and Section 4.2.1. Over- and underflows in arithmetic operations are not conserved when AN-encoded values are used, e. g., in an addition. If for example the result of an addition overflows, ED4I will most probably detect an error. This is a false positive because the C standard expects over- and underflows to form a ring of the unsigned integers. For example, the addition of two 32-bit integers a and b is expected to result in $a + b \mod 2^{32}$. For signed integers overflows are also required to be correct because the addition of a negative number in the end results in an overflow in its unsigned representation. For the details, see Section 4.2.1. In contrast to ED4I, we provide a ANSI-C conform encoded implementation of addition, subtraction, and multiplication.

TRUMP [CRA06] also uses an AN-code combined with an unencoded replica. In contrast to ED4I, TRUMP is applied on an assembler-like intermediate code. Every instruction is duplicated and the duplicate is AN-encoded whenever possible. TRUMP AN-encodes only operations that easily can handle encoded values such as additions and subtractions. In contrast to our approaches, bitwise logical operations remain unencoded. As in ED4I, TRUMP also ignores the over-/underflow issue.

In contrast to ED4I, TRUMP applies encoding only to registers and not to memory. In the end, that leaves supposedly only small parts of applications which are AN-encoded. As should be expected, the error injection experiments presented in [CRA06] show a non-negligible amount of undetected failures for most of the tested applications.

Our measurements (see Section 8.5.3) have shown that even complete AN-encoding as provided by our Encoding Compiler still does not detect a significant amount of errors. However, AN-encoded programs exhibit smaller slowdowns than the more safer ANB- and ANBDmem-encoded versions.

ANB- and ANBD-encoding

Forin's Vital Coded Processor (VCP) [For89] ANBD-encodes an application on source code level. Thereby, in addition to data modifications and operation errors, VCP also detects the usage of wrong operands, the usage of wrong operators, and lost updates. VCP provides a predictable and high error coverage. The achievable degree of safety can be influenced by the choice of A. Furthermore, VCP also detects errors introduced by the compiler or linker because encoding is done on the source code level.

VCP has the following disadvantages that restrict its use for execution of general software on commodity hardware:

- The complete data flow of the encoded program has to be known before the execution to be able to precompute the signatures of all output variables. This excludes the usage of dynamically allocated memory and function pointers.
- Special hardware is required to encode input variables, to store signatures, and to check the signatures of output variables.
- Encoding of control flow statements – especially nested ones – is a rather complex task because several timestamps and check values have to be considered. Furthermore, upper bounds for loop iteration counters are required to prevent overflows of the code words due to the iteration counter. Therefore the version counters have to be set back to zero when reaching the bound. This increases the encoding complexity. For the VCP, this complex encoding of control flow statements is not automated.

The description of VCP in [For89] is rather incomplete. Apart from an encoded addition, an encoded if statement, and the encoding of a single non-nested while loop no other encoded operations are described. Furthermore, the level of automation of the encoding remains unclear. It seems that encoded versions of operations such as addition exist and are automatically applied. We assume that the control flow on the other hand is hand-encoded.

SES [MSVZ07, SMM10a] also uses ANBD-encoding. SES executes two ANBD-encoded copies of a program redundantly. The copies are diversified by using different As. However, SES does not support dynamically allocated memory and does not provide the correct over- and underflow behavior for additions, subtractions, and multiplications. Furthermore, the description of SES remains also rather incomplete. Only the implementation of additions and subtractions, and encoded if statements is described. We do also not know if the encoding, which is applied at source code level, is completely automated or if parts of it have to be done by hand.

Steindl et. al. in [SMM+09, SMM+10b] present the implementation of SES-encoded programs on FPGAs. The objective is to provide a faster hardware-implemented ANBD-encoding solution. However, still only encoded additions and subtractions are described and these encoded versions do not provide a

correct overflow behavior. The authors do not demonstrate the encoding of any other operation.

Siemens obtained a patent [KSS02] for a diversified DMR system. The system executes once the original application and once an encoded version of the application. The encoded version uses the following encoding relation $x_c = -A * x_f - 1$, that is, all variables are ANB-encoded with $-A$ and the signature -1. Thus, exchanged operands are less probable to be detected because all variables have the same signature and can be exchanged with each other undetectably. The patent does not describe the implementation of encoded operations.

For none of the described ANB- and ANBD-encoding approaches (VCP, SES, diversified DMR by Siemens), an evaluation of the error detection capabilities using some kind of error injection was published.

Furthermore, for VCP and the diversified DMR by Siemens no runtime measurements were presented. For SES, the authors showed that their encoded addition on an embedded system has a slowdown of 3.5. For completely encoded applications the time needed for execution was not evaluated.

In contrast to [For89, MSVZ07, SMM10a, SMM+09, SMM+10b, KSS02], we presented extensive measurements of the runtime overhead and the error detection capabilities of our encoding schemes. Furthermore, to the best of our knowledge, we are the first that use and describe the encoding of a complete instruction set in a detailed manner. The publications [For89, MSVZ07, SMM10a, SMM+09, SMM+10b, KSS02] only describe additions, subtractions, if statements and simple non-nested loops.

10.3.3. Error Correcting Software

Assertion enforcement

MASK [CRA06] applies so-called assertion enforcement. The aim is to mask errors. For example, if it is known that the upper 33 bits of a value have to be zero, MASK ensures that these bits are indeed zero by explicitly zeroing them. Therefore, the code is extended at compile time with instructions that enforce invariants that can be found using static analysis. The measurements presented in [CRA06] show that MASK mostly reduces the amount of silent data corruptions. However, up to 13% of the error injection runs produced silent data corruptions despite MASK. Even worse, some applications produced more silent data corruptions with MASK than without.

Self-correcting software

As described in [dKS09] error mitigation is not necessary for many algorithms because they are self-correcting, that is, can tolerate errors. For example, algorithms using simulated annealing or the Monte Carlo method are inherently error tolerant. However, the authors of [dKS09] draw from their error injection experiments the conclusion that the error-tolerance provided by the algorithms is not sufficient.

Sloan et. al. in [SKKR10] transform programs into more error-tolerant versions by turning the solved problem into an optimization problem. The algorithm for solving the optimization problem is itself able to correct execution errors.

However, the authors restrict the error model to errors in the floating point unit of the processor. For all other errors, for example, in integer units or logical circuits controlling program execution, no error detection is guaranteed.

As consistency checking, using the self-correction abilities of programs and assertion enforcement depend on the application executed. No guarantees can be provided with respect to the error detection and correction capabilities. Furthermore, self-correction is only provided by a restricted set of algorithms.

10.3.4. Summary

The error avoidance techniques presented in Section 10.3.1 provide no sufficient coverage of the hardware for their usage in safety critical systems. The same applies to the error correcting software presented in Section 10.3.3.

To summarize the software-implemented error detection techniques presented in Section 10.3.2:

Replicated execution is, in contrast to arithmetic encoding, susceptible to permanent hardware errors that might influence all replicas equally. This issue can be overcome by using diverse versions of the software executed. The question is how diverse the different versions have to be. For example, some approaches change the program layout. However, for this example, a faulty addition still could go unnoticed. Using encoding to diversify the replicas removes this issue.

Consistency checking is application dependent. Only if the application that shall be protected provides enough consistency properties that can be checked or is prone to produce distinctive symptoms in case of an erroneous execution, consistency checking can successfully be used to detect hardware errors.

Control flow checking is not able to detect errors that solely disturb the computation of values, but do not lead to an invalid control flow that is not allowed for the program.

Software-implemented arithmetic codes provide error detection that is independent of the application. The codes always significantly reduce the amount of silent data corruptions observed. However, the error detection capabilities and overhead depend on the code chosen. In contrast to our encoding techniques, encoding techniques previously presented were incomplete and not well described and evaluated.

10.4. Approaches Combining Hardware and Software

Last, we present in this section approaches that are implemented in hardware and software as well. All approaches presented in the following try to implement as much of the detection in software. Only parts they cannot protect in software are protected by hardware solutions.

10.4.1. Error Detection

Hu et. al. [HLD+05] implement duplication on assembler code. To control the amount of duplication, the user provides an upper bound for the increase of the length of the program schedule. In contrast to the previously presented approaches, Hu et. al. use a hardware extension for consistency checking.

