Black One Sugar

A Fault-Tolerant Virtual Machine

Master's Thesis

Martin Fjordvald and Sebastian Lybæk

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Ø
http://www.cs.aau.dk/

Title:
Black One Sugar - A Fault-tolerant Virtual Machine

Theme:
Scientific Theme

Project Period:
Fall Semester 2014

Project Group:
d102f14

Participant(s):
Martin Fjordvald
Sebastian Lybæk

Supervisor(s):
René Rydhof Hansen
Mads Christian Olesen

Copies: 5

Page Numbers: 85

Date of Completion:
June 3, 2014

Abstract

The transistors in computers are decreasing in physical size for every generation resulting in an increased susceptibility to single event upsets. The rate of single event upsets causing an error in a system intensifies when ascending from sea level and to space. This increases the need for fault-tolerance implementations in modern computer systems. This project documents the process of implementing a fault-tolerant virtual machine based on semantics defining the virtual machine's language as well as the fault model. The fault tolerance techniques used for the virtual machine includes Hamming codes, AN-codes, and Hamming distance. The fault-tolerance techniques has been carefully picked out based on previous analysis and implemented into the virtual machine at well defined areas. The virtual machine has been made in three different versions which are tested against each other for data on overhead in runtime. A fault injection framework has been developed using aspect oriented programming that can inject faults in our predefined areas. Results and findings are discussed and lastly suggestions for improvements are provided.

Contents

Chapter 1

Introduction

Each hardware generation brings increasingly better and faster processors and micro controllers running at lower voltage levels, and features more transistors arranged on a decreasing area. While this is a great evolution for performance and cost it comes at a price. The susceptibility to faults caused by radiation increases as well and therefore the requirement for fault-tolerance implementation becomes increasingly relevant. There are many examples of faults caused by these seemingly random upsets that, if not taken seriously, can prove costly to correct. An example of this is the AAUSAT3 [3] which was almost lost due to a single event upset (SEU). The satellite stopped responding to requests from the ground station because the electronic power supply failed. Any attempt to restart was also impossible due to the weakened system. Only by consulting the whole development team and some luck they where able to restart the satellite by utilizing an exploit.

Another example was an incident reported by Sun Microsystems involving servers of sites such as America Online and eBay which crashed due to radiation [6]. For a company whose livelihood depends on server availability, randomly occurring downtime can potentially prove costly which emphasizes the need for reliable systems. Even though these kinds of upsets are increasing with each generation they are still not very frequent for computers at ground level [19]. This is not the case in space however, where the radiation levels are much higher than within the atmosphere. Because of this there is a much higher risk involved when using software on space equipment. For non-critical missions less sophisticated methods can be used to recover after faults, as is the case with AAUSAT, which uses a watchdog that will restart the system if required. This is however not a valid solution for safety-critical missions. A Credible source tells that the satellite can restart multiple times an hour. By implementing sufficient fault-tolerance it should be possible to significantly reduce the amount of restarts needed.

1.1 Project Description

The work done in [8] documents an investigation of different techniques to implement fault-tolerance in a system. In this project we are building upon this work, by implementing a VM that features fault-tolerance backed by semantics of a fault-model. The goal is to provide a VM with high availability, The definition of availability that we use is from [5].

We have chosen to implement a VM as it allows us to focus strictly on software-only recovery with a layer of indirection to the hardware level, all techniques will be working natively in the VM, and through the VM propagated to an underlying hardware platform, potentially the VM could be implemented directly on an FPGA.

To test our VM we are also including fault injection as a part of the project. The fault injection should be implemented in a way that isolates the additional code from the VM code to keep the VM clean. Since we need to test very specific types of errors the custom injection frameworks is deemed necessary.

Fault-tolerance is already being used on some micro-controllers but it is mainly based on hardware fault-tolerance. This increases the cost of each unit, this price could be brought down by using software-only fault-tolerance solutions executed on commodity hardware.

To sum up, we will implement software-only fault-tolerance techniques into a VM to harden it against SEUs.

Chapter 2

Project Scope

The following chapter defines the bounds of the project in the context of available hardware platforms, related work, a brief analysis of fault-tolerance techniques, and our fault-model.

2.1 Platforms

According to the GOMspace website [10], CubeSat satellites which AAUSAT is, features a processor with ARM7 architecture. We want to investigate different platforms that have similarities with ARM7 to get a detailed understanding of which features are provided by the different platforms and use this information to conclude if fault-tolerance is already existing for such platforms and if so how big the coverage is.

The ARM7 architecture: The ARM7 family was first introduced in 1994 and is to date the most widely used 32-bit embedded ARM processor family. Its main features is the low cost and low power consumption making it well fit for many embedded systems. The ARM7 family features no build in ECC i.e. Error Correcting Code, a hardware solution for fault-tolerance.

However today it is exceeded by newer generations of the ARM architecture. On ARMs website [4] there is a road map for upgrading an ARM7 to a newer product that either excels in performance or efficiency. Among these products are ARM9, ARM11, Cortex-M, Cortex-R, and Cortex-A. Even though the AAUSAT is running on an ARM7 it could be argued that in the near future an upgrade would be performed to one of the newer versions since they provide more computational power and in some cases with less power consumption.

The ARM9 architecture: This processor family is a single core processor. There are no fault-tolerance of the registers or cache. It features an MPU, which provides the feature to divide memory into regions with protection attributes associated with each region. The MPU only protects the cache against permission violations and is not able to detect any bit-flips. The area differs from 6.5 mm down to 0.2 mm however the smallest processor area is on processors without cache memory.

The ARM11 architecture: This processor family performance has an surface area varying from 1.17 to 3.26 mm. The ARM11MPCore processor has the ability to support from 1-4 cores. ARM11 does not feature any fault-tolerance or fault-protection. The ARM1156T2 also includes a MPU with the same limitations as for ARM9.

The Cortex-M achitecture: This processor family has a wide range of specifications and use. The Cortex-M0 and Cortex-M0+ has the lowest area and power consumption respectively of any ARM processor. Area-wise the processors starts at 0.56 mm and goes all the way down to 0.009 mm. None of the Cortex-M processors feature any fault-tolerance nor any MPU.

The Cortex-R achitecture: This processor family is well suited for real-time embedded systems which require a high availability and maintainability as well as fault-tolerance. The family consists of Cortex-R4, Cortex-R5 and Cortex-R7. The area of the core, ram and routing is 0.5 mm for the smaller processors and 0.7 mm for the R7 model. Most interesting for our project is the fault-tolerance of this processor family. Included on the cache and bus is the support for ECC. This means that the processor is able to detect and correct a fault caused by an SEU. The detection is done by utilizing a parity bit on each byte. Recovery involves forced write-through to ensure that no lines are dirty, and then reloading the faulty line from the original source. The correction system is limited to 1-bit errors.

The Cortex-A achitecture: This processor family consists of 7 processors used for various systems. The area based on Cortex-A5 and Cortex-A8 is in the range of 0.53 mm to just under 4 mm , the Cortex-A17 is claimed to be 28 nm but it is not stated if that includes caches and RAM. Most of the Cortex-A processors also includes ECC and parity checking in L2 memory, however it is not stated anywhere if the bus is also covered by ECC as is the case with Cortex-R5.

From our observations in the mentioned processors we can conclude that ECC although possible is not featured in a lot of processors and most notably not in the ARM7 processor used in AAUSAT. This gives us the motivation to try and make some fault-tolerance installments at software level to compensate for this lack of fault detection and correction within the processor.

2.2 Related Work

In this section we will describe other fault-tolerance projects that have similarities to our own project as well as discuss how they differ from this project.

There are many areas to investigate within the boundaries of fault-tolerance related to software. Besides the different way to introduce fault-tolerance using the different methods there are also other concerns like the effectiveness and requirement of fault-tolerant software or the cost of using fault-tolerant methods.

Shoestring

The article Shoestring: “Probabilistic Soft Error Reliability on the Cheap” [9] brings up the problem about manageable error rates for commodity hardware. A desktop computer or similar does not have the same requirements for availability, as the computer is not meant to run critical systems, which usually means the owner is not willing to spend additional money for the necessary software (or hardware) to avoid crashes. This calls for a requirement of very cheap solutions with little to no performance overhead. To meet this requirement a hybrid of two areas of research are used, namely symptom-based detection and software-based instruction duplication. Software-based instruction duplication provides a high coverage by duplicating every instruction and validating the result, however the overhead is high. To prevent this, symptom based detection is used where possible, since it is has a lower performance overhead, and only use instruction duplication when symptom based detection is not enough. The method is based on additional compiler passes. During these passes the instructions are analyzed and categorized by how vulnerable they are and if they are likely to generate symptoms. Then an additional pass will insert duplicates of instructions that are still at risk. The effectiveness of shoestring is reported to be around 80% of all unmasked faults. Unmasked faults has been measured to 8.1% of all faults. This approach is focusing on the runtime and minimal overhead while we are focusing on coverage and availability. The effectiveness of 80% is great for home computers but since our case is related to space equipment and satellites it would not be acceptable.

KESO and jino

Another approach is mentioned in the article: “A JVM for Soft-Error-Prone Embedded Systems” [23]. This approach focuses on the type safety of Java and to keep the spatial isolation intact even in the presence of a fault. Their fault model is also based around randomly occurring bit-flips. Fault-tolerance is achieved through the use of the KESO JVM in conjunction with its compiler jino. The approach is to lower the failure rate of the VM without taking a massive performance hit by utilizing several techniques. The approach uses many areas both at compiler level and runtime and it is based on an analysis of the runtime system. The fault-tolerance is limited to type safety which includes references, class identifiers, and virtual method tables as well as per-protection-domain data including static fields and heaps for each domain. The fault-tolerance is split into two categories: attack surface Reduction and reference checking. Attack surface is the term used in the article, and corresponds to our vulnerability surface which is defined as the memory area that the VM occupies that is vulnerable to bit-flips. The attack surface reduction is self explanatory, by reducing the attack surface the chance for a transient fault caused by radiation is also reduced. Reference checking uses a form of parity encoding of references to do parity checks. For reference checking three different variants are presented, one of which is not used so it will be omitted. `Dereference checking(DRC)` is decoding the reference every time it is dereferenced providing a high probability of detecting faults, however the checks has a probability of making unnecessary checks if the reference is used multiple times in a row. The alternative is `load reference check (LRC)` which checks references as they are loaded from memory. Once they are stored they are encoded and upon loading from memory they are decoded. LRC needs fewer checks than DRC but has a higher false-negative probability. The overhead in code is roughly 20% and the runtime increase is 30.71% in their tests for LRC. The method that they provide secures software based spatial isolation which is needed in other fault-tolerance techniques such as replication, so the overhead for a fully implemented fault-tolerant system is very likely to be higher than the benchmarks in this article. The contribution of this method is targeting the type safety and securing the spatial isolation in the presence of faults which is important but only covers a part of the entire VM, also even if the type safety is retained, values and instructions are still vulnerable to SEUs. The overhead is quite small compared to more heavy fault-tolerance techniques but is to be expected with the lower area coverage. We are aiming to also cover values, as faults in values can be very devastating and hard to discover once they have entered the system. The article tests the system by injecting bit-flips into each word of the allocated heap space but are pruning out all ineffective bit-flips.

We are also injecting faults in relevant memory areas but instead of using a predefined benchmarks we are creating our own fault injection using AspectJ to test exactly those areas that we cover with fault-tolerance.

State Machine Approach

The article: "A Fault-Tolerant Java Virtual Machine" [18] takes a different approach. They work with a state machine approach by defining a deterministic state machine which they replicate. Unfortunately a state machine of a JVM features non-determinism, so the paper suggest how to identify and eliminate the effects of non-determinism in the JVM. The fault-tolerance is introduced in the form of a redundant copy of the state machine. Each state machine will perform the same actions on the same input, and get the same results once non-determinism is eliminated. The state machine approach includes an abstraction of the output data by some internal way of determining the correct result in case of disagreement. The solution is based on a primary-backup architecture which means that the backup state machine logs the recovery information from the primary, and only runs if the primary fails. Compared to our project this approach is using a more theoretical approach by modeling the system as state machines. The faults covered in the state machine approach is more generally fail-stop failures while we are focusing on SEUs only. The Overhead of having two JVMs for fault-tolerance is depending on the application and was anywhere between 1-600%.

Fault-Tolerance Techniques

Besides these full VM solutions more general techniques for introducing fault-tolerance which are related to our project will now be described. The article [21] mentions three methods for fault-tolerance implemented at instruction level. While these are meant for implementing in a compiler it can also be modified to work with a VM. In [7] recovery blocks are mentioned, which is also related to fault-tolerance but contrary to the relatively low level instruction duplication, recovery block are more abstract and rely on an acceptance test to detect faults. Our fault-tolerance implementation is restricted to encoding and duplication of instructions and values. Recovery blocks higher abstraction level adds a layer of complexity in recovering from faults that we want to avoid.

2.3 Fault Model

Faults can be divided into two groups: Hard errors and soft errors. Our definition of soft errors are based on [16] and are errors in memory devices that are not permanent meaning that they can be corrected using software solutions. Our fault model is focused on SEUs caused by random external interferences. An SEU is defined as a single bit flipped at a random location in either main memory or a register. This bit-flip is caused by a small particle intersecting with a memory area. We are assuming that only a single bit-flip will happen in a word within the time interval it takes to correct it, which means that we will not deal with multiple flips at any time. We state that the action that causes an SEU has to be random. This means that we are not able to tell where in the system that the fault will happen. To make it more attainable we are slightly limiting the areas of the VM where an SEU can happen based on analysis of the VM to be able to focus on the most relevant areas. This also means that the fault location is not entirely random within the whole VM but will be random within the chosen areas.

We are only considering naturally caused faults by radiation, human attempts to attack the VM is not within the scope of this project. We assume that the fault caused by an SEU will have an impact of such caliber that it will generate a visible error and not simply be masked by the system.

Our fault model is based on our case and resembles a real space environment with cosmic radiation that can affect electronic equipment and hardware. However in order to limit the project's size to a manageable level we have made some assumptions and restrictions in the fault model. These restrictions are based on data from different papers about fault rates and fault occurrences. The assumption of a single bit-flip per correction cycle is based on [24]. This article is accounting for the fault rate of the Cassini spacecraft and the results are showing that the spacecraft could expect a fault rate of 280 errors a day with an increase to four times that amount during a solar flare. This fault rate is further categorized into single bit-flips and double bit-flips within the same word. The double to single bit-flip rate was about 0.7% so it is a negligible fault rate. We are also assuming that faults will not be masked which means that they will manifest as an error visible to the user. It is not all faults that becomes visible. In [26] it is mentioned that only 8 percent of faults within a computer system becomes visible to the user while the remaining 92 % is being masked by the system. Lastly we are assuming that SEUs will only happen in specific areas of the VM. This assumption might be naive but we are trying to target the most relevant and common faults to get the best availability.

2.4 Vulnerability Analysis

The overall goal of this project is to create a VM that can run even in the presence of SEUs. However, it is not easy to fully cover the entire VM without suffering a considerable overhead in both runtime and memory. An analysis of components of the VM is required to identify the vital points and what the benefits of implementing fault-tolerance are. In this section we will cover the different areas of the black one sugar Virtual Machine (bosVM) and point out the potential consequences of an SEU in each area.

