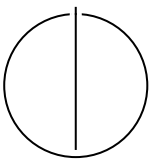
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Analysis and Validation of Semantic Mistranslation Errors in Emulators

Christian Krinitsin



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Analysis and Validation of Semantic Mistranslation Errors in Emulators

Analyse und Validierung Semantischer Übersetzungsfehler in Emulatoren

Author:	Christian Krinitsin
Examiner:	Prof. Pramod Bhatotia
Supervisor:	Sebastian Reimers, Theofilos Augoustis
Submission Date:	12.09.2025

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 12.09.2025 Christian Krinitsin

Acknowledgments

My sincere thanks to Professor Bhatotia for providing the opportunity to explore this fascinating topic, as well as Theo and Sebastian, for their exceptional mentorship, patience, and insightful feedback. To Wladi and Elena, thank you for your encouragement, for the much-needed distractions, and for always being there.

Abstract

CPU emulators are used to execute programs across different Instruction Set Architectures. They are inherently complex by having to support multiple architectures in different combinations and are thus prone to errors. We conduct a bug study over multiple user-mode emulators to determine the different types of bugs affecting emulators in order to decide if mistranslation errors are a significant cause of errors. Based on our bug study, we extend the symbolic execution backend of the translation validator Focaccia, a software system used to find and reproduce mistranslation errors in user-mode emulators. Therefore, using the bug study, we will be able to reproduce new errors that weren't supported by Focaccia before.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**

2 Background **3**
2.1 CPU Emulators . . . 3
2.2 Large Language Models . . . 4

3 Overview **7**
3.1 Scraping bugs . . . 7
3.2 Classifying reports . . . 8
3.3 Focaccia . . . 8

4 Design **11**
4.1 Gathering the dataset . . . 11
4.1.1 GitLab and GitHub . . . 11
4.1.2 Mailing List . . . 11
4.1.3 Launchpad . . . 12
4.2 Classification using LLMs . . . 12
4.2.1 Zero-Shot Classification . . . 12
4.2.2 Ollama . . . 13
4.3 Expanding Focaccia . . . 14

5 Implementation **16**
5.1 Identifier . . . 16
5.2 Creating the Prompt . . . 16
5.3 Miasm . . . 17

6 Evaluation **20**
6.1 Plan . . . 20
6.2 QEMU . . . 21
6.2.1 Manual Classification . . . 21

6.2.2 BART-large . . . 21
6.2.3 Qwen3 . . . 22
6.2.4 DeepSeek-R1 . . . 22
6.2.5 Gemma3 . . . 24
6.2.6 Summary . . . 26
6.3 Box64 . . . 26
6.4 FEX . . . 27
6.5 Focaccia . . . 28

7 Related Work **30**

8 Conclusion **32**

9 Future Work **33**

Abbreviations **34**

List of Figures **35**

List of Tables **37**

Bibliography **38**

1 Introduction

The landscape of Instruction Set Architectures (ISAs) is undergoing a significant change. While x86-64 has long dominated general-purpose computing, modern systems increasingly tend to alternatives such as Arm and RISC-V. These alternatives are often open-source and more energy efficient, which makes them more appropriate for specific tasks, i.e., servers.

However, to ensure proper usability and stability of new ISAs, support for existing software is needed. While it is possible for a lot of software projects to recompile for another ISA, we can find a lot of widely used legacy software, where the source code is not available anymore and therefore cannot be recompiled.

One approach for ensuring compatibility is emulation, which supports running software built for a different hardware configuration by translating guest instructions. But CPU emulators are fundamentally hard to implement, making them prone to mistranslation errors.

One proposed solution is translation validation, which is the approach taken by Focaccia [TUM]. This system finds and reproduces semantic mistranslation errors in emulators. This is achieved using symbolic execution, which is a technique to execute programs using symbolic values and track their changes throughout the execution. Focaccia compares the program semantics, extracted as state mutations using symbolic execution.

If a mismatch - thus a mistranslation - occurs, an error will be thrown and a minimal reproducer program will be returned. Such a reproducer program reconstructs the state prior to the mismatch. This approach relies on the completeness of Miasm, the symbolic execution backend of Focaccia, which makes it infeasible for unsupported instructions.

We will collect mistranslation bugs by analyzing bug reports of the different user-mode emulators: QEMU, Box64 and FEX. The data will be used to reproduce new bugs using Focaccia and to expand Miasm if needed. As the gathered dataset will be too large to deal with manually, we will use Large Language Models (LLMs) to automate this process. To determine which LLM best fits our needs, we will benchmark different models and compare them to a manual classification. Afterwards, we will make use of already existing mistranslation reports for CPU emulators, in order to expand Miasm.

We begin by looking at the background regarding CPU emulators and LLMs. Then

we share an overview, the design, implementation and evaluation of our two main contributions:

1. A **bug study** for existing user-mode CPU emulators using LLMs. This will give an overview of the occurring error types and the significance of mistranslation errors.

2. **Reproducing** mistranslation errors of user-mode emulators using Focaccia. For bugs that are not supported by Miasm, we will extend the backend to support the instruction in question.

In the end, we will discuss related work, present a conclusion and an outlook on future work.

The source code for Miasm and the bug study database are available at `https://github.com/taugoust/miasm` and `https://github.com/ckrinitsin/emulator-bug-study` respectively.

2 Background

2.1 CPU Emulators

CPU emulators are systems that execute programs implemented for a possibly different ISA. They need to support the guest ISA, by providing support for instructions and registers and reproducing their semantics on the target ISA [Mar+09; SN05].