SWIFT duplicates all instructions and registers, however, not the memory. Thus, when loading and storing values, these values might be modified undetectably. To overcome this issue, Reis et. al. [RCV+05b] combine SWIFT with hardware support for interfacing with the non-duplicated memory. Such a hardware implementation of the memory access can ensure that

- on a load the parity protecting data in memory is checked and the value safely is duplicated, or
- on a store the duplicates are checked for equivalence and the value is written safely to memory – including parity generation.

However, this hardware implementation of the memory access has to be protected from errors.

Note that also the encoding approaches VCP [For89] and SES [MSVZ07, SMM10a] implement the code checking in hardware. However, they fail to describe how this check is realized so that it is not susceptible to errors.

The code checking for our encoding approaches could also be implemented in hardware. However, we proposed in Section 8.4 also a software implementation that is made reliable by executing it several times on diverse hardware.

Bolchini et. al. in [BMR+08] describe a system that implements as much as possible of the hardware error detection in software. Only parts that the authors cannot cover with their chosen software approaches are protected by hardware means. The authors use replication and control flow checking in software. However, this does not facilitate the detection of control flow errors that occur in both replicas and disturb the execution within a basic block. To detect such errors, the authors replicate the instruction pointer in hardware.

10.4.2. Summary

As stated earlier, software-implemented detection of hardware errors is more flexible and does not require expensive, special hardware. However, most of the software-implemented approaches presented in Section 10.3 require some kind of check. Often this check itself is susceptible to execution errors.

Other approaches, for example SWIFT, combine different protection mechanisms. In case of SWIFT this is replication and ECC. However the transition between these different protection mechanisms also has to be protected.

The above described approaches that combine hardware- and software-implemented detection overcome these issues. However, the approaches described do not cover all possible hardware errors because they are using replication and control flow checking. Thus, they are still susceptible to permanent errors.

10.5. Conclusion

This chapter first demonstrated that extensive research exists that aims at detecting hardware errors in the hardware itself. However, protecting a whole system using these approaches is cumbersome and expensive because each approach aims only at parts of the whole system. Thus, several different approaches have to be combined safely. The resulting safe hardware is expensive and inflexible because due to its costs it will only be used for safety-critical tasks and will be replaced seldom.

On the other hand, many software approaches exist. Of these approaches arithmetic codes provide the most comprehensive error detection capabilities because they independently of the hardware used detect

- transient and permanent errors that
- modify data during transport and storage, or
- disturb computations including control and data flow.

In contrast to existing software-implemented approaches that use arithmetic codes,

- we demonstrate the AN-,ANB- and ANBD-encoding of a complete RISC instruction set,
- we support the encoding of arbitrary control and data flow – including data flow that is not statically predictable,
- our encodings are fully automated, and
- we presented extensive evaluations comprising runtime measurements and error injection results.

11. Conclusion

This chapter summarizes the contributions of this work and provides a brief outline of possible future work. Furthermore, we give an overview of publications, patent applications, and granted proposals resulting from the research presented in this thesis.

11.1. Contributions

Encoded operations

To the best of our knowledge, we are the first ones that present AN-, ANB-, and ANBD-encoded versions of the following operations:

- addition and subtraction with ANSI-C-conform over- and underflow behavior,
- multiplication,
- division,
- modulo,
- comparisons,
- bitwise logical operations,
- casts,
- shifts, and
- unaligned loads and stores.

Previous work [Rao74, For89, OMM02, RCV+05a] only presented encoded additions and subtractions with over- and underflow behavior that is not ANSI-C-conform. The encoding of further operations was not presented at all.

Dynamic signatures

Furthermore, none of the previously presented ANBD-encoding approaches [For89, MSVZ07, SMM10a] is able to encode dynamically accessed memory for which the access pattern is not known at encoding time. For facilitating the encoding of dynamically accessed memory, we developed dynamic signatures that are determined at runtime. Using these dynamic signatures our encoding approaches SEP and CEP support dynamically accessed memory.

SEP

Our first encoding approach, Software Encoded Processing (SEP), takes previously unencoded binaries and ANBD-encodes them at runtime. To the best of our knowledge, SEP is the first software-based implementation of ANBD-codes that is able to protect programs whose source code is not available from execution errors.

CEP

Our second encoding approach, Compiler Encoded Processing (CEP) similarly to VCP [For89] and SES [MSVZ07, SMM10a] applies encoding at compile time. However, in contrast to VCP and SES, CEP supports the encoding of

programs whose data and control flow cannot be predicted completely at compile time. Furthermore, in contrast to VCP, CEP is fully automated. We have no information about the tools existing for SES.

Encoding vs redundancy

Additionally, CEP provides different levels of safety by supporting encoding transformations using different arithmetic codes (AN-, ANB-, or ANBDmem-code). We compared these different encodings applied by CEP to each other and to redundancy-based error detection. As expected, we observed that the higher the error detection probability is for an approach, the higher is the increase of the runtime required. However, while the probability of detecting errors grows exponentially, the increase of the runtime grows only linearly.

Symptom-based error injection

For evaluating arbitrary error detection approaches – including ours –, we developed the symptom-based error injectors FITgrind and EIS. In contrast to the single bitflip error model used in recent research papers for evaluating error detection approaches, FITgrind and EIS provide a broader error model including data and control flow errors. Furthermore, EIS provides debugging support for error detection approaches by providing an execution log that only contains instructions that were influenced by an injected error.

11.2. Future Work

Optimization

The slowdowns of encoding have decreased dramatically with the usage of CEP instead of SEP. In the future, further optimizations can be researched to further reduce the overhead induced by encoding. The following optimizations can be analyzed:

- If we can verify for an execution of an operation that no over- or underflow can happen for this execution, an encoded version of this operation without the expensive over- or underflow correction can be used.
- Currently, the encoded program contains one table with precomputed values for each bitwise logical operation supported. These can be reduced to one because for example a `nand` is sufficient to implement `or`, `and`, and `not`. This reduces the size of the encoded binary. This is especially important for embedded systems, which have restricted resources.
- Currently, multiplication and division use expensive 128-bit operations because intermediate results of these encoded operations might require more than 64 bits. However, the expensive 128-bit operations should only be used when necessary. If A is chosen well (see Appendix A), overflows of the encoded values result in invalid code words. Thus, detecting overflows and if so using a non-overflowing 128-bit-based implementation is possible.
- Hardware support for the encoded operations would also speed up our encoded programs.

For improving the error detection capabilities the following can be done:

- A thorough error injection into the encoded operations and the encoded replacement operations will help to identify weaknesses. Further debugging should remove these weaknesses.

- Further pursuing the research presented in Chapter 5 will enable us to choose better encoding parameters, for example, an A that maximizes the probability of detecting errors.

Porting to embedded systems

For transferring CEP into the industrial practice, it will be required to support other backends than desktop computers. Therefore, embedded systems such as Infineon's TriCore should be supported.

Formal validation

Currently, encoding is just claimed and believed to be correct due to its simple arithmetic representation. However, the implementation is complicated in many places. Our implementation is tested by an extensive set of unit and integration tests and its error detection capabilities are evaluated using error injection. However, these measures cannot guarantee that the safe encoded software produces the same results as the unsafe original software in an error-free execution and that no implementation error prevents error detection. Applying formal verification techniques can provide guarantees with respect to the correctness of the transformation and the error detection capabilities achieved.

Debugging of hardware errors

The statically assigned signatures can also be used to localize permanent hardware errors and avoid them by reconfiguration. Therefore, the encoding of all results computed have to be checked immediately. If a check fails, the part of the hardware implementing the previously executed operation is erroneous. Of course, this expensive version that checks the code of all computations immediately can only be used for diagnosis. During normal operation of a system code checks have to be done regularly, but less often. Only if an error is detected such a system should switch into the expensive diagnosis mode.

Adaptive encoding

Some parts of a program are more critical than others [PGZ08]. CEP can be extended to support different encodings within one program. This would allow to adapt the encoding to the needs for safety even more.

Encoding for security

Encoding can also be used to detect security attacks. For example, if an attacker wants to insert new code using a buffer overflow attack, he has to ensure that his code conserves the encoding of output values produced and sends the correct check values to the watchdog. If he does not know the encoding parameters, this is difficult. However, further research is needed to determine the conditions under which attacks can be prevented.