Our VM ,bosVM, consists of a translator and execution engine. The translator will be skipped since its primary purpose is to mimic pre-compilation of TinyBytecode source files. We have divided the execution engine into the following areas:

- Instructions
- Program counter
- Operand stacks
- Local variables
- Call stack
- Meta-data

The instructions are a central part of the system. A large amount of instructions can be executed each second and if one of these instructions includes a bit-flip the system will be at risk of crashing or receive a silent data corruption i.e. an fault invisible to the user. To investigate the instructions further they can be split into opcodes and arguments.

Every instruction includes one opcode. There are also two other elements with similarities to opcodes, aritcodes and ifcodes, while opcodes are used to identify which type of TinyBytecode instruction is under execution, aritcodes are used to identify arithmetic operations, and ifcodes in the same way used to identify conditional branching operations, in parts of this report all the code types will be referred to by the common term codes. Common for all these codes are their sizes of one byte. During an SEU every bit in these bytes can be flipped resulting in a different code. Such upset in instruction opcodes can have two outcomes. The first is an unknown opcode resulting in the VM crashing, and the second would be that the new opcode is similar to an already existing opcode resulting in a different instruction being executed, which may also lead to a crash if the new opcode is not aligned with the old in terms of arguments and stack variables. The aritcodes and ifcodes face the same consequences when a bit-flip occurs, as instruction opcodes, either the VM will crash or another code will be run instead. If a bit-flip results in another code being used there is a chance of a silent data corruption. If for instance the VM executes an arithmetic instruction, but a bit-flip has occurred in the aritcode `add` making it a `sub` aritcode, the instruction will still be able to run because subtraction takes the same arguments as addition, but the result is not what is expected. Codes are present in every instruction and the effect of an SEU can be severe so this area will need fault-tolerance.

Values are of the type int, boolean, byte, or a reference. Each of these types can be expressed as an integer. SEUs can occur in every bit of an integer effectively changing its value. The consequence of changed values are hard to predict and depends on what the changed value is used for. A value used for arithmetic operations will produce a wrong result also called a silent data corruption which is quite severe. Values used for conditional branches can lead the program down an unexpected execution path which makes the execution unpredictable and can lead to both silent data corruptions or crashes. Some instructions have no values while others have two or more so the amount of values within the VM is dependent on the program, and with the severity of silent data corruptions that SEUs can cause it is desirable to also include fault-tolerance for values.

The program counter represents a small part of the overall virtual machine. However, if this is changed due to an SEU the outcome is very hard to predict. In some cases the program will be able to continue operating, but the state of the program is unknown and chances are that you have skipped important operations leading to an inconsistent state which might lead to other errors later in the execution. In other cases the VM will crash right away due to missing arguments or references since some instructions are skipped or run twice. Lastly there are rare cases where the system will continue running and no faults will be present leading to the fault being masked out. The severity of SEUs in the program counter calls for fault-tolerance even if the possibility of an SEU hitting the program counter is quite small.

The operand stack contains values of a frame and depending on what these operands will be used for the impact on the system can be anywhere between unnoticeable and disastrous. It is also varying how much space an operand stack will occupy as some frames can have many elements on the operand stack while other frames can have none. As described SEUs in values can lead to silent data corruptions. Overall throughout an execution of a VM the operand stacks from all the frames sums up to become a large enough part of the system, combined with the impact that SEUs can have, to require fault-tolerance.

The local variables consist of variables which are bound to values. As mentioned for values they will require fault tolerance due to the severe impact that an SEU can cause. Some local variable faults might be able to pass the system unnoticed but others can cause silent data corruptions or crashes, so they need to be able to tolerate faults.

The call stack is a stack with all the frames currently under execution. A frame consists of a program counter, operant stack, local variables and method. All these elements are covered earlier except for method which will be covered in meta-data.

Meta-data

The meta-data consists of a map of classes. This map uses keys to locate the values of the map. If the key is compromised due to an SEU a lookup in the map with the altered key would fail, or in the rare case of a match with another key a lookup would return the wrong class which could be viewed as a silent data corruption. The keys should therefore include fault-tolerance.

The meta-data classes contain a lot of additional information. In Figure 2.1 an overview is provided of the elements. Each element is analyzed in regards to the impact of an SEU.

Element	Sub-elements
Class	Name
	Super class
	Methods
	Fields
Method	Name
	Class
	Type
	Instructions
	Number of arguments
Field	Name
	Class
	Type

Figure 2.1: Meta data elements

Classes in the meta-data contain several elements these being the name of the class, a list of methods the class contains, a list of the fields contained by the class, and a super class that the class is derived from if such exists.

A class' name and its super class are connected in that the super class of a class is referenced by the name of that super class so by securing the name element of every class their super class element is also secured. The method list is also referenced by the name of the methods and the same goes for fields. The meta data is a substantial part of the system meaning a high vulnerability surface. The consequence of a bit-flip will result in an unknown name when referenced which will halt the VM, or in the rare case of a change of the name into another valid name the VM will reference the wrong class. Therefore it is of high priority to implement fault-tolerance.

The methods contains a method name, a type, the class which it came from, its instructions, and the number of arguments that it takes. The name needs to be tolerant to faults since a mismatch in a reference will halt the VM. The class from which it came is already covered by the class name already mentioned, as are the instructions. The number of arguments is simply an integer, meaning that the vulnerability surface is relatively small, but the impact of an SEU could be quite severe. If more arguments are expected than provided the VM will take whatever is left on the operant stack which can even create silent data corruptions. The type element is not utilized in our VM implementation so implementation of this as well as the related fault tolerance will be future work.

The last part of the meta-data are fields. They consist of a field name, a class from which they are derived, and a type. As mentioned just above the type element is irrelevant for the current implementation of the VM. The name is important because the consequences in the presence of an SEU are the same as with class and method names. The class from which it is derived is already covered by class name.

With this knowledge in mind we have decided that our fault tolerance implementation should be focusing on codes, values, and meta-data (with the exception of types).

2.5 Fault-Tolerance Analysis

With the areas defined in section 2.4, it is now relevant to look into different fault-tolerance methods and decide which can be of use in each area. The areas will be discussed in greater details and with regards to fault-tolerance below.

As mentioned in Chapter 1 the Cassini spacecraft [24] experienced and corrected about 10 SEU's pr hour. With this in mind we have decided on a philosophy for bosVM which focus on fault detection with low run-time overhead, while accepting slow fault-recovery.

2.5.1 Codes

Codes are a very simple structure and consist of only one byte. We have considered two techniques, using a parity bit or exploiting Hamming distances. A parity bit is an additional bit which will be set to 0 or 1 depending on if the amount of 1 bits are even or odd. In Figure 2.2 a parity bit, in parentheses, is added to the original string and set to 1 because the amount of 1 bits is 3. If a bit-flip would occur at one of the bits, including the parity bit, the parity integrity would be broken and thus a fault is detected.

Original		Parity bit
10010010	$\rightarrow$	(1)10010010

Figure 2.2: A parity bit, in parenthesis, added to a bit pattern.

There is, however, an even simpler method that is suitable for this area, Hamming distance. Hamming distance is not directly a method but merely an expression for a difference in bit patterns. Hamming distance can be utilized by making sure that a fault caused by a bit-flip will result in an error instead of being flipped to the code value of another instruction. A Hamming distance of two makes sure that in case of a single bit-flip it will not be possible to reach a valid code value. Neither parity bits nor Hamming distances have inherent error correction, however we can exploit the fact that bit patterns with a Hamming distance of two will always be more than a single bit-flip from each other. The reason for choosing Hamming distance is that the original codes does not have to be altered or extended with more bits for fault detection. Fault recovery can be done by consulting a reference copy of that particular part of the program, since codes are never intentionally modified.

2.5.2 Values

For value fault-tolerance three methods are considered. The three methods are Swift-R, Trump, and Mask [21]. These three methods are originally used in conjunction with a compiler and are based on the instruction level. Swift-R is a version of the original Swift introduced by [22]. and utilizes redundant instructions when loading and storing variables in the processor's registers. These redundant instructions ensures that if an SEU occurs it is possible to detect it. Swift uses one redundant instruction each time while Swift-R uses two redundant instructions, and Swift-R also includes a majority vote that makes recovery possible. Trump uses the same concept as Swift-R but instead of using two redundant copies it uses only one. This copy is then encoded using

AN-codes, which is a simple and efficient way to detect and correct errors in numeric values. AN-codes are arithmetic codes that are defined as

$$C = \{AN | N \in \mathbb{Z}, 0 \leq N < B\}$$

Arithmetic operations, here addition but any apply, on two integers N_0 and N_1 are then

$$AN_0 + AN_1 = R$$

Decoding is done by dividing

$$R/A$$

if the result is not a factor of A an error has occurred. The choice of A impacts how many errors can be corrected and the storage overhead. We use AN-codes only for error detection, and for this $A = 3$ is sufficient and cause minimal storage overhead [21].

Mask is a little different and only works on values that are statically known prior to their use. By setting these values right before use, potential SEUs that has happened prior to the execution will be masked and further problems can be avoided. A boolean value for instance can be presented as a byte and the first bit is determining if it is true or false, the remaining bits are 0s. By setting this boolean to the known value right before use all bits that should be 0 will be set to 0 negating any earlier SEUs.

Of the three methods described only Swift-R and Trump would be able to detect and correct a number of faults, while Mask is very cheap to run but also not able to recover non-static value faults. The primary argumentation for using Swift-R over Trump is that it can be used for all instructions whereas Trump can only be used on arithmetic expressions. Trump has a smaller overhead however so if we can use Trump or part of Trump instead of Swift-R we are able to reduce the required memory overhead. Our values are integers, booleans, bytes, and references, and while booleans will not normally be compatible with Trump in our bosVM a boolean is just a byte with all bits set to either 0 or 1, so by treating booleans as integer values as well we are able to encode the values. As Trump is specifically minded for instruction duplication and should be implemented in the compiler we cannot use Trump as it is, but what we can do is to use the fundamental encoding that Trump uses. AN-encoding stores enough information to be able to both detect and correct a potential fault, thus it is suitable for covering values with relatively little memory overhead.

2.5.3 Meta-data

The meta-data consists of names and values and while values are already covered the names also need to be fault-tolerant. Names are strings so AN-encoding cannot be used for these. In the context of the coding discussed in this section, the message is the uncoded data and the block is the coded message.

Two techniques for string encoding was considered, either to use variety of fault-tolerant CRC, or to encode all strings with Hamming codes. CRC is remainder division on a bit pattern or bit stream, which adds a minimal overhead and has simple error detection [20], it does however not have any inherent error correction. Error correction using CRC [17] requires a corrective bit pattern to be created and XOR'ed over the message. In contrast Hamming codes [25] is self-contained and support single error-correction or double error detection, the trade-off is simplicity in the error detection algorithm where Hamming codes are slower. We decided to implement Hamming codes for the purposes of this project as it was deemed counterproductive to add a potentially large vulnerability surface in the form of error correction tables which vary in size for each individual length CRC polynomial. Also, not all CRC polynomials can be used for error correction, which means new error correction viable CRC polynomials need to be calculated on the fly if encoding of strings above the length of pre-defined CRC polynomials is needed.

Hamming codes

Hamming codes is a class of linear binary codes. It works by adding a number of parity bits to the string that is being encoded. In Figure 2.3 is shown an example of the encoding of the string "aa" in the ASCII-US charset where the question marks represent the parity bit positions. The block length of a message encoded with Hamming codes is

$$2^r - 1$$

where r is the number of parity bits in the block, which means the messages max length is

$$(2^r - r) - 1$$

.

Input string:	aa
Bit pattern:	11000011100001
Parity bits:	5
Block(? on parity bit):	??1?000?0111000?011

Figure 2.3: Hamming code example, encoding "aa".

Each parity value is calculated based on the bit's position in the block by starting at the bit's position in the block, adding the following bits equal to the bits position including the bit itself i.e. one bit for p1, two bits for p2 and so on. After adding bits, continue by skipping bits equals to the bits positions and then adding and skipping in this manner until the end of the block is reached. When all parity values are calculated, by counting the amount of 1's, write a 1 in place of the parity bit if the parity value is an odd number, or write a 0 if it is even. Using even or uneven parity bits has no practical implications. Encoding the String "aa" will then become "x\tab", tab as in the horizontal tab character. the parity bit calculation of "aa" is shown in Figure 2.4.

$$\begin{aligned}
p1 &= & ?100010001 \\
& & 3 \rightarrow 1 \\
p2 &= & ?100110011 \\
& & 5 \rightarrow 1 \\
p4 &= & ?0001000 \\
& & 1 \rightarrow 1 \\
p8 &= & ?0111000 \\
& & 3 \rightarrow 1 \\
p16 &= & ?011 \\
& & 2 \rightarrow 0 \\
message &= & 1111000101110000011
\end{aligned}$$

Figure 2.4: Calculating parity bits p1...p16 for input string "aa".

Decoding the message is a simple as removing the parity bits, doing so will however not detect any errors. In order to detect errors the process described above is essentially reversed. For the sake of this example the bit at position 7 in the block will be flipped from 0 to a 1, making it

$$1111001101110000011$$

The parity values are then calculated in the same manner as when encoding, by adding and skipping bits, only this time the parity bit has a value. The parity value calculation is shown in Figure 2.5.

$$\begin{aligned}
p1 = & \quad 1101010001 \\
& \quad 5 \rightarrow 1 \\
p2 = & \quad 1101110011 \\
& \quad 7 \rightarrow 2 \\
p4 = & \quad 10011000 \\
& \quad 3 \rightarrow 4 \\
p8 = & \quad 10111000 \\
& \quad 4 \rightarrow 0 \\
p16 = & \quad 0011 \\
& \quad 2 \rightarrow 0
\end{aligned}$$

Figure 2.5: Calculating parity values.

The difference from encoding is that each parity bit which parity value is an uneven number is marked, and the parity bit's position is added with any other parity bits position with uneven parity value, in this example this is the $p1$,$p2$, and $p4$ parity bits. It follows that

$$1 + 2 + 4 = 7$$

which indeed is the bit position flipped in the block.

Chapter 3

TinyBytecode

In this chapter we introduce the semantics of our fault model, as well as the VM language TinyBytecode. The semantics serve the purpose of defining correct behavior of TinyBytecode programs, and expected fault occurrences and will serve as base for the implementation of our VM. This project is expanding on the semantics of the language TinyBytecode defined in the report [8]. The original TinyBytecode did not include exceptions and no fault model. For completeness sake we have included the domains of TinyBytecode. The following definitions, up until 3.4 are adopted from [8] with some additions. The method definition has been expanded with exception handlers which were not included in the original definition, also the syntax table in 3.2 has been expanded with a throw instruction which has also been added to the original syntax.

3.1 Structure

In this section a formal description of the notation used and the program structure will be presented. Following the notation of Hansen [12], programs are represented as abstract data structures which are accessed by special access functions. This provides a convenient way to extract relevant information as shown in Section 3.1.1.