CPU emulators are often split into two types:

- An **interpreter** emulates one instruction at a time in a loop. While the implementation is rather simple, the performance is extremely low due to a high overhead.
- A **binary translator** translates blocks of guest instructions into semantically equivalent host instructions as they are discovered. The resulting code is then written into a cache and executed. This approach is more complex in the implementation, but has also a significantly higher performance compared to the first one.

Common emulators use dynamic binary translation to emulate programs. The process is described in Figure 2.1: A **lifter** translates a sequence of instructions, here called Translation Block (TB), into a platform-independent Intermediate Representation (IR). Then, the IR gets **optimized** by applying optimizations like constant propagation or dead code elimination. A **code generator** translates the optimized IR into the host ISA, which gets cached and executed afterwards. The benefit of this approach is the possibility of using different ISA combinations with the same optimizer [SN05].

This approach is very complex. An emulator must support lifters and code generators for multiple ISAs, such that they work in different combinations and are compatible with the IR. Additionally, the optimizer has to make the IR more performant, while ensuring the stability of its semantics [SN05]. Therefore, dynamic binary translators are prone to mistranslation errors.

Finally, we distinguish between user-mode and system-mode emulation. On one hand, we have system-mode emulators that emulate an entire system, including an OS. On the other hand, we have user-mode emulators that support only unprivileged programs. In this work, we will focus on the second kind.

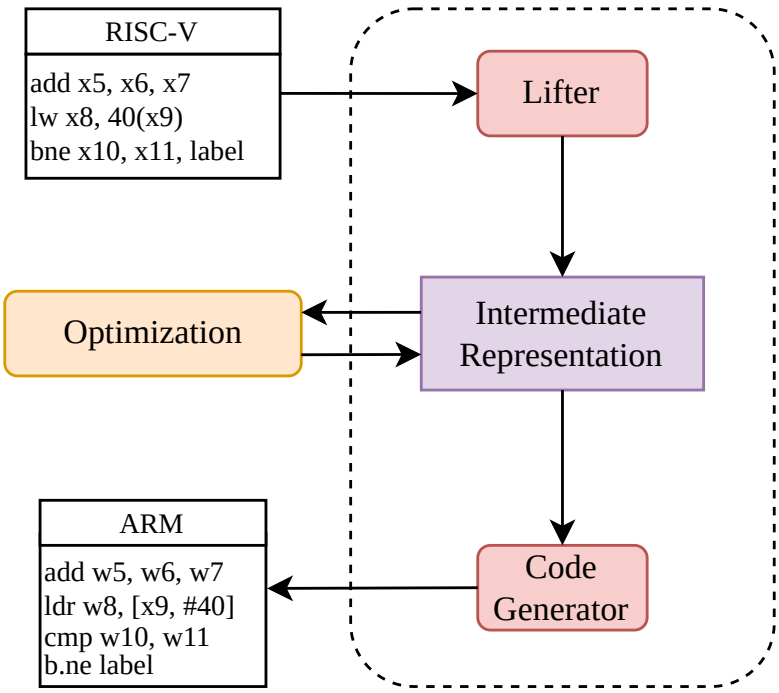


Figure 2.1: Overview of the binary translation approach on the example of a RISC-V to ARM translation. *A sequence of instructions in a guest ISA gets lifted to an Intermediate Representation. The IR gets optimized and the code for a host ISA is generated.*

2.2 Large Language Models

An LLM is a system designed to understand and generate human language. In a learning process, the model gets trained on a large amount of data, see Figure 2.2. The training data gets tokenized and the probabilities for a specific token, based on previously occurring ones, are learned. In a fine-tuning process, the model gets trained based on human feedback. The end result is a model ready to be prompted by a user.

The introduction of transformers was a breakthrough in the world of LLMs [Vas+23]. This architecture proposed the conversion of text into so-called tokens, which are embedded in a context window. Additionally, the use of tokens allows for highly efficient parallelization. Furthermore, transformers allow for a greater understanding of the context by maintaining order and cohesion of the tokens. This architecture is used today in every relevant LLM, such as DeepSeek [deeb] or Chat-GPT [Bro+20].

LLMs are very versatile; they can generate code, text, images, audio and videos. They can also semantically understand texts and assignments, making it useful to

categorize inputs. Since the addition of a reasoning mode, LLMs are able to reason with themselves about a problem, exposing their 'thought process' to the user. This enhances the transparency and the quality of a given answer.

However, there are many downsides one needs to discuss when talking about LLMs. One aspect is the process of gathering the training data [Yan+25]. It became clear that LLMs are partly trained on unethically claimed data, making it a huge privacy concern. Furthermore, the quality of results given by LLMs can vary. Especially vulnerability concerns for coding are coming up over time [Yao+24]. It is always advisable to approach LLM results with caution and cross-check them.