11.3. Publications, Proposals, and Filed Patents

We present our encoding technologies in the following publications:

- *"Hardware Failure Virtualization Via Software Encoded Processing"* presented at INDIN 2007 [WF07a]
- *"Software Encoded Processing: Building Dependable Systems with Commodity Hardware"* presented at SafeComp 2007 [WF07b]
- *"Software Protection Mechanisms for Dependable Systems"* presented at DATE 2008 [WM08a]
- *"AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware"* presented at SafeComp 2009 [SSF09]

- *"Software-Implemented Hardware Error Detection: Costs and Gains"* presented at DEPEND 2010 [SSSF10c]
- *"ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software"* presented at SafeComp 2010 [SSSF10a]

We present our error injection tools in the following publications:

- *"Hardware Fault Injection Using Dynamic Binary Instrumentation: FIT-grind"* (fast abstract) presented at EDCC 2006 [WF06]
- *"Slice Your Bug: Debugging Error Detection Mechanisms using Error Injection Slicing"* presented at EDCC 2010 [SSSF10b]

Two patents are filed to protect the intellectual property created in this thesis:

- *"Data processing arrangement comprises coding device, which is adjusted to assign codeword to data item to be stored in memory element based on signal information"* [Wap07]
- *"Verfahren zur Datenverarbeitung zum Bereitstellen eines Wertes zum Ermitteln, ob bei einer Ausführung eines Programms ein Fehler aufgetreten ist"* [Sch10]

Furthermore, several successful proposals and one industry cooperation resulted from the research presented in this thesis:

- Siemens for funded the one-year project *Optimization techniques and safety assessment of the coded monoprocessor.*
- The Deutsche Forschungsgemeinschaft (German Research Foundation) funded the two-year project *Software implemented, hardware independent detection of transient and permanent execution errors with adaptable safety.*
- EXIST by the German Bundesministerium für Wirtschaft und Technologie will fund the salaries of three researchers and one business manager for 1.5 years with the objective to transfer the findings of this thesis into a startup.
- The Reykjavík University will fund a postdoc position for one year with the objective to formally verify CEP.

A. Detection of Over- and Underflows by Choice of A

Several encoding approaches (for example, VCP [For89] and TRUMP [CRA06]) claim that they are able to detect overflows because these destroy the encoding. However, the authors of these papers do not demonstrate that this is indeed possible. Our findings in Section 4.2.1 show that overflows can remain undetected. Nevertheless, using encoding to detect over- and underflows would be useful for encoding applications in which over- and underflows should never happen.

In the following, we analyze how the A for an ANB-code has to be chosen for facilitating the detection of over- and underflows in computer-implemented arithmetic operations. Therefore, we try to choose A in such a way that an over- or underflow of the functional value always produces an invalid code word instead of a valid one. The considerations presented consider only unsigned encoded numbers.

A.1. Detecting Overflows

In Chapter 4 we defined that

- the functional values range from 0 to $2^n - 1$ and
- the encoded values from 1 to $2^m - 1$. The smallest encoded value is 1 because the signature of a code word has to be at least 1. Thus, the smallest functional value, 0, can be encoded to $A * 0 + 1 = 1$.

All computations are either executed $\bmod 2^n$ for functional values or $\bmod 2^m$ for encoded values. Thus, the results never become larger or equal to 2^n and 2^m respectively. However, in our formulas we will explicitly state this modulo operation to denote the difference to the arithmetic operations that do not overflow.

If we want to ensure that overflows in the functional values for an operation op destroy the encoding, we have to ensure that for its encoded operation op_c, the following equations hold:

$$(\mathrm{op}_c(x_c, y_c) \mod 2^m) \mod A = \mathrm{op}_{sig}(B_x, B_y) \quad \text{if } op(x, y) \text{ does not overflow}$$
$$(\mathrm{op}_c(x_c, y_c) \mod 2^m) \mod A \neq \mathrm{op}_{sig}(B_x, B_y) \quad \text{if } op(x, y) \text{ does overflow}$$

Note that the function $\mathrm{op}_{sig}(B_x, B_y)$ determines the signature expected for the operation op_c. The value returned by $\mathrm{op}_{sig}(B_x, B_y)$ is always larger than zero and smaller than A.

As long as the encoded operations themselves do not overflow, the encoding of their results is always valid – even if the functional values overflow. See Section 4.2.1 for the details. Thus, we will analyze if it is possible to choose an A that ensures that

- the encoded values overflow whenever the functional values do and
- do not overflow whenever the functional values do not overflow.

Furthermore, we have to show that for this A

- if the encoded values overflow the signature is destroyed, that is, becomes unequal to the expected signature.

In the following, we determine lower and upper bounds for A that ensure that overflows in the encoded values happen if and only if overflows in the functional values happen. Furthermore, we determine size restrictions for the functional and encoded values that ensure that at least one A exists fulfilling the size restrictions. Last, we show under which conditions it is guaranteed that an overflow in the encoded values results in an invalid code word.

A.1.1. Lower Bound for A

To ensure that functional and encoded values simultaneously overflow, we have to choose A in such a way that for the first functional value that overflows, i. e., that is not representable with n bits, its encoded version overflows, too. This is the case when

$$\text{if } z \geq 2^n \quad \text{then } z_c = A * z + B_z \geq 2^m$$

holds. The smallest possible signature B_z is one. And the smallest functional result of an operation that is overflown is 2^n. Thus, A's lower bound is determined by:

$$\begin{aligned} A * z + B_z &\geq 2^m \\ A * 2^n + 1 &\geq 2^m \\ A &\geq \frac{2^m - 1}{2^n} \end{aligned}$$

However, A must be an integer. Furthermore, as we explained in Section 5.1.1 A must not be a power of two. Thus, A's lower bound is:

$$A > \frac{2^m}{2^n}$$

A.1.2. Upper Bound for A

If A is chosen too large, we cannot guarantee that for every functional value a valid code word exists. Thus, we have to determine an upper bound for A.

The greatest functional value that must be encodable is $2^n - 1$. The largest possible signature B_{max} is a value smaller than A and larger than zero. We use

b to denote how much smaller than A B_{max} is. Thus, $B_{max} := A - b$ with $b \in \mathbb{N}$ and $0 < b < A$. The resulting largest code word $A * (2^n - 1) + B_{max}$ has to be smaller than 2^m because otherwise $2^n - 1$ would not be representable. Thus, A's upper bound is determined by:

$$\begin{aligned} A * (2^n - 1) + B_{max} &< 2^m \\ A * (2^n - 1) + A - b &< 2^m \\ A * 2^n - b &< 2^m \\ A &< \frac{2^m + b}{2^n} \end{aligned}$$

A.1.3. Interval of Available As

As we have shown in the previous sections, for ensuring that the functional and the encoded values overflow at the same time, A has to fulfill the following condition:

$$\frac{2^m}{2^n} < A < \frac{2^m + b}{2^n} \tag{A.1}$$

The larger b is, that is, the more we restrict the number of available signatures by reducing B_{max} the more possible As exist that fulfill this inequality.

The interval within which A must lie has the size

$$\begin{aligned} noOfPossibleAs &= \frac{2^m + b}{2^n} - \frac{2^m}{2^n} - 1 \\ noOfPossibleAs &= \frac{b}{2^n} - 1 \end{aligned}$$

Thus, b has to be larger than 2^n. Otherwise, $noOfPossibleAs$ will be smaller than or equal to zero, meaning that no A exists fulfilling the conditions.

Both, b and B_{max} depend on A: $A = B_{max} + b$. Thus, even if we choose $B_{max} = 0$, A has to be larger than 2^n. Such an A will at least add $n + 1$ bits of redundancy to the n-bit-sized functional values. This results in the condition $m > 2 * n$ for the sizes of the functional and the encoded values. The larger the difference between m and $2 * n$ is

- the larger can be b and, thus, the more possible As exist, or
- the larger can be B_{max} and, thus, the more possible different signatures exist.