3.1.1 Notation

Hansen defines a domain to be a set. With this set corresponding access functions are given to access and modify different elements of the set. For convenience the record notation is used to specify a domain with all its access functions:

$$\text{Dom} = (f_1 : \text{Dom}_1) \times \ldots \times (f_n : \text{Dom}_n)$$

The above equation defines the domain Dom with access functions f_i : Dom $\rightarrow$ Dom$_i$ for $1 \leq i \leq n$. The notation used to access an element f_i of $d \in$ Dom is written $d.f_i$ and changing the value of an element is written $d[f_i \mapsto v]$.

3.1.2 Types

Like Java bytecode, TinyBytecode is a strongly typed language. In the version of TinyBytecode presented here the primitive types are treated as is, the semantics have no special support for varying lengths or bounds. The complete type-system of TinyBytecode is shown in Figure 3.1. This includes a few select primitive types, reference types, and the method type. This should be sufficient for our purposes, while being easily expandable, if needed.

$$\begin{array}{lcl}
\text{Type} & ::= & \text{RefType} \mid \text{PrimType} \\
\text{PrimType} & ::= & \texttt{boolean} \mid \texttt{int} \mid \texttt{byte} \\
\text{RefType} & ::= & \text{ArrayType} \mid \text{ClassType} \\
\text{ClassType} & ::= & \text{ClassName} \\
\text{ReturnType} & ::= & \text{Type} \mid \texttt{void} \\
\text{MethodType} & ::= & \text{Type}^* \rightarrow \text{ReturnType} \\
\text{ArrayType} & ::= & \texttt{array}\ \text{ElemType} \\
\text{ElemType} & ::= & \text{PrimType} \mid \text{ClassType}
\end{array}$$

Figure 3.1: The type system TinyBytecode.

3.1.3 Program

In the JVM specification [15] packages are used to distinguish class implementation and support portability. For TinyBytecode the idea of packages was left out because the complexity of the program structure was to be as low as possible. Instead the program only contains a set of classes:

$$\text{Program} \;=\; (\mathit{classes} \;:\; \mathcal{P}(\text{Class}))$$

3.1.4 Class

Classes in Java are the main components constituting a program. Classes are instantiated as objects and provide ways of accessing the encapsulated information. A class is defined by a class name and the methods and fields defined by the class are accessed by their respective components. Furthermore, a class keeps reference to the class hierarchy by a `super` component. The class hierarchy is encoded by a function $super : \text{Class} \rightarrow \text{Class}_\perp$ which returns the superclass of a given class. In Java a special class exists, namely `Object`, which is implicitly the superclass of all classes except itself. Therefore by convention, using the *super* function defined before on the `Object` class will return the bottom value $\perp$.

$$\begin{aligned}\text{Class} \;=\; & (name : \text{ClassName}) \times \\ & (methods : \mathcal{P}(\text{Method})) \times \\ & (fields : \mathcal{P}(\text{Field})) \times \\ & (super : \text{Class}_\perp)\end{aligned}$$

A class inherits fields and methods of its superclass. In the Java specification [15] methods and fields can be static as well. For TinyBytecode this option has been removed for simplicity as well as abstract classes and interfaces. A class can provide its own implementation of an inherited method, and when such a virtual method is invoked, a lookup is performed at runtime to determine the class in the class hierarchy from which the implementation is to be found. This is also called *dynamic method dispatch* [15].

3.1.5 Method

Methods are identified by a class, a method name and its argument types. where the method has been defined and its type. In TinyBytecode the methods provide the functionality of a program by containing instructions needed to execute the method. These instructions are accessible through the function $instruction : \text{PC} \rightarrow \text{Instruction}_\perp$ which takes a program counter as parameter and returns the instruction at that program counter or the bottom value $\perp$, if no instruction is present. For later use a function *numArgs* is defined which returns the number of arguments the method has. The notation $\mathbb{N}_0$ is used to denote positive integers including zero. Also included are exception handlers which, as the name suggests contains the exception handlers if any is available from this method. This is an addition to the original adopted definition.

$$\begin{aligned}
\text{Method} \quad = \quad & (\mathit{class} \ : \ \text{Class}) \times \\
& (\mathit{name} \ : \ \text{MethodName}) \times \\
& (\mathit{type} \ : \ \text{MethodType}) \times \\
& (\mathit{instruction} \ : \ \text{PC} \rightarrow \text{Instruction}_{\perp}) \times \\
& (\mathit{numArgs} \ : \ \mathbb{N}_0) \times \\
& (\mathit{Exceptionhandlers} \ : \ \mathbb{N}_0 \rightarrow \text{ExceptionHandler})
\end{aligned}$$

The first instruction of a method is defined to be at program counter 0. To advance a program counter pc_i the program counter is incremented as pc_{i+1}.

3.1.6 Field

A field is identified by the class where it was defined, the name of the field and the type.

$$\begin{aligned}
\text{Field} \quad = \quad & (\mathit{class} \ : \ \text{Class}) \times \\
& (\mathit{name} \ : \ \text{FieldName}) \times \\
& (\mathit{type} \ : \ \text{Type})
\end{aligned}$$

3.2 Syntax

The syntax of TinyBytecode is a reduced set of Java bytecode instructions with a few modifications for simplicity. The JVM specification defines 149 instructions [14] that make up the Java bytecode language, these 149 instructions have been reduced to 20 instructions in TinyBytecode. Some instructions are generalized versions of their Java bytecode counterpart while others retain their Java bytecode format. The TinyBytecode language includes the basic stack operations, a notion of objects, as well as arrays and methods. The complete set of instructions is listed in Figure 3.2. Note that the throw instruction has been added to the adopted syntax.

			Imperative Core
TinyBytecode	::=	nop	No operation.
	|	push t v	Push v onto the stack; type t.
	|	pop $[n]$	Pops n elements off the stack.
	|	load t v_i	Load local variable v_i; type t.
	|	store t v_i	Store in local variable v_i; type t.
	|	dup	Duplicate the top element of the stack.
	|	swap	Swaps the two top elements of the stack.
	|	goto *pc*	Jump to program counter *pc*.
	|	if *op* goto *pc*	Conditional jump.
	|	inc v_i v	Increment local variable v_i by v.
	|	arit *op*	Number operation with operator *op*.
			Objects
	|	new c	Instantiate new object of class c.
	|	getfield *field*	Get value of object field.
	|	putfield *field*	Set object field.
			Methods
	|	invokevirtual m_0	Invoke virtual method m_0.
	|	return $[t]$	Return from a method with optional type t.
			Arrays
	|	anew t	Create a new array; element type t.
	|	alength	Length of an array.
	|	aload t	Load a value from an array; type t.
	|	astore t	Store a value in an array; type t.
			Exceptions
	|	throw	Throw exception

Figure 3.2: TinyBytecode instructions

3.3 Semantics

In Section 3.2 the syntax of TinyBytecode was introduced and in this section a formal semantics of that syntax will be provided. The constructed semantics is a small step structural operational semantics [13]. In the following sections the formal notation and domains will be introduced followed by the four parts of the semantics: imperative core, objects, methods and arrays.

Because the semantic rules here are all adopted from [8], only semantic rules for program counter manipulation is shown here for reference. The rest are included in the appendix A.

3.3.1 Domains

TinyBytecode is modeled according to the specification for the JVM [15] with some simplifications. A value is defined as either a number or a reference.

$$\text{Value} = \text{Ref} + \text{Number}$$

A number in TinyBytecode is simply expressed as an integer.

$$\text{Number} = \mathbb{Z}$$

A reference is a location on the heap or a null reference.

$$\text{Ref} = \text{Location} \cup \{\texttt{null}\}$$

Like the JVM, heap locations contain both arrays and objects.

$$\text{Heap} \quad = \quad \text{Ref} \rightarrow (\text{Array} + \text{Object})$$

An object is the instantiation of a class and thereby defined by it. Furthermore, an object contains the fields of its class which can be accessed by the function $fieldVal \;:\; \text{Field} \rightarrow \text{Value}$.

$$\begin{aligned} \text{Object} \quad = \quad & (class \;:\; \text{Class}) \times \\ & (fieldVal \;:\; \text{Field} \rightarrow \text{Value}) \end{aligned}$$

In TinyBytecode arrays are treated similarly to objects. The array object stores information about the element type and length of the array, as well as an access function *get* for retrieving elements of the array.

$$\begin{aligned} \text{Array} \quad = \quad & (type : \text{ArrayType}) \times \\ & (length : \mathbb{N}_0) \times \\ & (get : \mathbb{N}_0 \rightarrow \text{Value}) \end{aligned}$$

While the heap contains arrays and objects a separate area is needed for local variables in methods. Local variables can be primitive types, or references to heap elements, and they are stored in an area called local memory. Elements in local memory are not accessed by name, but simply by a number.

$$\text{LocalMemory} = \mathbb{N}_0 \to \text{Value}$$

TinyBytecode is modeled with two stacks: one local stack, the operand stack, for each method and the global stack which hold the frame under execution, as well as frames in queue. The operand stack is used for calculations and hold temporary values.

$$\begin{aligned}\text{CallStack} &= \text{Frame}^* \\ \text{OperandStack} &= \text{Value}^*\end{aligned}$$

We use two different notations for elements already on the stack and an element which is pushed on the stack as a result of a semantic rule. The notation $st_i : st_n$ is used for existing elements and the notation $st_i :: st_n$ is used for newly pushed elements. Additionally the notation $st_1 : \cdots : st_n$ is used to denote a range of elements.

A frame is written $\langle m, pc, lv, os \rangle$, its components are the current method under execution, the program counter, local memory and the operand stack. The program counter is a positive integer that points to next instruction of the method under execution.

$$\text{PC} = \mathbb{N}_0$$

Frames can now be defined as:

$$\text{Frame} = \text{Method} \times \text{PC} \times \text{LocalMemory} \times \text{OperandStack}$$

A TinyBytecode program will continue to execute as long as there are frames on the call stack. A running configuration can be defined as:

$$\text{RunConf} = \text{Heap} \times \text{CallStack}$$

A terminating configuration can be defined as:

$$\text{TermConf} = \text{Heap} \times \{\epsilon\}$$

The symbol ϵ denotes an empty stack. These two sets of configurations together give the definition of all valid configurations:

$$\text{Conf} = \text{RunConf} \cup \text{TermConf}$$

Finally, we can define the general form of the semantic reduction rules for TinyBytecode. For a program P ∈ Program, the reduction rules are of the form:

$$P, CP \vdash C \Rightarrow C'$$

In this rule $C, C' \in$ Conf. The CP element is the constant pool which is treated as a global object in TinyBytecode, and it contains information about types of methods and fields in classes.

3.3.2 Imperative Core

The core segment of TinyBytecode includes the instructions for stack manipulation as well as handling variables and values.

The instructions for manipulating the program counter are *nop*, *goto*, and *if*. The *nop* instruction does nothing but advance the program counter. The *goto* and *if* instructions both change the program counter to a specified value, with *if* only doing so if a conditional check succeeds. This check has the form $v_1\ op\ v_2$, where $op \in \{equals, nequals, grth, lsth\}$. In Figure A.3 the semantic rules for program counter manipulation are shown.

$$\text{nop}\ \frac{m.instruction(pc_i) = \text{nop}}{\begin{array}{c}P, CP \vdash \langle H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, os\rangle : CS\rangle\end{array}}$$

$$\text{goto}\ \frac{m.instruction(pc_i) = \text{goto}\ pc'}{\begin{array}{c}P, CP \vdash \langle H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc', lv, os\rangle : CS\rangle\end{array}}$$

$$\text{if}\ \frac{\begin{array}{c}m.instruction(pc_i) = \text{if}\ op\ \text{goto}\ pc_n \\ exp = v_1\ op\ v_2 \\ op \in \{equals, nequals, grth, lsth\},\ pc' = \begin{cases} pc_n & exp = true \\ pc_{i+1} & exp = false \end{cases}\end{array}}{\begin{array}{c}P, CP \vdash \langle H, \langle m, pc_i, lv, v_1 : v_2 : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc', lv, os\rangle : CS\rangle\end{array}}$$

Figure 3.3: Semantic rules for program counter manipulation.

The rest of the semantics can, as mentioned, be found in appendix A.

3.4 Exceptions

Exceptions in TinyBytecode are only supported with user defined exception handlers. If no such handler is defined within the call stack the system will stop. An exception handler consists of a type of the exception, start and end address which defines where the exception will happen, and a handler address which is where the exception handler is located. This is formally written as

$$\begin{aligned}\text{ExceptionHandler} \;=\; & (\mathit{catchType} : \text{Class}) \times \\ & (\mathit{startAddress} : \text{pc}) \times \\ & (\mathit{endAddress} : \text{pc}) \times \\ & (\mathit{handlerAddress} : \text{pc})\end{aligned}$$

An exception frame is a frame with an exception type, σ.

$$\text{Exception frame} =< \sigma >$$

The throw instruction is included in the TinyBytecode syntax. Exception handlers are treated as objects which belong to a subclass of the class `Throwable`. The throw instruction will put an exception frame on top of the operand stack. This state where an exception frame is the top element of an call stack will then check if the current method includes a valid handler. If so it will jump to that location and execute the exception handler, otherwise it will pop the top frame from the call stack and try to rethrow to the new top frame. The semantics for the throw instruction is shown in 3.4.

$$\text{throw}\;\frac{\begin{array}{c} m.instruction(pc_i) = \text{throw} \\ \sigma = H[loc].class \preceq throwable \quad loc \neq null \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, loc : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle \sigma\rangle :: \langle m, pc_i, lv, os\rangle : CS\rangle \end{array}}$$

$$\frac{handler(m, pc_i, \sigma) = \perp}{\begin{array}{c} P \vdash \langle H, \langle \sigma\rangle : \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle \sigma\rangle : CS\rangle \end{array}}$$

$$\frac{handler(m, pc_i, \sigma) = pc_x}{\begin{array}{c} P \vdash \langle H, \langle \sigma\rangle : \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_x, lv, os\rangle : CS\rangle \end{array}}$$

Figure 3.4: Semantic rules for the throw instruction.

There are several problems that must be addressed before we can proceed. First we need to know if the exception handler is even able to handle the exception type. Next we need to get the first handler that is able to handle the exception in case more than one handler is present. Last we want to get the handler's handlerAddress. Three helper functions, shown in Figure 3.5, are used to ensure that all these criteria are fulfilled.

$$\begin{aligned}
EHT &= \text{handler table} \\
h &= \text{specific handler} \\
handler(m, pc_i, \sigma) &= \begin{cases} EHT(i).handlerAddress \\ \quad \text{if } EHT = m.handlers \wedge \\ \qquad \exists i : isFirstHandler(EHT, i, pc_i, \sigma) \\ \perp \; otherwise \end{cases} \\
isFirstHandler(EHT, n, pc_i, \sigma) &= canHandle(EHT(n), pc_i, \sigma) \wedge \\
&\quad\; (\forall j : canHandle(EHT(j), pc_i, \sigma) \Rightarrow j \leq n) \\
canHandle(h, pc_i, \sigma) &= h.startAddress \leq pc_i \leq h.endAddress \wedge \\
&\quad\; \sigma \preceq h.catchType
\end{aligned}$$

Figure 3.5: Exceptions helper functions.

3.4.1 Runtime Exceptions

Besides the exceptions thrown by the programmer runtime exceptions are also defined in the semantics. Runtime exceptions are handled in the same way as user thrown exceptions by adding an exception frame to the call stack. This also implies that a user defined handler must be present or else the system will fail.