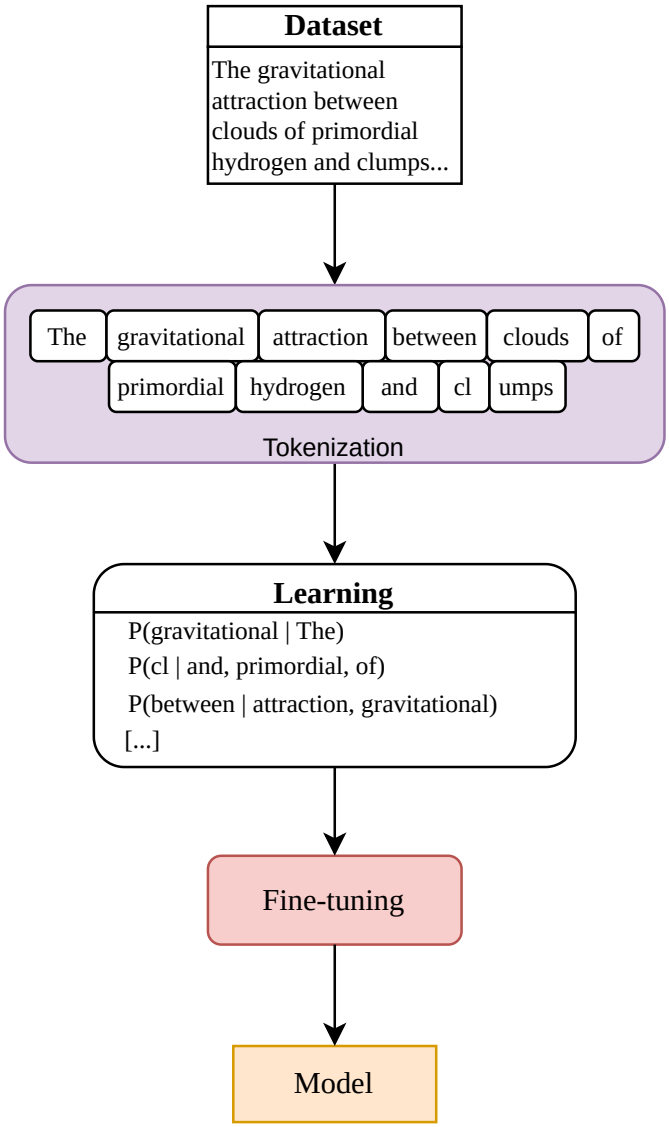


Figure 2.2: Overview of the training process of an LLM. *The dataset gets tokenized and the sequence of tokens learned by creating probabilities for a token prediction. Afterwards, the model gets fine-tuned manually, which results in a ready-to-use LLM.*

3 Overview

To answer the question of the significance of mistranslation bugs in user-mode emulators, we will do a bug study for three different emulators:

- QEMU: a general-purpose emulator that has been around for 20 years and supports emulation between all different kinds of architectures.
- Box64: a smaller user-mode emulator. The purpose of this emulator is to run x86-64 video games on Arm and RISC-V devices.
- FEX: a Work-in-Progress user-mode emulator, which aims for the same goal and use case as Box64.

Using these three emulators, we aim to determine whether mistranslations are a significant cause of errors.

Our approach consists of three parts: scraping all bugs, using an LLM to filter out non-mistranslation bugs and expanding the symbolic execution backend of our validator Focaccia.

3.1 Scraping bugs

QEMU. We have three sources for bug reports concerning QEMU: the developer mailing list [QEMc], the QEMU GitLab issues page [QEMa] and the Launchpad bug tracker [QEMb]. End-user bug reports had to be created in Launchpad until the year 2020. The QEMU maintainers suggested a complete migration to GitLab, including the bug tracker [Haj]. Since then, it is necessary to use the GitLab issues page to submit a bug report. The mailing list is the main communication tool for developers. While it is mainly used to discuss certain code aspects, there can also be found some bug reports. It should be noted that the Launchpad bug tracker was synchronized with the mailing list and that we can only access a Launchpad report through the dedicated e-mail thread.

By going through all three sources, we can collect a complete report list of QEMU. Due to time constraints, we will cover all reports from a period of ten years (April 2015 to May 2025).

Box64 and FEX. Box64 and FEX collect their bug reports via their respective GitHub issues pages [FEX; pti]. By implementing a generic scraper, which collects all issues from any repository on GitHub, we can easily create a report list for each of the two emulators.

3.2 Classifying reports

Now we will structure the report lists by classifying each report. We decided to use LLMs for that, as the dataset is too large to deal with manually. Our focus will be on deciding automatically whether the bug is a mistranslation between two architectures or not.

Since we focus on user-mode emulator bug reports, we need to filter out all reports that do not concern us. These include documentation-related issues, feature requests, retracted reports, and, in the case of QEMU, system-mode bug reports. We reduce our initial set of reports using an LLM and a manual review of the classification afterwards.

We have to decide which LLM to use. For that, we will do a manual classification of all QEMU reports and compare multiple models based on the accuracy of their classifications. After that, we will be able to do more automated classifications of the reports of the other emulators, so that we can estimate a lower bound for each category.

3.3 Focaccia

After having gathered all reported mistranslation bugs of the emulators, we will try to make every bug supported by the validator Focaccia. Figure 3.2 illustrates an overview of Focaccia and of it's approach. A test program gets executed by the emulator and by the symbolic execution backend. Focaccia uses the software Miasm as its backend. Miasm creates symbolic equations that track changes of memory and registers and sets up an expected state. This state gets compared with the actual state of the emulator we are testing. If the states are not equal, Focaccia throws an error and creates a minimal reproducer program.

While Focaccia is general across all kinds of programs, the symbolic execution backend does not include every instruction. Therefore, we aim to expand Miasm to enable support for as many instructions as possible. We do that by going through our bug list step-by-step and trying to reproduce the bugs that are not reproducible yet.