Assume, for example, $n = 32$ and $m = 70$. Thus, A can require 38 bits. If we constrain the amount of available signatures to $B_{max} = A - 2^{36}$ by setting $b = 2^{36}$, according to Equation A.1 only $\frac{2^{70}+2^{36}}{2^{32}} - \frac{2^{70}}{2^{32}} = 16$ values fulfill the conditions and are possible values for A. If we constrain the amount of signatures more by setting $b = 2^{37}$, $\frac{2^{70}+2^{37}}{2^{32}} - \frac{2^{70}}{2^{32}} = 32$ values fulfill the conditions and are possible values for A.

A.1.4. Condition for Code Invalidation

Furthermore, for guaranteeing the detection of an overflow, we have to ensure that the overflow in the domain of the code words results in an invalid code word, that is, the result is not a multiple of A with the expected signature B. To guarantee that an overflow destroys the signature of the resulting code word, requires further restrictions on A.

Assume that we have a functional value a that is

- larger than $2^n - 1$, and
- is encoded with A and the signature B_a.

This value overflows during its compution. If we ensure that functional and encoded values overflow simultaneously, $\bmod 2^m$ is applied to the encoded version $a_c = A * a - B_a$ of a. Furthermore, for code checking, the signature contained in a_c is computed by applying $\bmod A$ to the encoded value. This computed signature is compared to the expected signature B_a. For detecting an overflow, the computed signature and the expected signature have to be unequal:

$$((A * a + B_a) \mod 2^m) \mod A \neq B_a$$

Transforming all modulo operations according to the definition of modulo results for some $y, z \in \mathbb{N}$ in:

$$\begin{aligned} ((A * a + B_a) \mod 2^m) \mod A &\neq B_a \\ ((A * a + B_a) \mod 2^m) - A * y &\neq B_a \\ A * a + B_a - 2^m * z - A * y &\neq B_a \\ A(a - y) &\neq 2^m * z \end{aligned}$$

This inequality can be guaranteed to remain an inequality if A's factorization contains at least one factor larger than z and larger than 2. Under this condition, we cannot find an assignment for a, y, and z so that $A(a-y) = 2^m * z$ is fulfilled.

Note that z by its definition counts how often the encoded value $A * a + B_a$ is overflown, that is, how often 2^m fits into $A * a + B_a$. Thus, $z \geq 1$ holds because at least the one overflow that we want to detect happens.

To fulfill $A(a - y) = 2^m * z$ the following equations have to be fulfilled:

$$\begin{aligned} A &= 2^{m-k} * z_1 \\ (a - y) &= 2^k * z_2 \end{aligned}$$

with $z \geq 1$ and $z = z_1 * z_2$ and, thus, $z1 \leq z$ and $z2 \leq z$.

The prime factorization of a number is unique. If A's prime factorization contains at least one factor larger than z and 2, this factor cannot be a factor of $2^{m-k} * z_1$. Thus, the equation $A = 2^{m-k} * z_1$ cannot be fulfilled.

Thus, under the condition that A's prime factorization contains at least one factor larger than z and 2, $A(a-y) \neq 2^m * z$ holds and the detection of overflows is guaranteed.

A.2. Detecting Underflows

Underflows only occur for subtractions. For unsigned encoded values, the underflows of the functional and the encoded values already happen simultaneously (see implementation of the encoded subtraction in Section 4.2.1). Thus, we only have to show that such an underflow results in an unexpected signature.

Assume that we are subtracting two encoded values $x_c = A * x + B_x$ and $y_c = A * y + B_y$ with $y > x$. As we show in Section 4.2.1 the result of this underflowing subtraction is

$$x_c - y_c = 2^m - (A * y + B_y - (A * x + B_x)) = 2^m - A(y - x) + B_x - B_y.$$

The signature expected for $x_c - y_c$ is $B_x - B_y$. However, under the assumption that B_x and B_y were chosen so that $B_x - B_y < A$, the signature of the underflown result of the subtraction of x_c and y_c is

$$(2^m - A(y - x) + B_x - B_y) \bmod A = 2^m \bmod A + B_x - B_y.$$

Since A must not be a power of two,

$$2^m \bmod A + B_x - B_y \neq B_x - B_y$$

holds. Thus, underflows are detectable for unsigned encoded values.

A.3. Practical Aspects

Detecting over- and underflows using arithmetic encoding seems to be useful. However, at least for our AN-based encoding approaches, it has severe disadvantages.

For example, consider our implementation of encoded shift operations that we describe in Section 4.2.2. Therefore, shifts to the left are mapped to multiplications. If a shift to the left shifts bits out of the functional value, this is nothing else than an overflow in the multiplication that we use to realize the encoded version of this shift operation. Thus, with the overflow detection proposed above, valid operations, for example, our encoded shift to the left, would destroy the encoding.

The operations affected in this way by the proposed overflow detection are:

- shifts to the left,
- bitwise logical operations, and
- unaligned loads and stores.

Thus, code-based detection of overflows, makes encoding these operations more difficult.

Index

Bibliography

[AB09] Diogo José Costa Alves and Edna Barros. A logic built-in self-test architecture that reuses manufacturing compressed scan test patterns. In *SBCCI '09: Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design*, pages 1–6, New York, NY, USA, 2009. ACM.

[ACK+03] Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs, and Gunther H. Leber. Comparison of Physical and Software-Implemented Fault Injection Techniques. *IEEE Transactions on Computers*, 52(9):1115–1133, 2003.

[AGM+71] A. Avizienis, G.C. Gilley, F.P. Mathur, D.A. Rennels, J.A. Rohr, and D.K. Rubin. The STAR (Self-Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design. *Transactions on Computers*, Volume C-20:1312– 1321, 1971.

[AH99] Algirdas Avizienis and Yutao He. Microprocessor Entomology: A Taxonomy of Design Faults in COTS Microprocessors. In *DCCA '99: Proceedings of the conference on Dependable Computing for Critical Applications*, page 3, Washington, DC, USA, 1999. IEEE Computer Society.

[All70] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[ANS+04] W. Abdelmoez, D. M. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H. H. Ammar, Bo Yu, and A. Mili. Error Propagation In Software Architectures. In *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium*, pages 384–393, Washington, DC, USA, 2004. IEEE Computer Society.

[ASS09] Naga Durga Prasad Avirneni, Viswanathan Subramanian, and Arun K. Somani. Low overhead Soft Error Mitigation techniques for high-performance and aggressive systems. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009*, pages 185–194. IEEE Computer Society, 2009.

[Aus99] Todd M. Austin. DIVA: a Reliable Substrate for Deep Submicron Microarchitecture Design. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 196–207, Washington, DC, USA, 1999. IEEE Computer Society.

[Avi64] Algirdas Avizienis. A set of algorithms for a diagnosable arithmetic unit. Technical report, Jet Propulsion Laboratory, 1964.

[Avi65] Algirdas Avizienis. A Study of the Effectiveness of Fault-Detecting Codes for Binary Arithmetic. Technical report, Jet Propulsion Laboratory, 1965.

[Avi71] A. Avizienis. Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design. In *IEEE Transactions on Computers*, pages 1322 – 1331, 1971.

[Bau02] Robert Baumann. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction. In *International Electron Devices Meeting*, 2002.

[Bau05] Robert C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5:305– 316, 2005.

[BBV+05] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. NonStopAdvanced Architecture. *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 00:12–21, 2005.

[BCN+01] Alfredo Benso, Stefano Di Carlo, Giorgio Di Natale, Paolo Prinetto, and Luca Tagliaferri. Control-Flow Checking via Regular Expressions. In *Proceedings of the 10th Asian Test Symposium (ATS '01)*, page 299, Washington, DC, USA, 2001. IEEE Computer Society.

[BCPT00] Alfredo Benso, Silvia Chiusano, Paolo Prinetto, and L. Tagliaferri. A C/C++ Source-to-Source Compiler for Dependable Applications. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, page 71, Washington, DC, USA, 2000. IEEE Computer Society.

[BDH+98] Feng Bao, Robert H. Deng, Yongfei Han, Albert B. Jeng, A. Desai Narasimhalu, and Teow-Hin Ngair. Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 115–124, London, UK, 1998. Springer-Verlag.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology EUROCRYPT 97*, 1997.

[BDL01] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Eliminating Errors in Cryptographic Computations. *J. Cryptology*, 14(2):101–119, 2001.