Run time errors can be of different types depending on which condition causes the exception, in TinyBytecode we have defined 4 different types:

AritExc includes arithmetic exceptions such as division by zero.

NullPointExc includes all references that may be null, such as object and class references

NegArraySizeExc includes exceptions introduced when trying to create a new array with a negative length

IndexOutOfBoundExc includes exceptions caused by trying to write or read outside an array's boundary

Formally expressed as:

$$\begin{aligned}\text{RuntimeException} \quad = \quad & \{AritExc,\\ & NullPointExc,\\ & NegArraySizeExc,\\ & IndexOutOfBoundExc\}\end{aligned}$$

Here are the semantic rules of runtime exceptions. They are all handled somewhat similar by adding an exception frame to the call stack, note that *aload* and *astore* have two possible exceptions namely *NullPointExc* and *IndexOutOfBoundExc*. Runtime exceptions semantics where applicable are shown in Figure 3.6, Figure 3.7, Figure 3.8, and Figure 3.9.

$$\text{new}\,\frac{\begin{array}{c} m.instruction(pc_i) = \text{new } c \\ c = null \quad \sigma = NullPointExc \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle \sigma\rangle :: \langle m, pc_i, lv, os\rangle : CS\rangle \end{array}}$$

$$\text{getfield}\,\frac{\begin{array}{c} m.instruction(pc_i) = \text{getfield } field \\ obj = H[ref] \quad ref = null \quad \sigma = NullPointExc \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, ref : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle \sigma\rangle :: \langle m, pc_i, lv, ref : os\rangle : CS\rangle \end{array}}$$

$$\text{putfield}\,\frac{\begin{array}{c} m.instruction(pc_i) = \text{putfield } field \\ obj = H[ref] \quad ref = null \quad \sigma = NullPointExc \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, ref : val : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle \sigma\rangle :: \langle m, pc_i, lv, ref : val : os\rangle : CS\rangle \end{array}}$$

Figure 3.6: Runtime exceptions semantics for concerning objects.

$$\text{arit}_{binary}\,\frac{\begin{array}{c} m.instruction(pc_i) = \text{arit } op \\ op \in \{div, rem\} \quad v_2 = 0 \quad \sigma = AritExc \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, v_1 : v_2 : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle \sigma \rangle :: \langle m, pc_i, lv, v_1 : v_2 : os\rangle : CS\rangle \end{array}}$$

$$\text{invokevirtual}\,\frac{\begin{array}{c} m.instruction(pc_i) = \text{invokevirtual } m_0 \\ obj = H[ref] \quad ref = null \quad \sigma = NullPointExc \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, ref : v_1 : \cdots : v_n : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle \sigma \rangle :: \langle m, pc_i, lv, ref : v_1 : \cdots : v_n : os\rangle : CS\rangle \end{array}}$$

Figure 3.7: Runtime exceptions for arithmetics and method invocation.

$$\text{anew}\,\frac{\begin{array}{c} m.instruction(pc_i) = \text{anew } t \\ size < 0 \quad \sigma = NegArraySizeExc \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, size : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle \sigma \rangle :: \langle m, pc_i, lv, size : os\rangle : CS\rangle \end{array}}$$

$$\text{alength}\,\frac{\begin{array}{c} m.instruction(pc_i) = \text{alength} \\ aref = null \quad \sigma = NullPointExc \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, aref : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle \sigma \rangle :: \langle m, pc_i, lv, aref : os\rangle : CS\rangle \end{array}}$$

Figure 3.8: Runtime exception semantics for `anew` and `alength`.

$$\text{aload}\;\frac{\begin{array}{c} m.instruction(pc_i) = \text{aload } t \\ \sigma = \begin{cases} NullPointExc \; if \; aref = null \\ IndexOutOfBoundExc \; if \; len < 0 \vee len \geq H(aref).length \end{cases} \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, aref : index : os \rangle : CS \rangle \Rightarrow \\ \langle H, \langle \sigma \rangle :: \langle m, pc_i, lv, aref : index : os \rangle : CS \rangle \end{array}}$$

$$\text{astore}\;\frac{\begin{array}{c} m.instruction(pc_i) = \text{astore } t \\ \sigma = \begin{cases} NullPointExc \; if \; a = H[null] \\ IndexOutOfBoundExc \; if \; len < 0 \vee len \geq H(aref).length \end{cases} \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, aref : index : val : os \rangle : CS \rangle \Rightarrow \\ \langle H, \langle \sigma \rangle :: \langle m, pc_i, lv, aref : index : val : os \rangle : CS \rangle \end{array}}$$

Figure 3.9: Runtime exception semantics for `aload` and `astore`.

3.4.2 Fault Model Semantics

Our fault model is defined as SEUs occurring at random areas in a system. We have made some semantic rules for where the SEUs can happen and what effect it will have on the system.

Once an SEU occurs the configuration of the program is altered written as: $P \vdash C \Rightarrow_\phi C'$

$\phi \in Fault$

We have defined the locations in a VM where SEUs can happen:

SEUIOC are SEUs happening in an instruction's codes.

SEUIV are SEUs happening in values in the stack.

SEULV are SEUs happening in the local variables of a frame

SEUPC are SEUs happening in the program counter

Formally written as

$$\begin{aligned}\text{Fault} \;=\; & \{\text{SEUPC},\\ & \text{SEUIOC},\\ & \text{SEULV},\\ & \text{SEUIV}\}\end{aligned}$$

The semantic rules for the various SEUs types are shown in figures 3.10, 3.11, 3.12, 3.13, 3.14, and 3.15. Note that the faults are changing the program state and will persist in the system until corrected. This means that we assume that corrective action must be taken in order for the VM to continue execution, and faults will not be masked by the system, which follows our fault model defined in 2.3

An element E altered by a bit-flip is written as $E^{(b)}$

A bit-flip, $E \equiv_1 E^{(b)}$, means that E is differing from $E^{(b)}$ by one bit such that E xor $E^{(b)} = 2^i$ where i is the bit that has been flipped.

$$\text{SEUIOC}\;\frac{P' = P[m.instructionAt(pc_i).opcode \rightarrow m.instructionAt(pc_i).opcode^{(b)}]}{\begin{array}{c}\vdash \langle P, H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow_{SEUIOC}\\ \langle P', H, \langle m, pc_i, lv, os\rangle : CS\rangle\end{array}}$$

Figure 3.10: SEUIOC semantic rule: opcode.

$$\text{SEUIOC}\;\frac{\begin{array}{c}P.m.instructionAt(pc_i) = \text{arit op}\\ \text{op} \in \{add, mul, div, rem, shl, shr, ushr, and, or, xor\}\\ P' = P[m.instructionAt(pc_i).op \rightarrow m.instructionAt(pc_i).op^{(b)}]\end{array}}{\begin{array}{c}\vdash \langle P, H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow_{SEUIOC}\\ \langle P', H, \langle m, pc_i, lv, os\rangle : CS\rangle\end{array}}$$

Figure 3.11: SEUIOC semantic rule: aritcode.

$$\text{SEUIOC}\ \frac{\begin{array}{c} P.m.instructionAt(pc_i) = \text{if op goto pc} \\ \text{op} \in \{equals, nequals, lsth, grth\} \\ P' = P[m.instructionAt(pc_i).op \rightarrow m.instructionAt(pc_i).op^{(b)}] \end{array}}{\begin{array}{c} \vdash \langle P, H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow_{SEUIOC} \\ \langle P', H, \langle m, pc_i, lv, os\rangle : CS\rangle \end{array}}$$

Figure 3.12: SEUIOC semantic rule: ifcode.

$$\text{SEUIV}\ \frac{}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, os_1 : v : os_2\rangle : CS\rangle \Rightarrow_{SEUIV} \\ \langle H, \langle m, pc_i, lv, os_1 : v^{(b)} : os_2\rangle : CS\rangle \end{array}}$$

Figure 3.13: SEUIV semantic rule.

$$\text{SEULV}\ \frac{\begin{array}{c} lv : \mathbb{N} \rightarrow \mathbb{N} \\ \exists! n \in \mathbb{N} \\ lv' = lv[n \rightarrow lv(n)^{(b)}] \end{array}}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow_{SEULV} \\ \langle H, \langle m, pc_i, lv', os\rangle : CS\rangle \end{array}}$$

Figure 3.14: SEULV semantic rule.

$$\text{SEUPC}\ \frac{}{\begin{array}{c} P \vdash \langle H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow_{SEUPC} \\ \langle H, \langle m, pc_i^{(b)}, lv, os\rangle : CS\rangle \end{array}}$$

Figure 3.15: SEUPC semantic rule.

This concludes the chapter on semantics. The semantics now consists of rules for the imperative core, object manipulation, methods, arrays, and exceptions both thrown and runtime exceptions. The fault model has also been specified with semantics clarifying where faults can enter the VM. These semantics are used as the foundation for implementing the VM.

Chapter 4

The bosVM Virtual Machine

This chapter documents the bosVM prototype implemented as part of this project. The name, bosVM is shorthand for black-one-sugar Virtual Machine, this name was given as it follows the Java tradition of coffee related naming, with black being a metaphor for the simplest of coffees, in this case a Java style byte-code language, upon which we have added some sugar, the fault-tolerance techniques.

4.1 The bosVM: a TinyBytecode VM

The bosVM implements a subset of the semantics of language as described in chapter 3, and adds upon this, various fault-tolerance techniques. The bosVM is implemented in Java 7 which is also the minimum version required for execution.

The implemented components of the VM, which in varying detail will be addressed in this chapter, are:

- Code interpretation
- Fault handling on operation codes
- Fault handling on computed values
- Integrity of meta-data fields

4.2 Baseline VM

We will first give a brief description of the implemented baseline, no fault-tolerance, bosVM. The following section will describe the implemented fault-tolerance techniques in the context of bosVM.

4.2.1 Code Interpretation

This section will briefly cover the interpretation and internal translation of TinyBytecode in bosVM, as it will help ease understanding of the implementation of some of the fault handling techniques.

The bosVM accepts input in the proper TinyBytecode format, as previously described. The internal translator parse the source file line by line, subtracting meta-data information, such as classes and their contained fields and methods, and attributes associated with these. The translator also translates, in essence compiles, the TinyBytecode source to the internally used bosVM byte-code format, which is an essential component in our implementation of fault-tolerance in the program code, this will be described in detail in subsection 4.3.1. The bosVM byte-code format is similar to that of TinyBytecode, and is stored in an map on a per method basis, which maps from the pc of an instruction to the actual instruction line, internally an object array. The bosVM byte-code format is shown in Figure 4.1.

bosVM byte-code format:			
Core			
nop:	[pc]	→	[opcode]
push:	[pc]	→	[opcode, typeCode, value]
pop:	[pc]	→	[opcode, arg(optional)]
load:	[pc]	→	[opcode, typeCode, addr]
store:	[pc]	→	[opcode, typeCode, addr]
goto:	[pc]	→	[opcode, int]
if:	[pc]	→	[opcode, ifCode, int]
arit:	[pc]	→	[opcode, aritCode]
Methods			
invokevirtual:	[pc]	→	[opcode, string]
return:	[pc]	→	[opcode, typeCode(optional)]
Objects			
new:	[pc]	→	[opcode, string]
getfield:	[pc]	→	[opcode, string]
putfield:	[pc]	→	[opcode, string]
Exception			
throw:	[pc]	→	[opcode, string] (NYI)

Figure 4.1: The bosVM byte-code format

The entries in Figure 4.1 should be read as: A given instruction will at program counter `pc` map to an object array with the elements listed. Listing 4.1 shows a simple TinyBytecode method and Listing 4.2 shows the corresponding bosVM byte-code translation and mappings of the method.

```
method int pusher()
      push int 42
      return int
```

Listing 4.1: A basic example of TinyBytecode

```
[0] -> [4,1,42]
[1] -> [19,1]
```

Listing 4.2: bosVM byte-code example

In the example in Listing 4.2, the instructions have been translated in accordance with Table 4.1, except for the value, 42, of the first line, which is maintained. The values in Table 4.1 all have a Hamming distance of two on the bit pattern level. This property is not used for anything in the baseline VM implementation, but was implemented this way for convenience as it does not have any impact on lookups or general functionality and performance of the VM. The columns, `Instruction`, `aop`, `ifop`, and `type` show only operations and instructions supported in the latest revision of bosVM. The `aop` column are arithmetic operations, used with the `arit` instruction, `ifop` are conditional branches used with the `if` instruction, and finally the `type` column represents type annotations.

Instruction	aop	ifop	type	(op)code	byte value
nop	add	equals	void	00000001	1
push	sub	nequals	int	00000010	2
pop	mul	grth	byte	00000100	4
load	div	lsth	bool	00000111	7
store	-	-	ref	00001000	8
goto	-	-	-	00001011	11
if	-	-	-	00001101	13
arit	-	-	-	00001110	14
invokevirtual	-	-	-	00010000	16
return	-	-	-	00010011	19
new	-	-	-	00010101	21
getfield	-	-	-	00010110	22
putfield	-	-	-	00011001	25

Table 4.1: the bosVM machine code Hamming distance 2 codes

4.2.2 Meta-data

The meta-data area in bosVM contains data needed for creating objects and frames for execution, and is created during the translation step. The meta-data object contains a map from class strings, obtained from the TinyBytecode source code of the program under execution, to **bosClass** objects, which in turn holds maps of methods and fields tied to this class as well as other class related data. Meta-data for fields and methods are contained in the **bosField** and **bosMethod** objects respectively. The **bosField** objects holds the name and type of the field, as well as the containing class. The **bosMethod** objects holds the method name, return type, containing class, and argument data, as well as the instructions of the method. Argument data is the number of arguments as well as their variable names.

4.3 The Fault-tolerant bosVM

The fault-tolerant bosVM implements several fault detection and fault recovery techniques, targeting different area of the VM. This section will cover the implementation of these techniques. Each techniques will be described and their pros and cons evaluated.

4.3.1 Hamming Distance for Operation Codes: Ham2

As mentioned in subsection 4.2.1, the TinyBytecode source code is translated into an internally used bosVM byte-code format for instructions, if operations, arithmetic operations, and type annotations, all individually sharing the property of having a Hamming distance of two to the nearest value on the bit pattern level. We use this property to detect errors in the instructions of method frames. Since there is a Hamming distance of two to the nearest value we can, under our fault-model, easily detect if an error has occurred as the code will not be recognized by the VM. If an error is detected the VM will fetch a backup copy of the line in question and replace it and continue execution. A simple example of a bit-flip in a `POP` opcode is shown in Figure 4.2, changing the byte value from 4 to 12, which in the lookup tables will return the enumerated opcode type `UNREC`, which represents an unrecognized opcode.

POP		UNREC
00000100	flips to	00001100

Figure 4.2: A bit-flip in the pop opcode, making it unrecognizable

Pros: Fault detection using Ham2 is very simple, assuming only a single bit-flip will occur, and if no faults occur this technique is practically free as it does not come with any code overhead.

Cons: Using a Hamming distance of two, means there is no inherent fault correction, which requires a Hamming distance of three, this would in turn mean that the amount of instructions we can support would be severely limited if we assume codes are a single byte value. The error correction used in context with Ham2 comes with a memory or storage overhead, which can become an issue if storage space is a valuable resource on the target system. Also, in the current revision of bosVM we assume the backup copy of the instructions are stored in a safe storage area, which may not be viable in a real-life scenario.