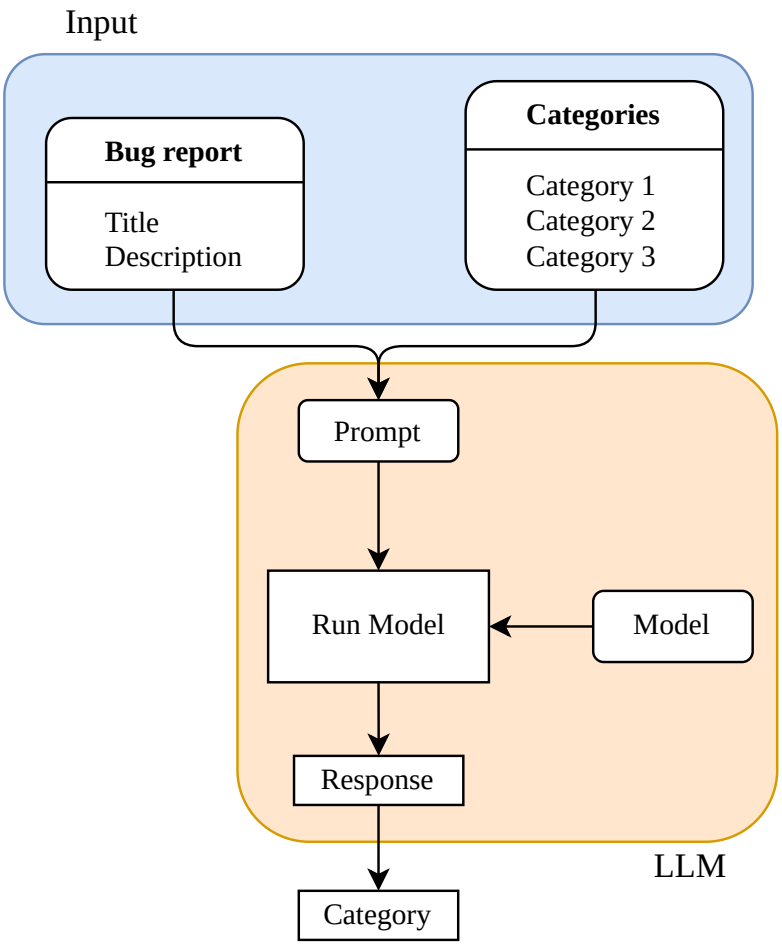


Figure 3.1: Overview of the classifier. *A bug report and fixed categories (along with explanations) get formed into a prompt that is presented to an LLM. The output has to be parsed so that the category can be extracted.*

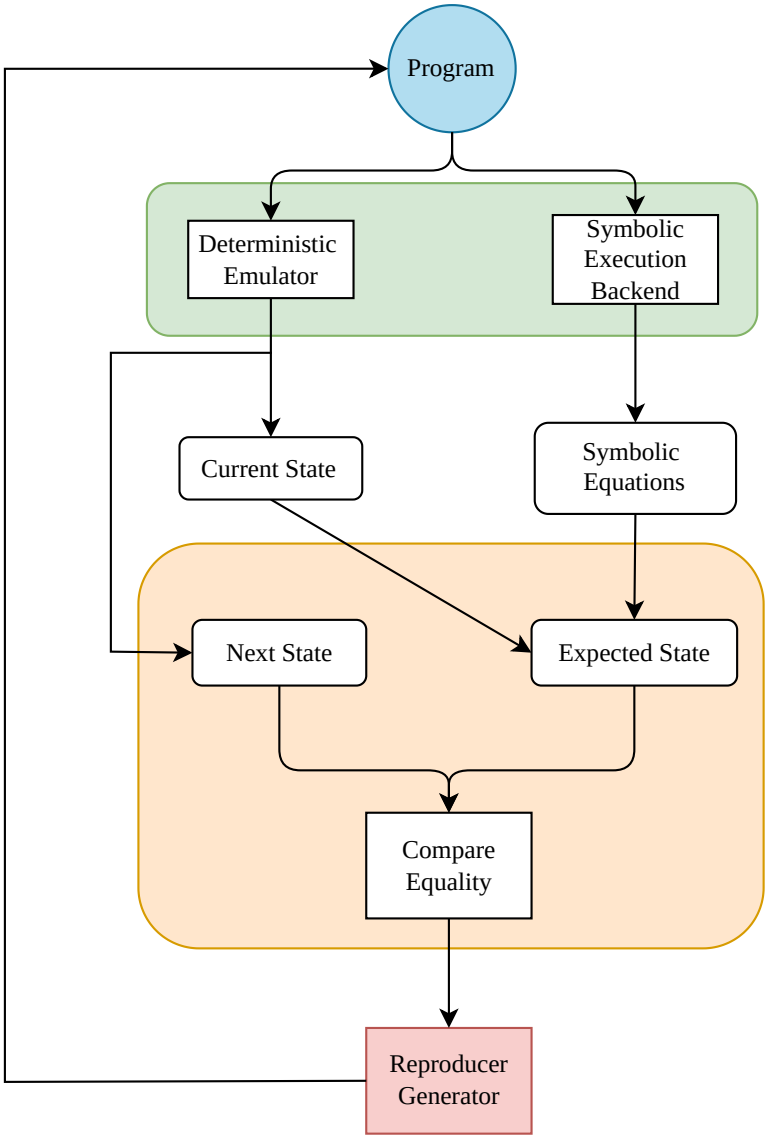


Figure 3.2: Overview of Focaccia. *A program gets executed by a symbolic execution backend and by the emulator. The backend creates symbolic equations and an expected next state, using the equations and the current state of the emulator. The next state of the emulator is then compared to the expected state. If a mismatch occurs, a minimal reproducer program is generated.*

4 Design

4.1 Gathering the dataset

In the following, we will describe the procedure of creating the dataset, using various sources for different user-mode emulators.

4.1.1 GitLab and GitHub

The process of collecting all reports is very similar for GitLab and GitHub. We will use their respective APIs [Gitb; Gita] to retrieve all reports along with metadata. Metadata includes:

- An identifier for the report.
- The creation date of the report.
- The state of the issue.
- Issuer defined labels.
- A Uniform Resource Locator (URL) to the website of the report.

Responses are given in pages with 100 items each, so we have to initiate multiple requests.