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference, FREENIX Track*, pages 41– 46. USENIX, April 2005.

[BHP+71] W. G. Bouricius, E. P. Hsieh, G. R. Putzolu, J. P. Roth, P. R. Schneider, and C. Tan. Algorithms for Detection of Faults in Logic Circuits. *IEEE Trans. Comput.*, 20(11):1258–1264, 1971.

[BI86] M A Breuer and A A Ismaeel. Roving Emulation as a Fault Detection Mechanism. *IEEE Trans. Comput.*, 35(11):933–939, 1986.

[BKIL03] S. Bagchi, Z. Kalbarczyk, R. Iyer, and Y. Levendel. Design and Evaluation of Preemptive Control Signature(PECOS) Checking. IEEE Transactions on Computers, 2003.

[BL85] Bella Bose and Der J. Lin. Systematic Unidirectional Error-Detecting Codes. *IEEE Trans. Comput.*, 34(11):1026–1032, 1985.

[BLR90] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 73–83, New York, NY, USA, 1990. ACM Press.

[BMBF05] Jason Blome, Scott Mahlke, Daryl Bradley, and Krisztián Flautner. A microarchitectural analysis of soft error propagation in a production-level embedded microprocessor. In *In Proceedings of the First Workshop on Architecture Reliability*, 2005.

[BMR+08] C. Bolchini, A. Miele, M. Rebaudengo, F. Salice, D. Sciuto, L. Sterpone, and M. Violante. Software and Hardware Techniques for SEU Detection in IP Processors. *J. Electron. Test.*, 24(1-3):35–44, 2008.

[Bog10] Walt Bogdanich. Radiation Offers New Cures, and Ways to Do Harm. The New York Times: http://www.nytimes.com/2010/01/24/health/24radiation.html?pagewanted=all, January 2010.

[Bor05] Shekhar Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, 2005.

[Bor06] Shekhar Borkar. Electronics beyond nano-scale CMOS. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 807–808, New York, NY, USA, 2006. ACM Press.

[BSO05] Fred A. Bower, Daniel J. Sorin, and Sule Ozev. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International*

Symposium on Microarchitecture, pages 197–208, Washington, DC, USA, 2005. IEEE Computer Society.

[BSW+00] Saurabh Bagchi, Balaji Srinivasan, Keith Whisnant, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Hierarchical Error Detection in a Software Implemented Fault Tolerance (SIFT) Environment. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):203–224, 2000.

[BUEA06] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. Architectural support for software-based protection. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 42–51, New York, NY, USA, 2006. ACM Press.

[BWWA06] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-Based Transparent and Comprehensive Control-Flow Error Detection. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 333–345, Washington, DC, USA, 2006. IEEE Computer Society.

[CCFC04] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[CCL+08] B.H. Calhoun, Yu Cao, Xin Li, Ken Mai, L.T. Pileggi, R.A. Rutenbar, and K.L. Shepard. Digital Circuit Design Challenges and Opportunities in the Era of Nanoscale CMOS. *Proceedings of the IEEE*, 96(2):343–365, Feb. 2008.

[Chi64] R. Chien. On linear residue codes for burst-error correction. In *IEEE Transactions on Information Theory*, 1964.

[CMS98] João Carreira, Henrique Madeira, and João Gabriel Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Trans. Softw. Eng.*, 24(2):125–136, 1998.

[Con03] Cristian Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4):14–19, 2003.

[CRA06] Jonathan Chang, George A. Reis, and David I. August. Automatic Instruction-Level Software-Only Recovery. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 83–92, Washington, DC, USA, 2006. IEEE Computer Society.

[CXIW02] Shuo Chen, Jun Xu, R.K. Iyer, and K. Whisnant. Evaluating the security threat of firewall data corruption caused by instruction transient errors. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 495–504, 2002.

[CXK+04] Shuo Chen, Jun Xu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Keith Whisnant. Modeling and evaluating the security threats of transient errors in firewall software. *Perform. Eval.*, 56(1-4):53–72, 2004.

[Dec05] G. Declerck. A look into the future of nanoelectronics. In *VLSI Technology, 2005. Digest of Technical Papers. 2005 Symposium on*, pages 6–10, June 2005.

[DHW09] Anand Dixit, Raymond Heald, and Alan Wood. Trends from Ten Years of Soft Error Experimentation. In *System Effects of Logic Soft Errors (SELSE)*, 2009.

[dKS09] Marc de Kruiff and Karu Sankaralingam. Exploring the Synergy of Emerging Workloads and Silicon Reliability Trends. In *IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2009.

[DMZ09] Martin Dimitrov, Mike Mantor, and Huiyang Zhou. Understanding software approaches for GPGPU reliability. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 94–104, New York, NY, USA, 2009. ACM.

[Dol06] Daniel Dollé. Vital software: Formal method and coded processor. In *Proceedings of Embedded Real Time Software (ERTS) 2006*, Toulouse, France, 2006.

[DT99] Debaleena Das and Nur A. Touba. Synthesis of Circuits with Low-Cost Concurrent Error Detection Based on Bose-Lin Codes. *J. Electron. Test.*, 15(1-2):145–155, 1999.

[DZ07] Martin Dimitrov and Huiyang Zhou. Unified Architectural Support for Soft-Error Protection or Software Bug Detection. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 73–82, Washington, DC, USA, 2007. IEEE Computer Society.

[DZ09] Martin Dimitrov and Huiyang Zhou. Anomaly-based bug prediction, isolation, and validation: an automated approach for software debugging. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 61–72, New York, NY, USA, 2009. ACM.

[FGAM10] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 385–396, New York, NY, USA, 2010. ACM.

[For89] P. Forin. Vital Coded Microprocessor Principles and Application for Various Transit Systems. In *IFA-GCCT*, pages 79–84, Toulouse, France, Sept 1989.

[GA03] Sudhakar Govindavajhala and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 154, Washington, DC, USA, 2003. IEEE Computer Society.

[GEJL10] Nishant J. George, Carl R. Elks, Barry W. Johnson, and John Lach. Bit-slice logic interleaving for spatial multi-bit soft-error tolerance. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010*, pages 141 – 150, Chicago, USA, 2010. IEEE Computer Society.

[GRSRV03] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Soft-error detection using control flow assertions. In *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 581–588, Nov. 2003.

[GSS06] T. S. Ganesh, Viswanathan Subramanian, and Arun Somani. SEU Mitigation Techniques for Microprocessor Control Logic. *European Dependable Computing Conference*, 0:77–86, 2006.

[GSVP03] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. *International Symposium on Computer Architecture*, 0:98, 2003.

[HA84] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Trans. Computers*, 33(6):518–528, 1984.

[HJS01] Martin Hiller, Arshad Jhumka, and Neeraj Suri. An Approach for Analysing the Propagation of Data Errors in Software. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 161–172, Washington, DC, USA, 2001. IEEE Computer Society.

[HLD+05] Jie S. Hu, Feihui Li, Vijay Degalahal, Mahmut Kandemir, N. Vijaykrishnan, and Mary J. Irwin. Compiler-Directed Instruction Duplication for Soft Error Detection. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1056–1057, Washington, DC, USA, 2005. IEEE Computer Society.

[IBM99] IBM. IBM Chipkill Memory. *Whitepaper*, 1999.

[IKP+07] Ravishankar K. Iyer, Zbigniew Kalbarczyk, Karthik Pattabiraman, William Healey, Wen-Mei W. Hwu, Peter Klemperer, and Reza Farivar. Toward Application-Aware Security and Reliability. *IEEE Security and Privacy*, 5(1):57–62, 2007.

[INKM05] Ravishankar K. Iyer, Nithin M. Nakka, Zbigniew T. Kalbarczyk, and Subhasish Mitra. Recent Advances and New Avenues in Hardware-Level Reliability Support. *IEEE Micro*, 25(6):18–29, 2005.

[ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[Joc02] Markus Jochim. Detecting Processor Hardware Faults by Means of Automatically Generated Virtual Duplex Systems. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 399–408, Washington, DC, USA, 2002. IEEE Computer Society.

[JRBS06] Vivek Joshi, Rajeev R. Rao, David Blaauw, and Dennis Sylvester. Logic SER Reduction through Flipflop Redesign. In *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 611–616, Washington, DC, USA, 2006. IEEE Computer Society.