Considering the almost non existent cost of using Ham2, and that the downsides will only be a concern if the complexity of TinyBytecode is vastly increased, Ham2 is an fine candidate for protecting our codes. We assume that a satellite will need some kind of protected storage for its payload, and in turn assume it should be possible to use said protected storage for a redundant copy.

4.3.2 Fault Handling of Computed Values

The AN-code implementation is encapsulated in the **bosVM2.FaultHandling.AnCodeFaultHandler** class which handles encoding, decoding, and validity checks. Listing 4.3 shows the Java code for the TinyBytecode `arit add` instruction, with added AN-coding. The variables `v1` and `v2` are bound to the parameter values and `vc1` and `vc2` are bound as encoded values. The function first adds the coded values and does a validity check of the result and uses it if **true** is returned, if **false** is returned the non-coded values are added and used.

```
		private void doAdd() {
			int v1,v2,vc1,vc2,r;
			v1 = pop();
			v2 = pop();
			vc1 = an.enCode(v1);
			vc2 = an.enCode(v2);

			r = vc1+vc2;
			if(!an.valid(r)) {
				push(v1+v2);
			} else push(an.deCode(r));
		}
```

Listing 4.3: arit add with an-coding

The AN-codes used in bosVM is rather a simplification of AN-codes, since we do not implement the error corrective properties that are possible by calculating Hamming weight and arithmetic distance, however, with the fault-model used as the basis for this project, the implementation presented here should be sufficient for handling any of the expected SEUs.

4.3.3 AN-codes in Storage

AN-codes is not only used for calculated values, it is also used for stack storage, the program counter, and for numeric values in the meta-data. The storage solution is similar to a double redundancy solution, only that by encoding one of the redundant copies it also provide error correction. The implementation of the push instruction is shown in Listing 4.4. Whenever the `push` instruction is used, two values will be pushed upon the stack, one encoded, and the original. This is also the case for `pc` and `numArgs` for methods in the meta-data, either of which will also update both values when updated. The functions associated with the `pc` is also shown in Listing 4.4.

```java
private void push(Object a) {
        os.push(an.enCode((Integer) a ));
        os.push(a);
}

public int getpc() {
    if(pc[0] == 0 && pc[1] == 0) { return 0;}
    else {
          if(pc[0] != an.deCode(pc[1]) ||
            (pc[1]%3 != 0 && pc[0] == an.deCode(pc[1]))) {
        return setpc(an.handleValueFault(pc[0], pc[1]));
        }
        else {
             return pc[0];
        }
    }
}

private void pcInc() {
        pc[0]++;
        pc[1] = pc[1] + an.enCode(1);
}

//returns the new pc
private int setpc(int pc) {
        this.pc[0] = pc;
        this.pc[1] = an.enCode(pc);
        return pc;
}
```

Listing 4.4: TinyBytecode `push` and `pc` Java code

Pros: The way we have used AN-codes is somewhat lightweight, as it requires only little calculation, code, and memory overhead. For bosVM which only operates with numeric types, there is no operations where we can not use it.

Cons: The approach used is troublesome with zero values, and can cause overflow errors if the values are sufficiently large. Additionally, we skip the redundant calculation if no fault is detected, as we assume only one bit-flip will occur, as by the definition of our fault model. If no fault occurs this will have no consequence, in the unlikely event that a bit-flip occurs in both the coded calculation and in the uncoded values, we have no defense. By the definition of our fault-model, we are sufficiently covered.

In summary AN-codes have been included as they provide an efficient and easy way to detect single bit-flips without much overhead. With the fault-model defined for this project, we can provide satisfactory error detection and correction by using a light-weight implementation of AN-codes and redundant value copies.

4.3.4 Integrity of Meta-data Fields

The third and final area we have touched upon is the meta-data of the VM. The meta-data of the VM is slightly different in that faults in this area will persist for the duration of the execution and thus potentially cause cascading faults. The meta-data consist of a variety of fields, and as already mentioned in subsection 4.3.2 the numeric fields of the meta-data is already handled by using the AN-code implementation. However, the meta-data area primarily consists of several maps which save its key values as strings, and some objects in the meta-data also have string fields, on which we can not use AN-codes. Shown in Table 4.2 is an overview of the meta-data objects and their contained fields as well as their types. The **numArgs** field in the BosMethod object is an **Integer** array and as mentioned earlier handled through AN-coding. The **BosField** and **BosMethod** objects both contain a **type** field of the enumerated type **Type**, as bosVM completely ignores types in the current revision there has not been implemented any fault handling for these fields. However, the values of **Type** are in the Ham2 format, and in a future revision this technique could potentially be used. Finally there are several fields containing **String** values or **String** keys

4.3.5 Hamming Codes in bosVM

The Hamming codes string encoder and decoder is implemented in the **bosVM2.Faulthandling.StringEncoderDecoder** class, details will be omitted here but can be found in Appendix B.

Hamming codes are used for the meta-data area, which is constructed before execution of any program and receives no running updates. This means that the error detection and correction performed using Hamming codes must not only correct retrieved elements, but also make correcting actions in the meta-data area. Hamming codes applies to two variations of string fields in the meta data: String fields that hold names of objects, and strings which are keys in the **classes**, **methods**, and **fields** maps. The code in Listing 4.5 shows the **get** and **set** functions of class names in **bosClass** objects, with the added

Java object	Fieldname	Java type
Meta-data	Classes	map<String,BosClass>
BosClass	cname	String
	_super	BosClass
	methods	map<String,BosMethod>
	field	map<String,BosField>
BosMethod	clss	BosClass
	methodName	String
	type	Type
	numArgs	Integer[]
	args	String[]
	ins	map<Integer,Object[]>
BosField	clss	BosClass
	fieldName	String
	type	Type

Table 4.2: Overview of meta-data objects and contained fields

error correction measures. The **ham.deCode(cname)** call in line 3 returns an object array, the error array, containing a boolean value on index 0, and the decoded and corrected string on index 1. The boolean value tells if there has been corrected an error or not, and the meta-data field is corrected if necessary.

```
1         //ClassName
2         public String getClassName() {
3                 Object[] temp = ham.deCode(cname);
4                 if((boolean) temp[0]) {
5                         cname = ham.enCode((String) temp[1]);
6                 }
7                 return (String) temp[1];
8         }
9
10        public void setClassName(String s)
11        {
12                cname = ham.enCode(s);
13        }
```

Listing 4.5: Handling names in the meta-data area

The error array returned from the decoding function will also have the faulty string on index 2, if an error has been detected. The faulty string is used to correct faults in strings that serve as keys in the **classes**, **methods**, and **fields** maps of the meta-data. If a method is requested and is not located by the **getMethod(String b)** function shown in Listing 4.6, the **methodsMapsKeyErrorCheckAndCorrection()** functions is called, which will return a boolean value depending on if a fault has been found or not. After the corrective measures have been performed a new attempt at locating the method is tried, if the method is not found **null** is returned, which consequently will halt execution.

```java
public BosMethod getMethod(String b) {
    String name = ham.enCode(b);
    BosMethod out = methods.get(name);
    if(out == null) {
        if(methodsMapKeyErrorCheckAndCorrection()) {
            out = methods.get(name);
        } else { return null; }
    }
    return out;
}
```

Listing 4.6: The **getMethod function with error correction**

At any time the **methodsMapKeyErrorCheckAndCorrection()** function, shown in Listing 4.7, is called all possible single bit-flips in the keys of the map will be corrected as all keys will have to be checked since we can not know beforehand which string it is that is in error. Any key found to be in error will have its error array passed on to the **methodsMapKeyErrorCorrection(object[] errorArray)** function which will locate the faulty string in the map, using the faulty string passed on in the error array, and replace it with the value of correct key on index 1 in the error array.

```java
private boolean methodsMapKeyErrorCheckAndCorrection() {
    ArrayList<Object[]> error = null;
    for(String s: methods.getKeys()) {
        Object[] temp = ham.deCode(s);
        if((boolean) temp[0]) {
            if(error == null) {
                error = new ArrayList<Object[]>();
            }
            error.add(temp);
        }
    }
    if(error != null) {
        for(Object[] o: error) {
            methodsMapKeyErrorCorrection(o);
        }
        return true;
    } else { return false; }
}
```

Listing 4.7: Error correction function for keys in maps

4.4 Aspects

It is desirable to be able to inject faults at random areas of the VM at random time intervals, thus there needs to be a way of accessing and altering the executing code at runtime. This is because we try to simulate a running satellite which will be exposed to cosmic radiation resulting in SEUs. As mentioned earlier SEUs are unpredictable as to where they will strike and at what time. Another wanted feature is that the injection part is kept isolated from the VM meaning that we do not have to alter the VM in order to inject faults. This is for testing purposes and means that we will be able to run the tests on a VM that features no fault injection to compare the runtime overhead, as this overhead will not be present in a real world application. At first we tried using multi threading by having a thread running on the side of the main thread. The main thread would execute the VM while the side thread would randomly choose a moment to inject a fault at a random location. The problem was to get the timing synchronized and making the two treads run concurrently and still execute in an intertwined manner. One thread would usually finish before the other would start to run. Instead we shifted our focus to aspect oriented programming. Aspects has the advantage of being able to intercept method calls at runtime as well as returning relevant data be it variables or objects and it is completely isolated from the original code. All these features are well suited for our requirements.

Aspect oriented programming, as the name implies, is focusing on aspects of a program. A good example of an aspect is metrics. Metrics that are used for instance for generating logs for a running program are not concerning the application, it is just a functionality running on the side and the application should be unaffected by it. It is called a crosscutting concern because it is a functionality that are cutting across multiple different points in an object model. Metrics can be implemented by adding the needed code into the existing methods, however this causes some problems like maintaining the code or maybe turning certain functionality off, because the code belonging to metrics is sprinkled throughout the rest of the code. Aspects makes it possible to keep such code gathered at one place while still being able to dealing with crosscutting concerns.

Java has its own extension to aspect oriented programming called AspectJ [1]. AspectJ provides additional functionality to Java concerning crosscutting and other aspect oriented features. As seen in Figure 4.3 the code contained in aspects are injected in a program through an aspect weaver that inserts the desired code at specified join points.

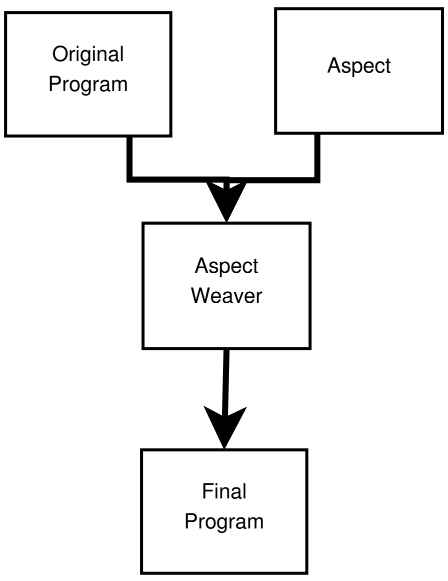

Figure 4.3: Model of the an aspect inserted into a source program

A program features several join points which are well-defined points during the execution of a program [2]. Such point includes method calls, exception handler calls and more, all these join points can be accessed by a pointcut. Pointcuts are very useful to inject additional code at runtime but needs another

```
	public Object executeFrame() {
			while(!(m.instructionAt(getpc()) == null))
			{
				//AspectJ hack: Makes fault injection possible
				getMethodFrame();
				...
			}
	}

	public MethodFrame getMethodFrame()
	{
		return this;
	}
```

Listing 4.8: The `getMethodFrame()` function and where it is inserted

concept before being able to do so, this being advises. Advises are associated with a pointcut and contains the desired code that will be injected. There are also different types of advises, the before advice for example will run right before the associated pointcut while an after advice will run right after.

By using AspectJ we are able to keep the injection of faults as well as logging separated from the rest of the code and thereby being able to easily turn it on and off.

Our random fault injection works by generating a random number every time an instruction is read and if the value rolled is below a threshold we inject a fault. The `MethodFrame` class executes these instructions within the method `executeFrame()`. We are not able to do a pointcut for `executeFrame` and get a cut at each instruction because these are organized within a loop, so to circumvent this we have made the method `getMethodFrame()` that will simply return the `MethodFrame` object and call it as the first thing from within the loop, as can be seen in Listing 4.8. This is an alteration of the original VM for the purpose of fault injection which we was trying to avoid, however it was deemed necessary and it is the only alteration leaving the rest of the VM untouched. With this new method it is possible to cut at each instruction executed, giving us the possibility to inject faults in the areas that we are targeting, the pointcut is shown in Listing 4.11. The method also returns the `MethodFrame` object which we can use to get access to all the required data that is to be injected. The injection is done in an `after() returning(MethodFrame m)` advice to use the returned `MethodFrame` as argument for the `inject(MethodFrame m)`. The inject method will then randomly select an injection type based on the input. An `opcode` injection is always applicable but `aritcode` an `ifcode` are only applicable if the instruction is a `arit` or `if` instruction respectively. Value

```java
	public int flip (int input)
	{
		int positionbyte = 1;
		positionbyte = positionbyte << rnd.nextInt(8);
		input = input^positionbyte;
		return input;
	}
```

Listing 4.9: Flip function

injection is only applicable if there is at least one value pushed to the operand stack so a check must be made before choosing this option. Since opcode injection is always an option this is the default injection type in case one of the others does not apply for the current instruction.

4.5 Fault Injection

Fault injection is important for our project for testing purposes. We need to be able to inject faults in order for the VM to detect and correct them. First we need to define which faults we want to be able to handle. Once we have specified that we need to be able to inject those faults. As stated in our fault model we are only concerned about SEUs. We also assume that we are able to correct the fault before a new one will enter the system.

The areas defined in section 2.4 are also the targets that we want to inject with SEUs. An SEU is defined in our fault model as a bit-flip in a random location of the VM. We are not targeting the VM completely at random though, but rather targeting our injection at the protected areas, in order to demonstrate that the implemented techniques have an affect. To help us flip a bit in a random location we have made a function called **flip(int input)**. The function can be seen in Listing 4.9. The function takes an integer as input rather than a byte, since we treat all internal numeric values in the VM as integers to avoid clashes with signed values, we will refer to the input as a byte. The variable `positionbyte` will start out as a byte of the value `00000001` which will then be left-shifted a random number of bits. The bit is flipped through an `XOR` of the position byte over the input byte, an operation demonstrated in Figure 4.4, the resulting byte is returned.

```java
        public String stringHandler(String s)
        {
                char[] a = s.toCharArray();
                int index = rnd.nextInt(a.length);
                a[index] = (char)flip((int)a[index]);
                s = String.valueOf(a);
                return s;
        }
```

Listing 4.10: String handler

Input:	01100010	00000111
	XOR	XOR
Position byte:	00100000	00000100
Result:	01000010	00000011

Figure 4.4: XOR function on bytes

The **flip(int)** function is the fundamental function in all injection methods. The only difference is the acquisition of the input byte. For program counter injection the program counter is send to the **flip(int)** function directly. For value injection each value is pushed on the stack two times and once required they are both popped. During this action they are compared to each other, therefore the whole operand stack is acquired to randomly select a value from both the originals and their copies. Instructions are organized in object arrays consisting of opcodes in the first position and depending on the instruction the remaining elements are put as the second position, third position etc. By getting the instruction at a certain program counter it is possible to get the opcode from the first position of the resulting object array. The opcodes are represented by a byte so it can be used by the **flip(int)** function. The aritcodes and ifcodes are similar to opcodes only they are located in an instruction at the second position of the resulting object array. These types of codes are only relevant in `arit` and `if` instructions respectively. The meta data is consisting mostly of strings so an assisting method has been created to deal with strings. The function can be seen in Listing 4.10, it takes a string and breaks it into a **char** array. Then selects a randomized index in that array to get a random character from the original string. This character is then flipped using the above mentioned **flip(int)** function resulting in a character flipped in a randomized bit location. The String is then reconstructed however with the flipped character.