The scraping of the GitLab/GitHub issues results in two output formats: Tom's Obvious, Minimal Language (TOML) and text. The TOML format includes important metadata and the text format is included, because it is easier to feed into an LLM, as it resembles natural language.

4.1.2 Mailing List

The archive website of the mailing list contains an indexed version of all e-mail threads regarding QEMU. The conversion of the mailbox format to Hypertext Markup Language (HTML) is done automatically by the software Mhonarc [sym]. This information is important for parsing the mailing list. The structure, along with the parsing procedure, will be explained now.

Threads are organized by months. If a thread spans multiple months, it gets split into different threads for the respective month. An indicator that tells that the thread was already discussed is the foregoing *'Re:'* in the e-mail subject. By starting with the oldest e-mails and working through to today's date, we can open up new threads and then append ongoing ones to the next months. This allows us to maintain the coherence of each thread.

As we only want to retrieve all e-mail threads regarding QEMU bugs, we concentrate on thread titles, which contain the word *bug*, case-insensitive, enclosed in square brackets. All the other threads are discarded, as they do not contain relevant information for us.

As mentioned in 3.1, all reports on the Launchpad were notified per e-mail with a link to the Launchpad website. The titles of those threads follow the form *[Bug <launchpad-id>]*, whereas *<launchpad-id>* is a Launchpad-wide unique identifier. The parsing of those reports is discussed in 4.1.3.

Given a thread, we can use the reference link, which redirects us to the next e-mail of a thread, and append all the e-mail contents to one file until we reach the last e-mail. As a result, we get a flattened mailing list archive with all the discussed bugs over the last ten years.

4.1.3 Launchpad

Launchpad offers an API to retrieve all of the information of a report [Lau]. We can construct the needed URL due to having the identifier of the report from the last step. We extract the title, the content of the report and all replies. By concatenating these fields, we get a conversation, which is similar to a mailing thread.

4.2 Classification using LLMs

We tried two types of classification: *Zero-Shot-Classification* and prompting using *Ollama*. Both ideas will be explained in the following.

4.2.1 Zero-Shot Classification

Zero-Shot Classification [Hug] is a method used in LLMs where you try to classify text which has not been seen by the model in the training process. The LLM takes text, categories and a model as inputs and returns scores for each category. A manual postprocess of the scores is required to extract a category. For instance, we consider an output nonsensical if multiple categories achieve a very high score. Through a

trial-and-error method, we find appropriate score thresholds and apply them. The following cycle describes the process:

1. Add categories.
2. Apply thresholds.
3. Inspect the output of the classifier. Evaluate the categorization.
4. Repeat if necessary.

We observed that for different models, different thresholds need to be set up, leading to the classifier being dependent on the model. One would have to manually set up meaningful thresholds for each used model to guarantee the best results.

It is important to keep in mind that the only adjustments we can make, apart from the model and categories, are the thresholds. Even if they are set perfectly - which is practically impossible - we can never detect outright false categorizations of a model. Because of that, we introduce a **comparison model** to the classifier. A classifier uses the second model on the same categories and text to get another, independent result. It is important to use a model that differs heavily from the original one, as the differences in faulty bugs are more recognizable. This additional measure helps us to get the best result out of *Zero-Shot Classification*.

4.2.2 Ollama

The very first approach of classifying the bugs was to use the online version of DeepSeek [deea]. By first prompting a mistranslation as a reference, it is possible to let the LLM decide whether the next bug is a mistranslation or not. As the results were promising, we decided to run different models locally using *Ollama*. This framework allows us to pull various models from the internet easily and embed them in our scripts.

We have to prepare a prompt as an input containing the report, assignment and categories. Then we need to postprocess the response of the LLM. This is done by extracting the category.

One advantage of this approach over the *Zero-Shot Classification* is the ability to explain categories to the model. On the other hand, we also observed instances of the LLM disobeying the assignment. If the text of the report is too long, the model "forgets" the task it was given and tries to help the issuer, presumably due to the insufficient context window size. We mitigate that by rearranging the order of the prompt and by shortening the reports. This can be done by removing comments, as they do not add useful information to the report itself.

4.3 Expanding Focaccia

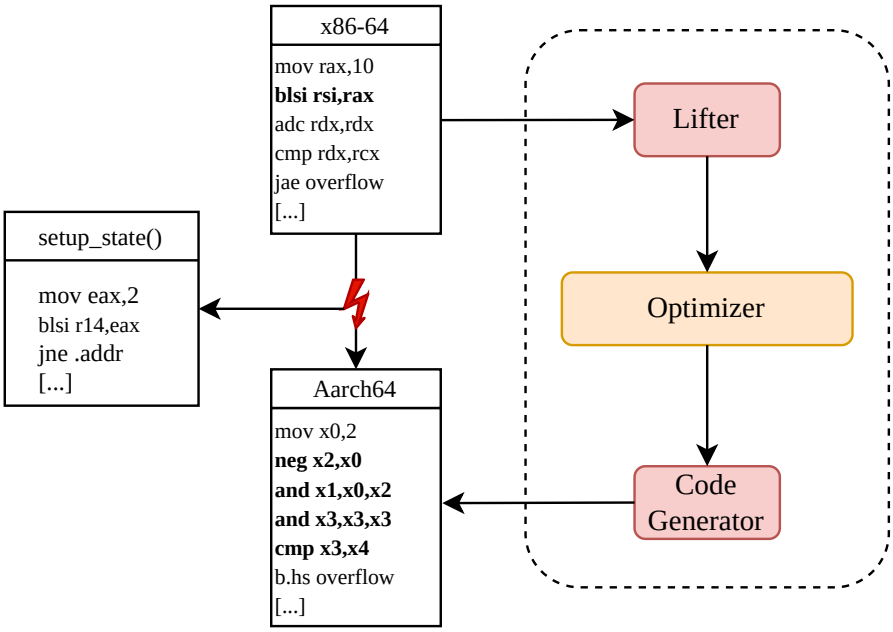


Figure 4.1: An example for a mistranslation of the `BLSI` instruction. *The flags set by the instruction are not set correctly by the translation. Therefore, the semantic equivalence is not given and a minimal reproducer is generated.*

As a result of the bug study, we collected various mistranslation errors for emulators, which we can try to reproduce using Focaccia. Figure 4.1 shows an example of a mistranslation. Focaccia aims to detect the semantical inconsistency between the native and the emulated states.