[JRSMF98] Jr. J. R. Samson, W. Moreno, and F. Falquez. A Technique for Automated Validation of Fault Tolerant Designs Using Laser Fault Injection (LFI). In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, page 162, Washington, DC, USA, 1998. IEEE Computer Society.

[KHP04] Tanay Karnik, Peter Hazucha, and Jagdish Patel. Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. *IEEE Trans. Dependable Secur. Comput.*, 1(2):128–143, 2004. Senior Member-Tanay Karnik and Member-Peter Hazucha.

[KIBW99] Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, Saurabh Bagchi, and Keith Whisnant. Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):560–579, 1999.

[KIT93] Wei-lun Kao, Ravishankar K. Iyer, and Dong Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults. *IEEE Trans. Softw. Eng.*, 19(11):1105–1118, 1993.

[KKA95] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Trans. Comput.*, 44(2):248–260, 1995.

[Kna06] Thomas Knauth. Performance Improvements of the Vital Encoded Interpreter. Großer Beleg, Technische Universität Dresden, 2006.

[KS06] Thomas Kottke and Andreas Steininger. A Reconfigurable Generic Dual-Core Architecture. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 45–54, Washington, DC, USA, 2006. IEEE Computer Society.

[KSS02] Richard Krüger, Andreas Schenk, and Frank Schiller. Patent DE 102 19 501 B4: System und Verfahren zur Verbesserung von Fehlerbeherrschungsmassnahmen, insbesondere in Automatisierungssystemen, April 2002.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and*

optimization (CGO), page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[Lau06] Olivier Lauzeral. WNR Helps Solve New Reliability Challenges on Electronic Devices. LANSCE Los Alamos Neutron Science Center Activity Report 2006, 2006.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[LCP+09] Galen Lyle, Shelley Cheny, Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar Iyer. An end-to-end approach for the automatic derivation of application-aware error detectors. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009*, pages 584–589. IEEE Computer Society, 2009.

[LG04] Xiaobin Li and Jean-Luc Gaudiot. A Compiler-Assisted On-Chip Assigned-Signature Control Flow Checking. In Yew and Xue [YX04], pages 554–567.

[LH07] Daniel Larsson and Reiner Hähnle. Symbolic Fault Injection. In *4th International Verification Workshop in connection with CADE-21*, pages 85–103, Bremen, Germany, 2007.

[LLV] The LLVM Compiler Infrastructure. http://llvm.org/.

[LMM08] Yanjing Li, Samy Makar, and Subhasish Mitra. CASP: concurrent autonomous chip self-test using stored test patterns. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 885–890, New York, NY, USA, 2008. ACM.

[LOBR09] Daniel Limbrick, Edward Ossi, Bharat Bhuva, and William Robinson. Soft-error Mitigation at the Architecture-Level Using Berger Codes and Instruction Repetition. In *IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE 5)*, 2009.

[LRS+08] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. *SIGOPS Oper. Syst. Rev.*, 42(2):265–276, 2008.

[LSHC07] Xin Li, Kai Shen, Michael C. Huang, and Lingkun Chu. A memory soft error measurement on production systems. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[LTRN92] Jien-Chung Lo, Suchai Thanawastien, T. R. N. Rao, and Michael Nicolaidis. An SFS Berger check prediction ALU and its application to self-checking processor designs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11(4):525–540, 1992.

[Man67] David Mandelbaum. Arithmetic codes with large distance. In *IEEE Transactions on Information Theory*, 1967.

[Mas64] J. L. Massey. Survey of residue coding for arithmetic errors. *ICC Bulletin*, Volume 3:195–209, 1964.

[MBS07] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 210–222, Washington, DC, USA, 2007. IEEE Computer Society.

[MHH+05] S.E. Michalak, K.W. Harris, N.W. Hengartner, B.E. Takala, and S.A. Wender. Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer. In *IEEE Transactions on Device and Materials Reliability*, pages 329 – 335, 2005.

[Mil] Ethan L. Miller. DLX-Compiler: http://www2.ucsc.edu/courses/cmps111-elm/dlx/install.shtml. UC Santa Cruz, School of Engineering.

[MR98] A. Maamar and G. Russell. A 32-Bit Risc Processor with Concurrent Error Detection. In *Proceedings of the 24th Conference on EUROMICRO (EUROMICRO '98)*, pages 461 – 467, Washington, DC, USA, 1998. IEEE Computer Society.

[MS07] Albert Meixner and Daniel J. Sorin. Error Detection Using Dynamic Dataflow Verification. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 104–118, Washington, DC, USA, 2007. IEEE Computer Society.

[MS08] Albert Meixner and Daniel J. Sorin. Detouring: Translating software to circumvent hard faults in simple cores. In *DSN*, pages 80–89. IEEE Computer Society, 2008.

[MSVZ07] Jürgen Mottok, Frank Schiller, Thomas Völkl, and Thomas Zeitler. A Concept for a Safe Realization of a State Machine in Embedded Automotive Applications. In Francesca Saglietti and Norbert Oster, editors, *The 26th International Conference on Computer Safety, Reliability and Security (SafeComp 2007)*, volume 4680 of *Lecture Notes in Computer Science*, pages 283–288. Springer, 2007.

[MSZ+05] Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust System Design with Built-In Soft-Error Resilience. *Computer*, 38(2):43–52, 2005.

[NDMF97] Michael Nicolaidis, Ricardo O. Duarte, Salvador Manich, and Joan Figueras. Fault-Secure Parity Prediction Arithmetic Operators. *IEEE Des. Test*, 14(2):60–71, 1997.

[Net04] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation.* PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, November 2004.

[NGY+05] Bogdan Nicolescu, Nicolas Gorse, Savaria Yvon, El Mostapha Aboulhamid, and Raoul Velazco. On the use of model checking for the verification of a dynamic signature monitoring approach. *IEEE transactions on nuclear science*, 2005.

[NPI07] Nithin Nakka, Karthik Pattabiraman, and Ravishankar Iyer. Processor-Level Selective Replication. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 544–553, Washington, DC, USA, 2007. IEEE Computer Society.

[NS07] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74, New York, NY, USA, 2007. ACM.

[NV03] B. Nicolescu and Raoul Velazco. Detecting Soft Errors by a Purely Software Approach: Method, Tools and Experimental Results. In *Design, Automation and Test in Europe (DATE '03)*, pages 20057–20063. IEEE Computer Society, 2003.

[OMM02] Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. ED4I: Error Detection by Diverse Data and Duplicated Instructions. *IEEE Trans. Comput.*, 51(2):180–199, 2002.

[OÖ89] James L. Olivier and Füsun Özgüner. Design of concurrent error-detecting systolic arrays using $|g3N|_M$ codes. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 8(10):1089–1099, 1989.

[OSM02] N. Oh, P.P. Shirvani, and E.J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51:111–122, 2002.

[Oze92] Patrick Ozello. The coded microprocessor certification. In *Proceedings of SafeComp*, pages 185–190, 1992.

[Pap09] The Paparazzi Project. http://paparazzi.enac.fr/wiki/ Main_Page, 2009.

[PF82] J. H. Patel and L. Y. Fung. Concurrent Error Detection in ALU's by Recomputing with Shifted Operands. *IEEE Trans. Comput.*, 31(7):589–595, 1982.

[PGZ06] Karthik Pattabiraman, Vinod Grover, and Benjamin G. Zorn. Samurai - Protecting Critical Heap Data in Unsafe Languages. Technical report, Microsoft, 2006.

[PGZ08] Karthik Pattabiraman, Vinod Grover, and Benjamin G. Zorn. Samurai: protecting critical data in unsafe languages. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 219–232, New York, NY, USA, 2008. ACM.

[PH90] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[PKI05] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Application-Based Metrics for Strategic Placement of Detectors. In *PRDC '05: Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing*, pages 75–82, Washington, DC, USA, 2005. IEEE Computer Society.

[PKI07] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Automated Derivation of Application-aware Error Detectors using Static Analysis. In *IOLTS '07: Proceedings of the 13th IEEE International On-Line Testing Symposium*, pages 211–216, Washington, DC, USA, 2007. IEEE Computer Society.