With the string handler function it is easy to flip a string from the meta-data. This requires that there are getters and setters for the meta-data. We have provided these to be able to easily alter the meta-data. These functions however should not be available outside of fault injection. Another part of the meta-data are the keys used for the maps that contain all the meta-data. These keys are also vulnerable to bit-flips and should be injected for testing purposes. The string handler is used for this as well however the procedure of setting the value is a little different. If a key is set using traditional setters the new value will just be added as a new key for the map. Instead of setting the key directly must first delete the key and its value, and then creating a new entry with the new key which is flipped at a random bit location.

All injection related functionality is put into nine different methods that all utilizes the flip and string handler functions. These nine methods are then evenly distributed, as pseudo random numbers allow it, in a switch case that will randomly select an injection method to use. This function is called `inject()` and is called from the aspect.

4.6 The Logger

For output we have made a logger that will write a log containing information of detected and corrected faults during execution. This system also utilizes the aspect to cut at each fault handler method which can be seen in Listing 4.11 At the end some statistics are provided with information on how many faults where found and corrected of each fault type as well as the overall number of fault corrected and runtime to of the overall run through.

The log entries are consisting of a time of occurrence measured in milliseconds followed by a small text describing which type of fault was corrected at that specific time. The log entries are created in an after advice that is based on the logging pointcut. Whenever a fault handler is called the advice will be run, creating a log entry and sending the join point as a string as argument. This argument is then used to determine which of the eight fault handlers was run and create an appropriate log entry as well as increasing the count of faults. The final pointcut shown in Listing 4.11 called `done()` is used in an after advice that will generate the end of the log where all fault types are added and the final runtime of the run through is logged. All the log entries are written to an external log text file using a **PrintWriter**.

```
pointcut MethodFrame(): //used for injecting fault at every
    instruction
                call(MethodFrame getMethodFrame());

pointcut logging(): //used for logging purposes
                call(void handleOpcodeFault(int, BosMethod,
                    ExecutionEngine )) ||
                call(void handleAritcodeFault(int, BosMethod,
                    ExecutionEngine)) ||
                call(void handleIfcodeFault(int, BosMethod,
                    ExecutionEngine)) ||
                call(int handleValueFault(int, int)) ||
                call(char[] correctBitFlip(int ,char[]))||
                call(void mapKeyErrorCorrection(Object[])) ||
                call(void methodsMapKeyErrorCorrection(Object[])) ||
                call(void fieldsMapKeyErrorCorrection(Object[]));

pointcut done(): //used to finish the log with statistics
                call(void execute());
```

Listing 4.11: All Pointcuts

Chapter 5

Benchmarking

In this chapter we introduce some test cases which test the VM in both performance and abilty to correct faults.

5.1 Test Cases

To gauge the performance of our VM we have made three different test cases each based on the different versions of our VM. The different cases are:

Test case 1 - The baseline VM: This test case is performed on the baseline VM. We are interested in the runtime of a VM without any form of fault-tolerance or injection. We are also testing with injection enabled to see if the VM will actually crash as expected or the fault will somehow be masked by the VM.

Test case 2 - The Clean VM: This test case is performed on the Clean VM, we are interested in the runtime of the VM with fault-tolerance but no injections, this is important because our approach is to minimize the overhead in runtime as long as there are no faults present in the system.

Test case 3 - The VM with injection: This test case is performed on the VM with injection and logging enabled. This test is where the VMs fault-tolerance is tested through fault injection. Here we are interested in the fault rate at which faults are being injected as well as the runtime for completeness sake.

Each test case uses the same test input to be able to compare the results. This input is a test program written in TinyBytecode that exercises each possible operation available. The test program is then run a number of times in succession to simulate a longer runtime in order to come closer to a real world case. The test input can be seen in Listing 5.1. It instatiates three different objects and invokes their methods to extend the run time of a single run through and to increase the chance of rare faults happening. The first method `bubbleSort.bubbleMain.sort`, is a traditional bubble sort algorithm that greatly increases the runtime as well as using several if-statements which increases the chance of an ifcode fault which is very rare. The second method `simpleCalc.calc.fibonachi`, is calculating the Fibonachi number until reaching a certain threshold set to one million in this example. This method uses a lot of if-statements as well as arithmetic expressions thereby increasing the chance of an ifcode or aritcode injection. The last method `bos.sys.langtest.instest` includes the full instruction subset of TinyBytecode implemented in bosVM, to test that all valid instructions are able to run.

```
class testMain

method void main()
       new bubbleSort.bubbleMain
       store ref bmain0
       load ref bmain0
       invokevirtual bubbleSort.bubbleMain.sort
       new simpleCalc.calc
       store ref calc0
       push int 1000000
       push int 2
       push int 1
       load ref calc0
       invokevirtual simpleCalc.calc.fibonachi
       new bos.sys.langtest
       store ref ltest0
       push int 0
       load ref ltest0
       invokevirtual bos.sys.langtest.instest
       return
```

Listing 5.1: Test input main function

The tests are run on a Lenovo T520 with an intel i5-2430M quad core CPU running at 2.40 GHz x 4 and 4 GB of RAM. The operating system is Ubuntu 14.04. The tests are set up to run the the test program a number of times in succession.

5.2 Runtime Tests

First we look at the runtime of each version of the VM. Each test is run 10000 times in succession. The first round of runs is done by using a shell script to run the test program 10000 times. These results can be seen in Figure 5.1, all values are in milliseconds, which is the case for all test cases.

	Baseline	Clean	Aspect
Worst	84ms	259ms	362ms
Best	58ms	212ms	299ms
Average	63ms	231ms	320ms

Figure 5.1: Runtimes in milliseconds for 10000 continuous runs with Jit enabled.

Another test we did was to run the program 10000 times within a loop in a Java program to simulate a continuous execution run. For this to work properly the just-in-time compiler has been disabled since it would make each consecutive run faster and the data would not be comparable to the other test. The results can be seen in Figure 5.2.

	Baseline	Clean	Aspect
Worst	89ms	408ms	671ms
Best	63ms	382ms	403ms
Average	65ms	387ms	414ms

Figure 5.2: Runtimes in milliseconds for 10000 continuous run with Jit disabled.

These two tests shows that the difference between a baseline implementation of our VM and the clean implementation is roughly 6 times worse on average for when just-in-time is disabled and roughly 3.5 times worse on average with just-in-time enabled. Figures 5.3, 5.4, and 5.5 shows the spread of the results of Jit disabled runs, the red lines indicating the mean value. The plots are reduced to the first 1000 runs for the sake of readability. All the plots show fairly steady run times with a few spikes, The AspectJ runs have a far wider gap between its best and worsts run, also if ignoring spikes in the plot, which is not unexpected as the AspectJ runs include injection, detection, recovery, logging, and noise from AspectJ code. Since Jit disabled runs are the most realistic in a real world use case, we have not included the spread for Jit enabled runs.

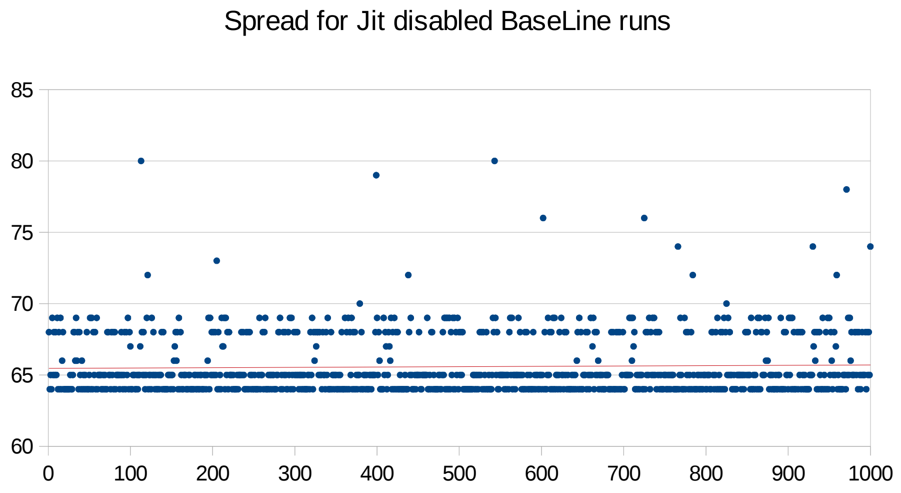

Figure 5.3: BaseLine Spread of runs.

Comparing the Jit enabled and disabled runtimes shows us that there is still things that can be improved in the implementation or on compile time.

5.3 Fault-tolerance Tests

Next we test the VM to see if it is able to tolerate the faults that we inject. In this test case we run the test program 15000 times in succession by looping the test program. Our fault rate is set to 2. This means that for each instruction executed by the VM there is a 2 in 10000 chance that the injector will inject a randomized fault, all in all this roughly translates to 1 injection every three seconds. This fault rate is much higher than a real world scenario so if the VM can handle this test it should also be able to handle a real world situation. The results of the test can be seen in Figure 5.6

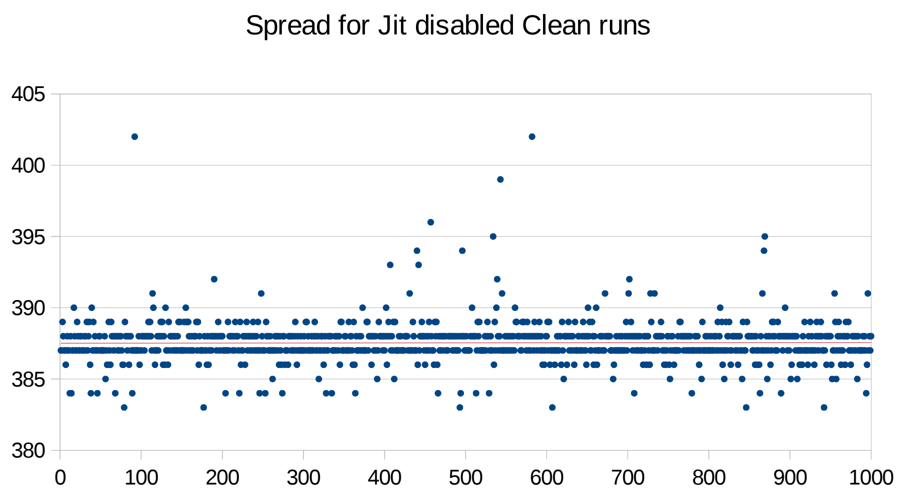

Figure 5.4: Clean Spread of runs.

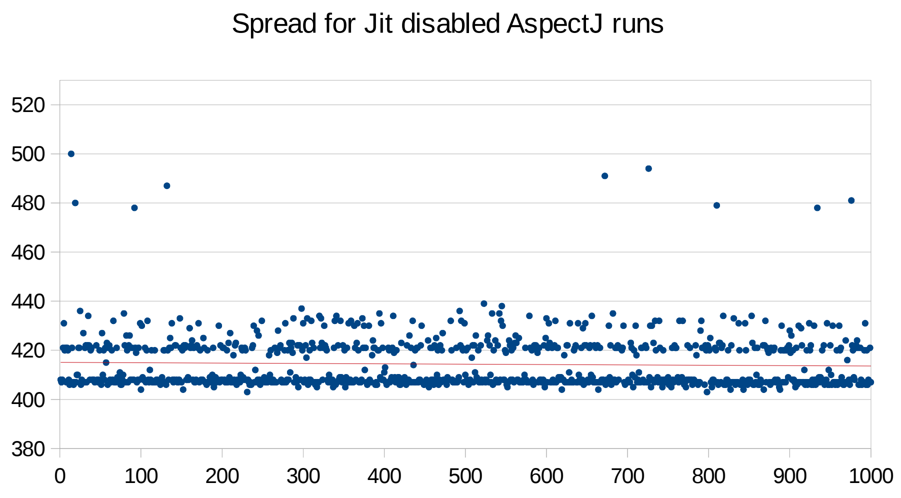

Figure 5.5: AspectJ Spread of runs.

Opcode faults corrected:	501
Value faults corrected:	298
AritOpcode faults corrected:	9
IfOpcode faults corrected:	26
String faults corrected:	88
Map key faults corrected:	65
Total faults corrected:	987
Total runtime:	185502

Figure 5.6: Results from VM test

During the test the VM would occasionally crash. Through analysis of various debugging outputs it was concluded that they are caused by double bit-flips. The high fault-rate is partly to blame for this, but also that we base our injection on each instruction rather than a time window. This and the fact that we use randomly generated numbers means that we will occasionally experience double bit-flips. We have not counted the exact rate, but through observations alone it was deemed to happen in less than 15 of 15000 runs. Even though we make sure to only inject one bit-flip for each instruction, meta-data injection will not be detected immediately as this data is checked only when referenced, with the higher fault rate it is possible for a meta-data element to become injected twice before it is referenced making it unrecoverable for our VM.

5.4 Baseline Failure Tests

As mentioned we also injected faults into our baseline bosVM implementation to show that bit-flips in the expected areas does in fact cause trouble. The most obvious of faults are caused by opcode, aritcode, and ifcode faults as these will cast Java exceptions, and consequently crash the Java VM, the stack traces as well as old and new values of these types of faults are shown in Figure 5.7, Figure 5.8, and Figure 5.9. The stacktraces have been reduced to their last line for easier readability, the content is likely unintelligible for anyone but the authors.

```
opcode
old 21
new 53
java.lang.Exception: Syntax error
at bosVM2.Engine.Frames.MethodFrame.executeFrame(MethodFrame.java:96)
...
```

Figure 5.7: An opcode fault stacktrace with old and new value.

```
aritcode
old 1
new 33
Error detected in Arit Operation
java.lang.Exception: Arit code fault
at bosVM2.Engine.Frames.MethodFrame.doArit(MethodFrame.java:180)
...
```

Figure 5.8: An aritcode fault stacktrace with old and new value.

```
ifcode
old 4
new 36
Error detected in If operation
java.lang.Exception: ifCode fault
at bosVM2.Engine.Frames.MethodFrame.doIf(MethodFrame.java:160)
...
```

Figure 5.9: An ifcode fault stacktrace with old and new value.

The results of injecting a key error is shown in Figure 5.10, where a bit-flip alters the key `bos.sys.langtest` to `bos.óys.langtest`, resulting in the heap attempting to allocate a null value, causing a Java exception to be thrown.

```
key
old bos.sys.langtest
new bos.óys.langtest
java.lang.NullPointerException
at bosVM2.meta.Objects.BosObject.<init>(BosObject.java:12)
at bosVM2.Engine.Heap.Allocate(Heap.java:17)
...
```

Figure 5.10: Map key bit-flip, cause an allocation error on the heap.