When Miasm does not support the instruction in question, Focaccia is not able to detect the mistranslation error. Therefore, we need to implement the missing instructions. In the following, we will describe how Miasm works and what needs to be done to enable support for a specific instruction.

Miasm's execution of a function consists of two steps:

- Find the appropriate mnemonic and operands for a specific opcode.
- Execute the associated function - which resembles the instruction - and write the value into the destination operand.

All instruction opcode combinations - linked with their associated semantic function - are given in a list. When executing machine code, Miasm compares the given bits

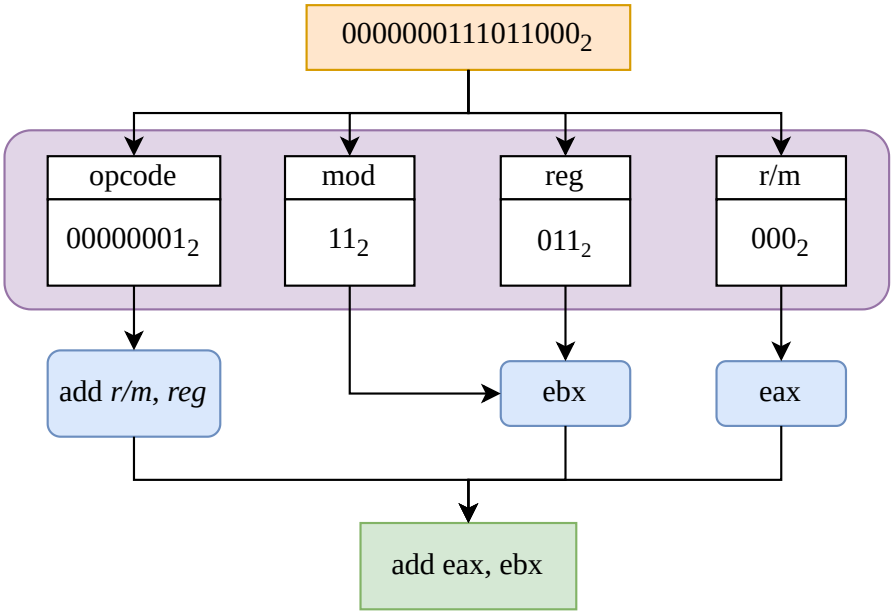


Figure 4.2: An example for disassembling an x86-64 instruction in Miasm. *The instruction gets split into dedicated bitfields, implemented by Miasm. The opcode describes the mnemonic* `add` *and the order of operands. The operands are described by the fields mod, reg and r/m. Each bitfield executes its own encoding, and all modules get combined to form the instruction and its operands.*

to the opcodes in the list. If only one match could be found, Miasm disassembles the instruction. This is achieved by using a modular approach to implementing opcodes. Each opcode consists of different bitfields with a dedicated purpose. The encoding to an internal representation is done by each bitfield, see Figure 4.2. By having every bitfield of the opcode encoded, a complete internal representation of the instruction can be constructed.

The difficulty varies between these two steps: while it is easy to implement the logic of an instruction in a high-level programming language, it is rather difficult to add new opcodes. Here you have to consider prefix-bytes, different operand sizes and completely new prefixes. Our task is the following: use existing bitfields to specify the opcodes of unsupported instructions and manually add missing bitfields.

5 Implementation

5.1 Identifier

We need a unique identifier for each report. For Box64, we take the issue's ID. For QEMU, we need to prevent collisions between different sources:

- For the GitLab reports, we can take the internal IDs of the GitLab issue. This is the identifier that is unique in the project's scope. The GitLab issue IDs range from single-digit to four-digit numbers.
- Launchpad reports are also assigned an ID already. These range from six-digit to seven-digit numbers.
- Lastly, the mailing list threads have no identifiers whatsoever, so we have to build them ourselves. This is done by hashing the title of each thread and truncating the number to a length of eight digits. Though unlikely, we manually ensure that there are no hash duplicates.

Different lengths of identifiers ensure that there are no collisions between different sources.

5.2 Creating the Prompt

Table 5.1 shows the prompt templates that were used as the input for the LLMs. After the bug report is presented, the assignment is given, and the categories are explained. These categories are the result of a thorough trial-and-error approach, which included evaluating LLM results and manually reviewing the bugs.

In the beginning, we aimed to categorize every QEMU bug report, including the system-mode ones. Categories like `boot`, `graphic` and `os` were present. We decided to quickly filter out system-mode bug reports, as they are inherently more complex than user-mode ones.

To discard all reports that are non-user-mode related in QEMU, we decided to add an additional classification. This was not necessary for Box64 and FEX, as they are user-mode emulators and therefore contain mostly user-mode related bug reports.