[PNKI08] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer. SymPLFIED: Symbolic program-level fault injection and error detection framework. *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 472–481, June 2008.

[Pow95] David Powell. Failure Mode Assumptions and Assumption Coverage. Technical report, 1995.

[PSC+06] Karthik Pattabiraman, Giacinto Paulo Saggese, Daniel Chen, Zbigniew Kalbarczyk, and Ravishankar Iyer. Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware. In *Proceedings of the Sixth European Dependable Computing Conference (EDCC 2006)*, pages 97 – 108, Coimbra, Portugal, May 2006.

[PSR00] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. A study of slipstream processors. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 269–280, New York, NY, USA, 2000. ACM.

[Rao70] T.R.N. Rao. Biresidue Error-Correcting Codes for Computer Arithmetic. In *Transactions on Computers*, 1970.

[Rao74] Thammavarapu R. N. Rao. *Error Coding for Arithmetic Processors*. Academic Press, Inc., Orlando, FL, USA, 1974.

[RAZR06] Vimal K. Reddy, Ahmed S. Al-Zawawi, and Eric Rotenberg. Assertion-Based Microarchitecture Design for Improved Fault Tolerance. In *Proceedings of the International Conference on Computer Design (ICCD 2006)*, pages 362 – 369, San Jose, USA, 2006.

[RBS06] Rajeev R. Rao, David Blaauw, and Dennis Sylvester. Soft error reduction in combinational logic using gate resizing and flipflop selection. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 502–509, New York, NY, USA, 2006. ACM.

[RCA+06] George A. Reis, Jonathan Chang, David I. August, Robert Cohn, and Shubhendu S. Mukherjee. Configurable Transient Fault Detection via Dynamic Binary Translation. In *Proceedings of the 2nd Workshop on Architectural Reliability (WAR)*, 2006.

[RCMM07] Paul Racunas, Kypros Constantinides, Srilatha Manne, and Shubhendu S. Mukherjee. Perturbation-based Fault Screening. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 169–180, Washington, DC, USA, 2007. IEEE Computer Society.

[RCV+05a] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the international symposium on Code generation and optimization (CGO)*, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[RCV+05b] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Design and Evaluation of Hybrid Fault-Detection Systems. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 148–159, Washington, DC, USA, 2005. IEEE Computer Society.

[RG71] T. Rao and O. Garcia. Cyclic and multiresidue codes for arithmetic operations. In *IEEE Transactions on Information Theory*, 1971.

[RLC+08] Eduardo Luis Rhod, Carlos Arthur Lisbôa, Luigi Carro, Matteo Sonza Reorda, and Massimo Violante. Hardware and Software Transparency in the Protection of Programs Against SEUs and SETs. *J. Electron. Test.*, 24(1-3):45–56, 2008.

[RM00] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 25–36, New York, NY, USA, 2000. ACM Press.

[Rot99] Eric Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 84, Washington, DC, USA, 1999. IEEE Computer Society.

[RR07] Vimal Reddy and Eric Rotenberg. Inherent Time Redundancy (ITR): Using Program Repetition for Low-Overhead Fault Tolerance. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP*

International Conference on Dependable Systems and Networks, pages 307–316, Washington, DC, USA, 2007. IEEE Computer Society.

[RR08] Vimal K. Reddy and Eric Rotenberg. Coverage of a Microarchitecture-level Fault Check Regimen in a Superscalar Processor. In *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–10, 2008.

[RRTV99] Maurizio Rebaudengo, Matteo Sonza Reorda, Marco Torchiano, and Massimo Violante. Soft-Error Detection through Software Fault-Tolerance Techniques. In *DFT '99: Proceedings of the 14th International Symposium on Defect and Fault-Tolerance in VLSI Systems*, pages 210–218, Washington, DC, USA, 1999. IEEE Computer Society.

[RRVT01] Maurizio Rebaudengo, Matteo Sonza Reorda, Massimo Violante, and Marco Torchiano. A Source-to-Source Compiler for Generating Dependable Software. *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, 00:0035, 2001.

[SA03] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 2–12, London, UK, 2003. Springer-Verlag.

[SAC+99] T.J. Slegel, R.M. Averill, M.A. Check, B.C. Giamei, B.W. Krumm, W.H. Krygowski, C.A.and Li, J.S. Liptay, J.D. MacDougall, T.J. McPherson, J.A. Navarro, E.M. Schwarz, K. Shum, and C.F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19:12 – 23, 1999.

[Sch09] André Schmitt. Development of an ANBD-Checker for ParExC. diploma thesis, Technische Universität Dresden, 2009.

[Sch10] Verfahren zur Datenverarbeitung zum Bereitstellen eines Wertes zum Ermitteln, ob bei einer Ausführung eines Programms ein Fehler aufgetreten ist. Patent pending: DE 10 2010 037 457.1 applied for by Technische Universität Dresden, 2010. Inventors: Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer.

[SCK04] Daniel P. Siewiorek, Ram Chillarege, and Zbigniew T. Kalbarczyk. Reflections on Industry Trends and Experimental Research in Dependability. *IEEE Trans. Dependable Secur. Comput.*, 1(2):109–127, 2004.

[Ser08] Amazon Web Services. Amazon S3 Availability Event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html, 2008.

[SG98] L. Spainhower and T. A. Gregg. G4: A Fault-Tolerant CMOS Mainframe. In *FTCS '98: Proceedings of the The Twenty-Eighth*

Annual International Symposium on Fault-Tolerant Computing, page 432, Washington, DC, USA, 1998. IEEE Computer Society.

[SG99] L. Spainhower and T. A. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research*, 43, 1999.

[SJGF09] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46, New York, NY, USA, 2009. ACM.

[SK99] Dennis Sylvester and Kurt Keutzer. Rethinking Deep-Submicron Circuit Design. *Computer*, 32(11):25–33, 1999.

[SKK+02] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 389–398, Washington, DC, USA, 2002. IEEE Computer Society.

[SKKR10] Joseph Sloan, David Kesler, Rakesh Kumar, and Ali Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010*. IEEE Computer Society, 2010.

[SLR+08] S.K. Sahoo, Man-Lap Li, P. Ramachandran, S.V. Adve, V.S. Adve, and Yuanyuan Zhou. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks (DSN)*, pages 70–79, June 2008.

[SM04] V.K. Stefanidis and K.G. Margaritis. Algorithm Based Fault Tolerance : Review and experimental study. In *Proceedings of the International Conference of Numerical Analysis and Applied Mathematics*, 2004.

[SMF06] Yasser Sedaghat, Seyed Ghassem Miremadi, and Mahdi Fazeli. A Software-Based Error Detection Technique Using Encoded Signatures. In *21st IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'06)*, pages 389–400. IEEE Computer Society, 2006.

[SMM+09] Michael Steindl, Jürgen Mottok, Hans Meier, Schiller Frank, and Markus Früchtl. Diskussion des Einsatzes von Safely Embedded Software in FPGA-Architekturen. In *Proceedings of the 2nd Embedded Software Engineering Congress*, pages 655–661, 2009.

[SMM10a] Michael Steindl, Jürgen Mottok, and Hans Meier. SES-based Framework for Fault-tolerant Systems. In *Proceedings of the 8th*

IEEE Workshop on Intelligent Solutions in Embedded Systems, Heraklion, Greece, 2010.

[SMM+10b] Michael Steindl, Jürgen Mottok, Hans Meier, Schiller Frank, and Markus Früchtl. Safeguarded Processing of Sensor Data. In *Embedded Real Time Software and Systems (ERTS2 2010)*, Toulouse, France, 2010.

[sof09] SoftFloat. http://www.jhauser.us/arithmetic/SoftFloat.html, 2009.

[SPW09] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: a Large-scale Field Study. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 193–204, New York, NY, USA, 2009. ACM.

[SSF09] Ute Schiffel, Martin Süßkraut, and Christof Fetzer. AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware. In *The 28th International Conference on Computer Safety, Reliability and Security (SafeComp 2009)*, 2009.

[SSSF10a] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *The 29th International Conference on Computer Safety, Reliability and Security (SafeComp 2010)*, 2010.