The resulting stacktrace caused by en **EmptyStackException** is shown in Figure 5.11, this trace is a little more interesting as it actually has two `pc` bit-flips. This should emphasize the importance of the `pc` fault-tolerance implemented, from the first bit-flip we do not know what the current state of the program actually is. Ultimately a `pop` is attempted on an empty operand stack, causing the exception and crashing the VM. From the stacktrace it can be deduced that the `pc` is set to a `getField` instruction which did not have its associated reference pushed upon the stack beforehand.

```
pc
old pc 76
new pc 108
old pc 0
new pc 4
java.util.EmptyStackException
at java.util.Stack.peek(Stack.java:102)
at java.util.Stack.pop(Stack.java:84)
at bosVM2.Engine.Frames.MethodFrame.doGetField(MethodFrame.java:103)
...
Unknown execution error
```

Figure 5.11: EmptyStackException caused by `pc` bit-flips.

It may not be immediately clear why a value fault caused the resulting stacktrace shown in Figure 5.12, even more so considering that four additional value bit-flips occurred prior to the crash. There have been marked two bit-flips in the trace, as the bit-flips is most likely caused by a reference for a field that has been altered to a different value. The two is likely the same reference that has been flipped twice in between stores and loads, this is not an impossible scenario with the fault rate used. While this is an unlikely scenario, it does show us that a fault happening may not have immediate consequences, as is also evident by the other three bit-flips. From this trace alone we can not tell if they are references or values used in some arithmetic operation, but common for them all is that they will very likely cause the outcome of the method to be erroneous.

```
value
old 0
new 4
old 4
new 5
old 1 <-
new 65 <-
old 9
new 8
old 65 <-
new 193 <-
java.lang.NullPointerException
at bosVM2.Engine.Frames.MethodFrame.doPutField(MethodFrame.java:108)
...
```

Figure 5.12: Value fault stacktrace.

Shown in Figure 5.13 is the stacktrace for a bit-flip in the name of a method in the meta-data. This type of fault is identical and have similar consequences to bit-flips in classes and fields, and for this reason they have been omitted. The method called is `pushint` of the `simpleCalc.calc` class, the bit-flip altered the name to `pushant` which does not exist, leading to a **NullPointerException**.

```
method
Methodname before: pushint
Methodname after: pushant
java.lang.NullPointerException
at bosVM2.meta.Objects.BosMethod.<init>(BosMethod.java:27)
at bosVM2.Engine.Frames.MethodFrame.<init>(MethodFrame.java:30)
at bosVM2.Engine.Frames.MethodFrame.doInvokeVirtual(MethodFrame.java:124)
...
```

Figure 5.13: Method name bit-flip.

To sum up, the outcome of injecting faults into a VM with no defenses, caused pretty much exactly what we expected. A VM unable to perform anything sensible.

Chapter 6

Recapitulation

In this chapter we will comment and evaluate our results and findings as well as discuss the possible reasons of the results including suggestions for improvements. We will also conclude on the project and touch upon possible future work that this project leads to.

6.1 Discussion

The discussed fault-tolerance techniques and their implementation, at least for the Hamming distance and value protection fits well the idea of a VM with relatively low overhead in the presence of no faults. Our major loss of performance was observed with the addition of the Hamming codes to the VM. One of the reasons for this is obviously the nature of the algorithm ,which compared to the job it does, have quite a bit of implementation code. That said, we could possibly reduce the overhead suffered from Hamming codes since we use the encoder rather aggressively in our implementation. We decode a string every time that it is retrieved or accessed, this leads to a number of redundant decoding. An alternative approach would be to keep everything encoded and do comparisons on encoded values and rely on missed map accesses or failed comparisons for initial fault-detection before attempting to decode or error-correct strings.

Regarding the implemented VM and TinyBytecode and the associated semantics and language definitions, the TinyBytecode subset implemented in bosVM does not include exceptions. This is not an oversight but rather a choice of focus, the implemented techniques covers much of the areas of the VM for general use. Exceptions is not strictly needed for core usage of bosVM and TinyBytecode, but could provide an extra dimension regarding masking of faults and defensive programming.

The results of our testing has shown that we have implemented a VM which can in fact catch a number of faults in expected areas. The bosVM does however have considerable run-time overhead, as much as up to 350% with the Java jit compiler enabled and 600% without, from the baseline bosVM. We have already mentioned that we expect to be able to decrease the overhead from Hamming codes. It is also worth considering that these numbers should be compared to running the baseline bosVM in dual or triple redundancy which have an inherent overhead of at least 200% or 300% respectively, and then arbiter implementation is added upon this. We are not trying to argue that 600% is an acceptable run-time, but we have provided a VM solution with build-in error recovery, though not on all components yet. With this done, focus can be moved on to optimizing the solution in regards to run-time. Fault-tolerance is and should be the prime concern. With regards to overhead on the AspectJ enabled bosVM, it is also worth noting that AspectJ does add some noise, we do however not know how much of an performance impact AspectJ has. Regarding AspectJ being used as a fault injection tool, it has come to our attention in the final stages of the project that Hadoop, an open-source software framework for storage and large-scale processing, has developed some kind of AspectJ injection framework [11], the scale of which has not been further studied at this time.

6.2 Conclusions

We have extended the bytecode language TinyBytecode by formalizing semantics for exceptions, as well as behavior of the language under a semantically formalized fault model. A subset of this semantic has been implemented in the VM bosVM, with added fault-tolerance techniques, designed to protect against the faults that we expect to encounter under said fault model. The bosVM has fault detection as well as fault correction and can correct any single bit-flip error in its coverage area, it does however run with a significant runtime overhead. Furthermore we have implemented a fault injection framework using AspectJ targeting the specific fault types of the fault model. AspectJ has also been used to implement logging of corrective measures engaged.

6.3 Future Work

Following is a few suggestions for work that could be done with this project at its base.

Formalizing the last few instructions of TinyBytecode, namely the instructions for arrays should be the first order of business, as well as implementing these in the bosVM and add support for exceptions. The semantics could also be extended to encompass the heap area of the VM. Revisiting the implementation of the bosVM for the purpose of runtime optimization would be a logical next step.

The TinyBytecode language is a very simple language, and we wish to keep it this way. We have considered adding VM specific extensions that would add commonly used mechanics with inherent fault-tolerant design, examples could be monitors, semaphores, lists, maps or anything you might expect from a high level programming language.

Appendix A

Semantics of TinyBytecode

A.1 Imperative Core

The core segment of TinyBytecode includes the instructions for stack manipulation as well as handling variables and values. The instructions for direct stack manipulation are *push*, *pop*, and *dup* which duplicates the top element of the stack, and *swap* which swaps the top two elements of the stack. The instruction *pop* has an optional parameter, which is used to pop multiple variables at a time. The semantic rules for stack manipulation are shown in Figure A.1.

$$\text{push}\frac{m.instruction(pc_i) = \text{push } t\ v}{\begin{array}{c}P, CP \vdash \langle H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, v :: os\rangle : CS\rangle\end{array}}$$

$$\text{pop}_1\frac{m.instruction(pc_i) = \text{pop}}{\begin{array}{c}P, CP \vdash \langle H, \langle m, pc_i, lv, os_h : os_t, \rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, os_t\rangle : CS\rangle\end{array}}$$

$$\text{pop}_2\frac{m.instruction(pc_i) = \text{pop } n}{\begin{array}{c}P, CP \vdash \langle H, \langle m, pc_i, lv, os_1 : \ldots : os_n : os_t, \rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, os_t\rangle : CS\rangle\end{array}}$$

$$\text{dup}\frac{m.instruction(pc_i) = \text{dup}}{\begin{array}{c}P, CP \vdash \langle H, \langle m, pc_i, lv, os_h : os_t, \rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, os_h :: os_h : os_t\rangle : CS\rangle\end{array}}$$

$$\text{swap}\frac{m.instruction(pc_i) = \text{swap}}{\begin{array}{c}P, CP \vdash \langle H, \langle m, pc_i, lv, os_1 : os_2 : os_{3-n}\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, os_2 : os_1 : os_{3-n}\rangle : CS\rangle\end{array}}$$

Figure A.1: Semantic rules for stack manipulation.

Instructions for loading and storing values in local memory are *load* and *store*, the *inc* instruction is included for directly incrementing a stored local variable. Figure A.2 shows the semantic rules for local memory manipulation.

$$\text{load}\ \frac{m.instruction(pc_i) = \text{load}\ t\ v_i}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, lv[v_i] :: os\rangle : CS\rangle \end{array}}$$

$$\text{store}\ \frac{m.instruction(pc_i) = \text{store}\ t\ v_i}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, os_h : os_t\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv[v_i \mapsto os_h], os_t\rangle : CS\rangle \end{array}}$$

$$\text{inc}\ \frac{m.instruction(pc_i) = \text{inc}\ v_i\ v}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv[v_i \mapsto lv[v_i] + v], os\rangle : CS\rangle \end{array}}$$

Figure A.2: Semantic rules for local memory manipulation.

The instructions for manipulating the program counter are *nop*, *goto*, and *if*. The *nop* instruction does nothing but advance the program counter. The *goto* and *if* instructions both change the program counter to a specified value, with *if* only doing so if a conditional check succeeds. This check has the form $v_1\ op\ v_2$, where $op \in \{equals, nequals, grth, lsth\}$. In Figure A.3 the semantic rules for program counter manipulation are shown.

$$\text{nop}\ \frac{m.instruction(pc_i) = \text{nop}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, os\rangle : CS\rangle \end{array}}$$

$$\text{goto}\ \frac{m.instruction(pc_i) = \text{goto}\ pc'}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc', lv, os\rangle : CS\rangle \end{array}}$$

$$\text{if}\ \frac{\begin{array}{c} m.instruction(pc_i) = \text{if}\ op\ \text{goto}\ pc_n \\ exp = v_1\ op\ v_2 \\ op \in \{equals, nequals, grth, lsth\},\ pc' = \begin{cases} pc_n & exp = true \\ pc_{i+1} & exp = false \end{cases} \end{array}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, v_1 : v_2 : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc', lv, os\rangle : CS\rangle \end{array}}$$

Figure A.3: Semantic rules for program counter manipulation.

The instruction used for logical and arithmetic operations is *arit*. This instruction handles both unary and binary operations. Unary operations take the form $op\ v$ where $op \in \{neg\}$. Binary operations take the form $v_1\ op\ v_2$ where $op \in \{add, sub, mul, div, rem, shl, shr, ushr, and, or, xor\}$. Figure A.4 shows the semantic rules for logical and arithmetic operations.

$$\text{arit}_{unary}\frac{\begin{array}{c} m.instruction(pc_i) = \text{arit } op \\ op \in \{neg\} \\ res = op\ v \end{array}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, v : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, res :: os\rangle : CS\rangle \end{array}}$$

$$\text{arit}_{binary}\frac{\begin{array}{c} m.instruction(pc_i) = \text{arit } op \\ op \in \{add, sub, mul, div, rem, shl, shr, ushr, and, or, xor\} \\ op \in \{div, rem\} \Rightarrow v_2 \neq 0 \\ res = v_1\ op\ v_2 \end{array}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, v_1 : v_2 : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, res :: os\rangle : CS\rangle \end{array}}$$

Figure A.4: Semantic rules for logical and arithmetic operations.

A.1.1 Objects

In this section the semantic rules concerning objects will be presented. The first rule is *new* which is used for instantiation of objects. As a helper function *alloc* is introduced which takes care of allocating a new heap location for the object. This function takes as parameters the class to be instantianted, as well as the heap, and returns a reference to the instantiated object, as well as a modified heap that contains this object. The *new* instruction takes a class as a parameter and it is required that instantiated object is an instance of that class. It is also required that the heap location found is not already in use and that the reference returned by *alloc* is mapped to the instantiated object on the heap. The use of the *new* instruction results in the reference to the object being pushed on to the operand stack, and the heap being modified to include the new object.

Another aspect of the object semantics is fields. Objects contain fields and they store values on the heap in the respective objects. To fetch a field the $getfield$ instruction is used and to modify a field the $putfield$ instruction is used. The use of $getfield$ requires a reference to an object as the top element on the operand stack. The reference is popped, the object reference is resolved and the specified field value is fetched and pushed on the operand stack. The use of *putfield* requires a reference followed by the new value to be the top elements of the operand stack. The reference and the value are popped from the operand stack and the object reference resolved. The specified field is then mapped to the new value on the heap.

The semantic rules concerning objects are shown in Figure A.5.

$$\text{new}\ \frac{\begin{array}{c} m.instruction(pc_i) = \text{new } c \\ (ref, H') = alloc(c, H) \quad obj = H'[ref] \quad ref \notin dom(H) \\ c \in P.classes \quad obj.class = c \end{array}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, os\rangle : CS\rangle \Rightarrow \\ \langle H', \langle m, pc_{i+1}, lv, ref :: os\rangle : CS\rangle \end{array}}$$

$$\text{getfield}\ \frac{\begin{array}{c} m.instruction(pc_i) = \text{getfield } field \\ obj = H[ref] \quad val = obj.fieldVal(field) \end{array}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, ref : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, val :: os\rangle : CS\rangle \end{array}}$$

$$\text{putfield}\ \frac{\begin{array}{c} m.instruction(pc_i) = \text{putfield } field \\ obj = H[ref] \quad obj[field \mapsto val] \end{array}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, ref : val : os\rangle : CS\rangle \Rightarrow \\ \langle H[ref \mapsto obj], \langle m, pc_{i+1}, lv, os\rangle : CS\rangle \end{array}}$$

Figure A.5: The semantic rules concerning objects.

A.1.2 Methods

In this section the semantic rules concerning method invocation and return will be presented. As mentioned earlier a choice was made to leave out static methods, static fields and interfaces. For this reason the only instruction concerning method invocation is the *invokevirtual* instruction. As discussed in Section 3.1.4 the actual method invoked is resolved at runtime. To resolve this method a function called *lookup* is used. This function is based on a procedure with similar functionality in the JVM specification [15]. This function takes the method identification and a class as parameters. The method identification in this case is considered to be the method's name and parameter types. First a check is made whether or not the provided class contains an implementation of that method. If it does, then that particular method is returned, but if it does not, then a recursive call is made on the class' superclass with the same

```
lookup(methodId, class):
        if(class contains methodId)
                return method
        elseif(class has superclass)
                return lookup(methodId, superclass)
        else
                fail
```

Listing A.1: Pseudocode for the *lookup* function.

method identification. If no method has been found and the current class does not have a superclass, then the function will fail. By performing this lookup the most specialized version of the method will be returned as desired. The pseudocode for the function *lookup* is shown in Listing A.1.

To be able to use the *invokevirtual* semantic rule some conditions have to be fulfilled. On the stack it is required to have a reference to the object where the method is to be invoked followed by the parameters for that method. It is also required that the reference on the stack is not null. The number of parameters for the method is obtained by using the function *numArgs*, introduced in Section 3.1.5, on the method returned by *lookup*. When this semantic rule is used, a new frame is created and pushed onto the call stack. The reference and parameters are popped from the stack of the calling frame and placed into local variables of the newly created frame. The method of the new frame is set according to the method returned by *lookup*. The program counter is set to 0 as by convention and the operand stack will be empty.