Input	Prompt
QEMU reports	<bug-report> *This is a bug report regarding qemu.* *Classify the bug report into either 'user-mode' related,* *'system-mode' related or 'other'.* *System-mode reports often include higher privileges,* *peripherals, devices and operating systems.* *The 'other' category includes non-runtime reports, for example* *feature requests, documentation or build issues.* *Respond only with one word, either 'user', 'system' or 'other'.*
QEMU user-mode reports	<bug-report> *Classify the given bug report. It is part of qemu.* *These are the possible categories:* *instruction: A faulty instruction is described in the bug report.* *syscall: An error with a syscall.* *runtime: Other errors which happen in runtime.* *Respond only with a single word, the name of the category.*
Box64/FEX reports	<bug-report> *Classify the given bug report. It is part of Box64/FEX* *These are the possible categories:* *instruction: A faulty instruction is described in the bug report.* *syscall: An error with a syscall.* *runtime: Other errors which happen in runtime.* *other: These are bug reports, which have nothing to do with* *running the program.* *This includes for example: building the program, documentation* *or feature requests.* *Respond only with a single word, the name of the category.*

Table 5.1: Prompt templates used for the automated classification by LLMs.

We decided on these three user-mode emulator bug categories:

- `mistranslation`: An incorrect translation of the semantics of the emulated program.
- `syscall`: An incorrect emulation of a system call invoked by the emulated program.
- `runtime`: Groups all remaining errors that lead to an incorrect execution.

5.3 Miasm

We will implement the x86-64 `BLSI` instruction, which copies the lowest set bit from the SRC operand to the DEST operand and sets all the other bits to 0. The carry flag is set

if the SRC operand has the value 0 and thus no bit set. In QEMU-7.2 [Yon], the carry flag is always set wrong. The bug is currently not reproducible, because Miasm - the symbolic execution backend of FOCACCIA- does not support the instruction. It is part of the BMI extension and uses the VEX-prefix, an opcode prefix that enables the use of extended registers, among other things. As the VEX-prefix is not supported by Miasm, we need to implement it in order to make the `BLSI` instruction work.

To understand how the VEX-prefix works, we will start by exploring the predecessor, the REX-prefix.

Table 5.2: Table of the REX encoding bits.

7	6	5	4	3	2	1	0
0	1	0	0	W	R	X	B

Table 5.2 shows the bit-encoding of the REX-prefix. We need a constant, which indicates the REX-prefix, and four modifiable bits. Their purpose is the following [Int25]:

- **W**: This bit changes the operand size from 32 to 64 bits.
- **R**: This bit expands the *reg* field of the *ModR/M* byte to four bits, so that we can address the new *r8* to *r15* registers.
- **X**: This bit expands the *SIB* (Scale/Index/Base) index field.
- **B**: This bit expands either the *r/m* field of the *ModR/M* byte, or the *SIB* base field, depending on the *Mod* field.

The REX-prefix is already implemented. Miasm parses the prefix in the *pre-disassembling* stage, where it sets specific global variables, depending on those four bits. With this approach, you don't have to specify the prefix in the opcode definition of an instruction, as the prefix is ignored after the first parse. We have to add the VEX-prefix to the same stage.

Table 5.3: Table of the VEX encoding bits.

7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	0
$\overline{R}$	$\overline{X}$	$\overline{B}$	m_4	m_3	m_2	m_1	m_0
W	$\overline{v}_3$	$\overline{v}_2$	$\overline{v}_1$	$\overline{v}_0$	L	p_1	p_0

The bit-encoding of the VEX-prefix can be found in Table 5.3. We again have a constant, which indicates the VEX-prefix, the same four bitfields from the REX-prefix (some of them are given in the complement) and a few new bitfields, which we will cover now [Int25]:

- **m**: This bitfield is an opcode extension. This allows us to distinguish between multiple instructions that share the same opcode.
- **v**: This bitfield is an additional source register. You can therefore have three operand instructions, by using this bitfield and the *ModR/M* byte.
- **L**: This bit changes the operand size from 128-bit to 256-bit for vector instructions.
- **p**: This bitfield encodes additional prefix-bytes.

We can reuse a few components of Miasm to make the VEX-prefix work, for instance, the four bitfields from the REX-prefix. Additionally, we map already implemented prefix-bytes to the encoding of the **p** bitfield. We can also use the implementation of **W** for implementing the **L** bit.

The only bitfield that needs a completely new implementation is the **v** operand. We create a module, which encodes the value inside that bitfield, complements it, and then returns the addressed register. But because the operand is specified in the prefix, our bitfield has a length of zero bits and gets its value from the before parsed prefix value. This way, we can specify that this instruction uses this operand and can also set the order of the operands.

Now that all bitfields that are needed for the `BLSI` instruction are implemented, we can add the appropriate opcode to the list of all available opcodes.

We set the constant bitfields:

- **m** = 2,
- **opcode** = 0x3f and
- **reg** = 3.

Then we specify our new **v** bitfield as the first operand and the already existing **rm** and **mod** bitfields as the second operand.

We link this opcode to our new function *blsi*, which emulates the instruction. It calculates the bitwise AND of the SRC operand with the negated SRC operand:

$$\text{DEST} = \text{-SRC} \ \& \ \text{SRC}. \tag{5.1}$$

We set the zero, overflow, and carry flags accordingly and return the result. The instruction `BLSI` is now fully supported in Miasm.

6 Evaluation

6.1 Plan

In the following, we will evaluate the accuracy of our bug study and the support for new Miasm instructions.

Our bug study evaluation consists of two parts:

- Benchmark different models based on the accuracy of their classification of all QEMU user-mode bugs.
- Expand the evaluation to the other user-mode emulators, Box64 and FEX, in this case.

Table 6.1: Overview of the benchmarked models.