[SSSF10b] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. Slice Your Bug: Debugging Error Detection Mechanisms using Error Injection Slicing. In *Eighth European Dependable Computing Conference (EDCC 2010)*, 2010.

[SSSF10c] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. Software-Implemented Hardware Error Detection: Costs and Gains. In *The Third International Conference on Dependability (DEPEND 2010)*, 2010.

[SSSL10] Pramod Subramanyan, Virendra Singh, Kewal K. Saluja, and Erik Larsson. Energy-efficient fault tolerance in chip multiprocessors using Critical Value Forwarding. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010*. IEEE Computer Society, 2010.

[TLH+07] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: diagnosing production run failures at the user's site. *SIGOPS Oper. Syst. Rev.*, 41(6):131–144, 2007.

[TMBS10] Avi Timor, Avi Mendelson, Yitzhak Birk, and Neeraj Suri. Using Underutilized CPU Resources to Enhance Its Reliability. *IEEE Trans. Dependable Secur. Comput.*, 7(1):94–109, 2010.

[VA06] Ramtilak Vemu and Jacob A. Abraham. CEDA: Control-flow Error Detection through Assertions. In *IOLTS '06: Proceedings of the 12th IEEE International Symposium on On-Line Testing*,

pages 151–158, Washington, DC, USA, 2006. IEEE Computer Society.

[VFM06] Alireza Vahdatpour, Mahdi Fazeli, and Seyed Ghassem Miremadi. Experimental Evaluation of Three Concurrent Error Detection Mechanisms. In *Proceedings of the International Conference on Microelectronics (ICM)*, pages 67–70, 2006.

[VHM03] Rajesh Venkatasubramanian, John P. Hayes, and Brian T. Murray. Low-Cost On-Line Fault Detection Using Control Flow Assertions. *Proceedings of the 9th IEEE On-Line Testing Symposium (IOLTS)*, 00:137, 2003.

[VPC02] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. *SIGARCH Comput. Archit. News*, 30(2):87–98, 2002.

[WA01] Chris Weaver and Todd M. Austin. A Fault Tolerant Approach to Microprocessor Design. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 411–420, Washington, DC, USA, 2001. IEEE Computer Society.

[Wal00] Martin G. Walker. Modeling the wiring of deep submicron ICs. *IEEE Spectr.*, 37(3):65–71, 2000.

[Wap07] Data processing arrangement comprises coding device, which is adjusted to assign codeword to data item to be stored in memory element based on signal information. Patent pending: DE102007040721A1 applied for by Technische Universität Dresden, 2007. Inventors: Ute Wappler and Christof Fetzer.

[WB97] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *J. ACM*, 44(6):826–849, 1997.

[WBB+05] J.D. Wilkinson, C. Bounds, T. Brown, B.J. Gerbi, and J. Peltier. Cancer-radiotherapy equipment as a cause of soft errors in electronic equipment. *Device and Materials Reliability, IEEE Transactions on*, 5(3):449–451, Sept. 2005.

[WCS09] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Mixed-mode multicore reliability. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 169–180, New York, NY, USA, 2009. ACM.

[Web08] Charles F. Webb. IBM z10: The Next-Generation Mainframe Microprocessor. *IEEE Micro*, 28:19–29, 2008.

[WEMR04a] Christopher Weaver, Joel Emer, Shubhendu S. Mukherjee, and Steven K. Reinhardt. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 264, Washington, DC, USA, 2004. IEEE Computer Society.

[WEMR04b] Christopher T. Weaver, Joel Emer, Shubhendu S. Mukherjee, and Steven K. Reinhardt. Reducing the Soft-Error Rate of a High-Performance Microprocessor. *IEEE Micro*, 24(6):30–37, 2004.

[Wes00] Tim Wescott. PID without a PhD. *Embedded Systems Programming*, 13(11), 2000.

[WF06] Ute Wappler and Christof Fetzer. Hardware Fault Injection Using Dynamic Binary Instrumentation: FITgrind. In *Proceedings of the Sixth European Dependable Computing Conference (EDCC 2006) [Fast Abstract]*, volume Proceedings Suplemental, 2006.

[WF07a] Ute Wappler and Christof Fetzer. Hardware Failure Virtualization Via Software Encoded Processing. In *5th IEEE International Conference on Industrial Informatics (INDIN 2007)*, 2007.

[WF07b] Ute Wappler and Christof Fetzer. Software Encoded Processing: Building Dependable Systems with Commodity Hardware. In *The 26th International Conference on Computer Safety, Reliability and Security (SafeComp 2007)*, 2007.

[WM08a] Ute Wappler and Martin Müller. Software Protection Mechanisms for Dependable Systems. *Design, Automation and Test in Europe (DATE '08)*, 2008.

[WM08b] Kai-Chiang Wu and Diana Marculescu. Power-aware soft error hardening via selective voltage scaling. In *ICCD*, pages 301–306. IEEE, 2008.

[WP06] Nicholas J. Wang and Sanjay J. Patel. ReStore: Symptom Based Soft Error Detection in Microprocessors. In *IEEE Transactions on Dependable and Secure Computing*, pages 30–39, Washington, DC, USA, 2006. IEEE Computer Society.

[WQRP04] Nicholas J. Wang, Justin Quek, Todd M. Rafacz, and Sanjay J. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 61, Washington, DC, USA, 2004. IEEE Computer Society.

[WsKWY07] Cheng Wang, Ho seop Kim, Youfeng Wu, and Victor Ying. Compiler-Managed Software-Based Redundant Multi-threading for Transient Fault Detection. In *International Symposium on Code Generation and Optimization (CGO)*, 2007.

[XCKI01] Jun Xu, Shuo Chen, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. An Experimental Study of Security Vulnerabilities Caused by Errors. *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 00:0421, 2001.

[Yeh96] Y Yeh. Triple-triple redundant 777 primary flight computer. In *In Proceedings of the 1996 IEEE Aerospace Applications Conference 1*, pages 293–307, 1996.

[YGS09] Jing Yu, Maria Jesus Garzaran, and Marc Snir. ESoftCheck: Removal of Non-vital Checks for Fault Tolerance. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 35–46, Washington, DC, USA, 2009. IEEE Computer Society.

[YMOS07] Mahmut Yilmaz, Albert Meixner, Sule Ozev, and Daniel J. Sorin. Lazy Error Detection for Microprocessor Functional Units. In *DFT '07: Proceedings of the 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, pages 361–369, Washington, DC, USA, 2007. IEEE Computer Society.

[YO09] Chengmo Yang and Alex Orailoglu. Processor reliability enhancement through compiler-directed register file peak temperature reduction. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009*, pages 468–477. IEEE Computer Society, 2009.

[YS96] Charles R. Yount and Daniel P. Siewiorek. A Methodology for the Rapid Injection of Transient Hardware Errors. *IEEE Trans. Comput.*, 45(8):881–891, 1996.

[YWZ04] Wenbin Yao, Dongsheng Wang, and Weimin Zheng. A Fault-Tolerant Single-Chip Multiprocessor. In Yew and Xue [YX04], pages 137–145.

[YX04] Pen-Chung Yew and Jingling Xue, editors. *Advances in Computer Systems Architecture, 9th Asia-Pacific Conference, ACSAC 2004, Beijing, China, September 7-9, 2004, Proceedings*, volume 3189 of *Lecture Notes in Computer Science*. Springer, 2004.

[ZCM+96] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin. IBM experiments in soft fails in computer electronics (1978–1994). *IBM J. Res. Dev.*, 40(1):3–18, 1996.

[ZL09] Chang N. Zhang and Xiao Wei Liu. An Algorithm Based Fault Tolerant Scheme for Elliptic Curve Public-Key Cryptography. In *DEPEND '09: Proceedings of the 2009 Second International Conference on Dependability*, pages 28–33, Washington, DC, USA, 2009. IEEE Computer Society.

[ZM06] Q. Zhou and K. Mohanram. Gate sizing to radiation harden combinational logic. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2006.

[ZMM+06] Ming Zhang, Subhasish Mitra, T.M. Mak, Norbert Seifert, Nicholas Wang, Quan Shi, Kee Sup Kim, Naresh Shanbhag, and Sanjay Patel. Sequential Element Design with Built-In Soft Error Resilience. *IEEE Trans. VLSI*, December 2006.