For return an optional parameter was specified in the syntax which is reflected in the semantic rules. In total there are four different semantic rules for return. Two rules to handle when returning without a specified parameter, also known as void, and two rules for handling when returning with a parameter. The reason for having two rules for each is to handle the special case when returning from the last frame on the call stack. This case needs to be handled differently since the program is to be terminated when this case occurs. When returning without a parameter and the call stack is empty, the frame configuration is changed to *Terminate*. If the call stack contains more than one frame then the current frame is popped and control is transferred to the next frame on the call stack.

In the cases where the optional type parameter has been specified, a return value is expected on top of the operand stack. If there are more than one frame on the call stack when returning, the return value is then popped and pushed on top of the operand stack of the next frame on the call stack. The current frame is then popped off the call stack and control is transferred again. In the case where the operand stack only contains one frame, then the return value is popped and the frame configuration is again changed to $Terminate$. The semantic rules concerning method invocation and return is shown in Figure A.6.

$$\text{invokevirtual}\,\frac{\begin{array}{c} m.instruction(pc_i) = \text{invokevirtual } m_0 \\ lv' = ref : v_1 : \cdots : v_n \quad ref \neq null \quad obj = H[ref] \\ m' = lookup(m_0, obj.class) \quad n = m'.numArgs() \end{array}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, ref : v_1 : \cdots : v_n : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m', 0, lv', \epsilon\rangle :: \langle m, pc_i, lv, os\rangle : CS\rangle \end{array}}$$

$$\text{return}_1\,\frac{m.instruction(pc_i) = \text{return}}{P, CP \vdash \langle H, \langle m, pc_i, lv, os\rangle : \epsilon\rangle \Rightarrow \langle H, \langle Terminate\rangle\rangle}$$

$$\text{return}_2\,\frac{m.instruction(pc_i) = \text{return}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, os\rangle : \langle m', pc', lv', os'\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m', pc', lv', os'\rangle : CS\rangle \end{array}}$$

$$\text{return}_3\,\frac{m.instruction(pc_i) = \text{return } t}{P, CP \vdash \langle H, \langle m, pc_i, lv, v : os\rangle : \epsilon\rangle \Rightarrow \langle H, \langle Terminate\rangle\rangle}$$

$$\text{return}_4\,\frac{m.instruction(pc_i) = \text{return } t}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, v : os\rangle : \langle m', pc', lv', os'\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m', pc', lv', v :: os'\rangle : CS\rangle \end{array}}$$

Figure A.6: The semantic rules concerning method invocation and return

A.1.3 Arrays

In TinyBytecode support for arrays is limited to creating a new array objects, accessing array elements, and returning the lengths of arrays. The instruction *anew* creates a new array with a given length and allocates space on the heap, a reference to the array is pushed on the stack upon completion. To aid in this the function *allocA* is used. This function is similar to the *alloc* function, and takes the size of the array, its type, and the heap as parameters, and returns a reference to the allocated array, as well as a modified heap containing the array. The instruction *aload* is used to load a value from an array at a given index, while the instruction *astore* is used to store a given value at a given index in an array. Finally, the instruction *alength* pushes the length of a given array onto the stack. The semantic rules for arrays are shown in Figure A.7.

$$\text{anew}\,\frac{\begin{array}{c} m.instruction(pc_i) = \text{anew } t \\ (aref, H') = allocA(size, t, H) \\ a = H'[aref] \qquad size \geq 0 \\ a.type = t \qquad a.length = size \end{array}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, size : os\rangle : CS\rangle \Rightarrow \\ \langle H', \langle m, pc_{i+1}, lc, aref : os\rangle :: CS\rangle \end{array}}$$

$$\text{alength}\,\frac{\begin{array}{c} m.instruction(pc_i) = \text{alength} \\ len = H[aref].length \end{array}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, aref : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, len :: os\rangle : CS\rangle \end{array}}$$

$$\text{aload}\,\frac{\begin{array}{c} m.instruction(pc_i) = \text{aload } t \\ elem = H[aref].get(index) \end{array}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, aref : index : os\rangle : CS\rangle \Rightarrow \\ \langle H, \langle m, pc_{i+1}, lv, elem :: os\rangle : CS\rangle \end{array}}$$

$$\text{astore}\,\frac{\begin{array}{c} m.instruction(pc_i) = \text{astore } t \\ a = H[aref] \qquad a' = a[get \mapsto a.get[index \mapsto val]] \\ H' = H[aref \mapsto a'] \end{array}}{\begin{array}{c} P, CP \vdash \langle H, \langle m, pc_i, lv, aref : index : val : os\rangle : CS\rangle \Rightarrow \\ \langle H', \langle m, pc_{i+1}, lv, os\rangle :: CS\rangle \end{array}}$$

Figure A.7: Semantic rules for arrays

Appendix B

The bosVM Implementation

This appendix will contain some general documentaion regarding the bosVM implementation code.

Full source

The full source code of bosVM can be checked out from the repository

https://code.google.com/p/sfmaster/

Hamming Code Implementation

The Hamming Codes encoding is conducted by retrieving the bit pattern of the input, calculating the number of parity bits needed, creating the message, and calculating and writing the parity bit information before returning as an UTF-16 string. The source for the primary encoding function is shown in Listing B.1

```java
int byteLength = 16;
String encoding = "UTF-16";
//Hamming codes encoding of a string
public String enCode(String s) {
    char[] block = getBitPattern(s,false);
    //retrieve the bit pattern
    int n = block.length;

    //calculate the needed amount of parity bits
    int parityBits = calculateEncodingParityBits(n);
    //create message
    char[] message = new char[n+parityBits];

    int nextPower = 4 ,powerIndex = 2;
    int blocki = 0;
    for(int i = 0; i < message.length; i++) {
        if(i+1 == nextPower) {
            message[i] = '0';
            powerIndex++;
            nextPower = (int) Math.pow(2, powerIndex);
        } else if ( i < 2 ) {
            message[i] = '0';
        }
        else{
            message[i] = block[blocki];
            blocki++;
        }
    }

    //calculate each parity bit
    for(int bit = 0; bit < parityBits; bit++) {
        int skip = (int) Math.pow(2, bit);
        int val = 0;
        for(int i = skip-1; i < message.length; i = i + skip) {
            for(int j = 0; j < skip && i < message.length; j++) {
                val = val + Character.getNumericValue(message[i]);
                i++;
            }
        }
        //Write the parity bit
        if(val % 2 == 0) {
            message[skip-1] = '0';
        } else { message[skip-1] = '1';}
    }
    return toASCII(message);
}
```

Listing B.1: Hamming Codes encoding function

The **getBitPattern(String,boolean)** retrieves the bit pattern using the **Integer.toBinaryString(string)** function, for calculation purposes the individual chars are padded with 0s to match the byte length of 16. This is necessary as we can only retrieve a bit pattern to the most significant 1 bit, which would make decoding impossible as 0 bits after the most significant 1 bit could be parity bits, but we would have no way of telling where in the string it should be without storing redundant information, the source code is shown in Listing B.2. Notice that we return the bit pattern in reverse order, the order is reversed again when storing, this serves the purpose preserving 0 bits in the last byte of the encoded message. The second part of the **getBitPattern(string,boolean)** function, after **else**, on line 14 bit pattern retrieval of encoded strings.

```java
private char[] getBitPattern(String s, boolean encoded) {
    String temp = "";
    //Get byte array of the message, convert to bit string
    if(!encoded) {
        for(char b: s.toCharArray()) {
            String bin = Integer.toBinaryString(b);
            if(bin.length() < byteLength) {
                temp = temp + String.format("\%0"+(byteLength-bin.length()+"d"
                    +"\%s"), 0, bin); }
            else {
                temp = temp + bin;
            }
        }
    }
    else {
        boolean first = true;
        for(char b: s.toCharArray()) {
            String bin = Integer.toBinaryString(b);
            if(first)
            { temp = temp + Integer.toBinaryString(b); first = false; }
            else {
                if(bin.length() < byteLength) {
                    temp = temp + String.format("\%0"+(byteLength-bin.length()
                        +"d"+"\%s"), 0, bin); }
                else {
                    temp = temp + bin;
                }
            }
        }
    }
    return reverseOrder(temp.toCharArray());
}
```

Listing B.2: **getBitPattern(string,boolean)** source code

Calculating the parity bits needed is done with a call to the **calculateEncodingParityBits(int)** function which takes the message length as a parameter, and returns the number of parity bit needed to reach the block length, the source code is in Listing B.3

The calculation of parity bit values is already shown in Listing B.1, at the very end of this listing the return line is found, reading **return toASCII(message)**. This functions, as implied by the name, convert the bit pattern back to a Java string, despite the name of the function of the name, the string is in UTF-16 encoding, and not ASCII. This is simply a leftover name from an early revision of the algorithm implementation.

Due to the nature of Hamming Codes the decoding algorithm is almost

```java
public int calculateEncodingParityBits(int length) {
    int parityBits = 0;
    while((Math.pow(2, parityBits)-parityBits-1) < length) {
        parityBits++;
    }
    return parityBits;
}
```

Listing B.3: Function which calculates number of needed parity bits for encoding

```java
public int calculateDecodingParityBits(int length) {
    int parityBits = 0;
    while((Math.pow(2, parityBits)-1) < length) {
        parityBits++;
    }
    return parityBits;
}
```

Listing B.4: Functions which calculates number of parity bit for decoding

identical to encoding, and varies only in the details as evident by comparing Listing B.3 to Listing B.4, which shows the function for calculating the number of parity bits in an encoded message. The differences in the code is comparable in magnitude to this. Where the decoding function is a different is that it corrects errors in the returned string, the source for the decoding functions is shown in Listing B.5

```java
public Object[] deCode(String s) {
    boolean error = false;
    //retrieve the bit pattern of the input string
    char[] message = getBitPattern(s, true);

    //Calculating the number of parity bits needed for the string
    int parityBits = calculateDecodingParityBits(message.length);

    //error detection and correction
    //Calculation of parity bit values
    int pv = calculateParityValues(parityBits,message);

    //If a fault was found, correct it
    if(pv > 0) {
        error = true;
        message = correctBitFlip(pv-1,message);
    }
    //Remove parity bits and build output string
    message = removeParityBits(parityBits, message);

    Object[] out;
    if(!error) {
    out = new Object[]{error, toASCII(message)};
    } else { out = new Object[]{error, toASCII(message),s}; }
    return out;
}
```

Listing B.5: Hamming codes decoding function

When decoding the parity value is calculated which tells if any bit is in error, the value will then be an integer larger than 0 corresponding to the bit in error. The parity value is passed on to the **correctBitFlip(int,char[])** function which takes the parity value and the string as input, returning the corrected string, source code is shown in Listing B.6

```java
//Corrects a single bit flip in a char array.
public char[] correctBitFlip(int pv,char[] message)
{
    if(message[pv] == '1') {
        message[pv] = '0';
    } else { message[pv] = '1'; }
    return message;
}
```

Listing B.6: The **correctBitFlip(int,string)** function

Appendix C

Resumé

This project covers the process of implementing a virtual machine that can resist faults caused by single event upsets. Single event upsets are relevant because the project is based on a case with a satellite. Satellites' deployment environment have a higher rate of single event upsets than at sea level, so a program running on a satellite must be hardened to not run the risk of crashing or get silent data corruptions. To get an overview of the distribution of fault-tolerant components an investigation has been made featuring a collection of micro-controllers to see what kind of fault-tolerance they include if any at all. Related work has also been explored to see what solutions already exists. The VM is based on TinyBytecode which semantics have been expanded to include exceptions and a suiting fault model. The VM implemented based on the semantics features fault-tolerance through encoding, using AN-codes and Hamming codes, as well as duplication of values for recovery in case of a fault. For testing purposes a testing framework has been implemented using AspectJ to keep the testing code separated from the VM code. The additional overhead for runtime has been estimated, through testing, to 6 times the base runtime. This massive overhead is partly because of an aggressive encoding politic. The ability to correct single event upsets is also tested with great success, the VM is able to handle way above a realistic fault rate.

Bibliography

[1] AspectJ, crosscutting objects for better modularity. `http://eclipse.org/aspectj/`. Last visited on 2014-05-29.

[2] 2002-2003 Palo Alto Research Center 1998-2001 Xerox Corporation. The AspectJTM programming guide. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`. Last visited on 2014-05-29.

[3] AAUSAT3. 060524 err abnormal beacon. `http://www.space.aau.dk/aausat3/index.php?n=Main.Log130601`, May 2013. Last Visited on 2013-11-25.

[4] ARM. Arm infocenter. `http://infocenter.arm.com/help/index.jsp`. Last visited on 2014-02-20.

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004.

[6] R. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, page 121, 2002.

[7] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. International Computer Science Series. Addison-Wesley, fourth edition, 2009.

[8] Emil Custic, Martin Fjordvald, Martin Sørensen, Rune Hansen, and Sebastian Lybæk. Fault-tolerance: An exploration of software-based solutions. part 1 of a master thesis, 2014.

[9] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 385–396, New York, NY, USA, 2010. ACM.

[10] GomSpace. Arm infocenter. `http://gomspace.com/index.php?p=products-a712c`. Last visited on 2014-05-29.

[11] Hadoop. Hadoop fault injection framework and development guide. `http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/FaultInjectFramework.html`.

[12] René Rydhof Hansen. *Flow Logic for Language-Based Safety and Security.* PhD thesis, Technical University of Denmark, 2005.

[13] Hans Hüttel. *Transitions and Trees.* Cambridge University Press, 2010.

[14] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java virtual machine instruction set. `http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5`, February 2013. Last Visited on 2013-11-26.

[15] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java® virtual machine specification. `http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`, February 2013. Last Visited on 2013-11-26.

[16] T.C. May and Murray H. Woods. Alpha-particle-induced soft errors in dynamic memories. *Electron Devices, IEEE Transactions on*, 26(1):2–9, Jan 1979.

[17] Bill McDaniel. An algorithm for error correcting cyclic redundance checks, June 2003.

[18] Jeff Napper, Lorenzo Alvisi, and Harrick Vin. A fault-tolerant java virtual machine. In *In Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003), DCC Symposium*, pages 425–434, 2002.

[19] E. Normand. Single event upset at ground level. *Nuclear Science, IEEE Transactions on*, 43(6):2742–2750, Dec 1996.

[20] W.W. Peterson and D.T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, Jan 1961.

[21] George A. Reis, Jonathan Chang, and David I. August. Automatic instruction-level software-only recovery. *IEEE Micro*, 27(1):36–47, 2007.

[22] 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 '05, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[23] Isabella Stilkerich, Michael Strotz, Christoph Erhardt, Martin Hoffmann, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. A jvm for soft-error-prone embedded systems. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '13, pages 21–32, New York, NY, USA, 2013. ACM.

[24] Gary M. Swift and Steven M. Guertin. In-flight observations of multiple-bit upset in DRAMs. *IEEE Transactions on Nuclear Science*, 47(6):2386–2391, December 2000.

[25] Near R. Wagner. chapter Chapter 6. The Hamming Code for Error Correcetion. Published online by the Author., 2003.

[26] N.J. Wang and S.J. Patel. Restore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):188–201, July 2006.