Model	Parameters	Library	Thinking-mode
BART-large	407m	Hugging Face	-
Qwen3	32b	Ollama	-
DeepSeek-R1	32b	Ollama	enabled
DeepSeek-R1	32b	Ollama	disabled
DeepSeek-R1	70b	Ollama	disabled
Gemma3	27b	Ollama	-

Table 6.1 describes the different models we are testing. With this selection, we want to cover the significance of parameters, libraries and the thinking mode (where the model reasons with itself before giving a final answer) on the quality of the classification.

A manual classification of all QEMU user-mode bugs will be conducted to set a baseline for the benchmark. The accuracy of a model will be based on the number of misclassifications. There, we will differentiate between two types: false positives and false negatives. If a non-mistranslation bug report was considered as one, then we call it a false positive. A false negative happens if a mistranslation bug was classified as a non-mistranslation bug. Due to the premise that an automated classification should find all mistranslations, we prioritized minimizing false negatives.

6.2 QEMU

6.2.1 Manual Classification

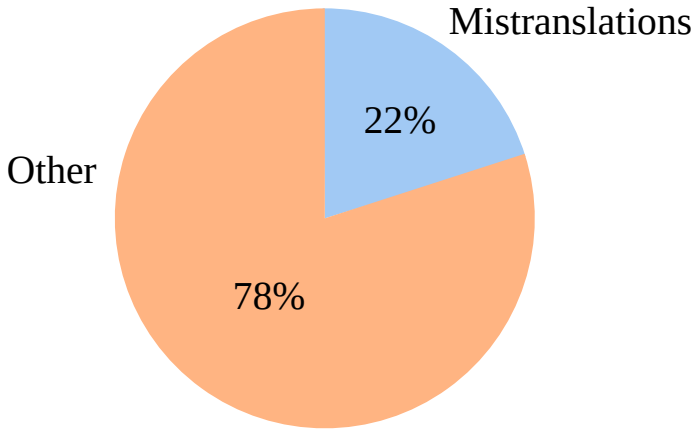


Figure 6.1: Manual classification of QEMU user-mode bugs, focused on mistranslations. *Out of 551 user-mode bugs, 119 are mistranslation ones.*

The result of collecting and classifying QEMU bugs over the course of the last ten years is the following:

- We got 5812 bugs in total,
- 551 user-mode bugs and
- 119 mistranslation bugs (see Figure 6.1).

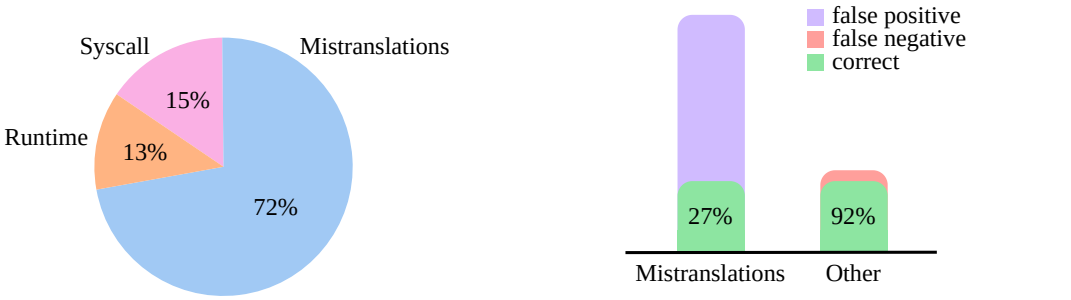
These numbers will be our reference for the classification via LLMs.

6.2.2 BART-large

Our first model *BART-large* uses the *Zero-Shot Classification*. The results of the classification can be seen in Figure 6.2. Compared to our manual classification, the model is too inclusive regarding mistranslations. Additionally, we have too many false negatives, which makes this classification highly inaccurate in total.

We assume two reasons why this classification did not succeed:

- The model has an insufficient number of parameters.



(a) Distribution of error categories.

(b) Quality of classification based on false positives and false negatives.

Figure 6.2: Results of the classification using the BART-large LLM.

- The approach of the *Zero-Shot Classification* might be inappropriate for this task due to being more focused on casual texts and not technical topics.

As we have no other options to adjust the classifier to get better results, we claim that this method was not successful.

6.2.3 Qwen3

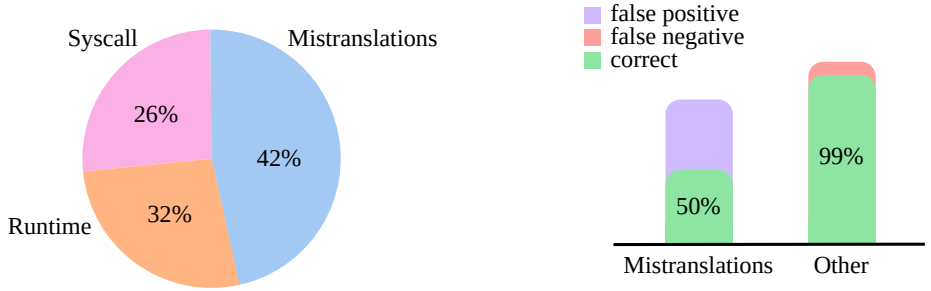
The model Qwen3, which is a bigger model that uses the approach of prompting, got far better results in the classification, compared to the BART-large LLM. The results can be seen in Figure 6.3. The amount of falsely classified non-mistranslation bugs is smaller. Other than that, the quality of the classification is not sufficient for our needs. With the premise that half of the bugs, which were classified as a mistranslation, are categorized falsely, we cannot rely on the classification at all. Additionally, we discarded some actual mistranslation bugs, which is not acceptable.

6.2.4 DeepSeek-R1

In this section, we want to determine two things: the effect *(1)* of the thinking mode and *(2)* of heavier models on the quality of results.

Thinking Mode

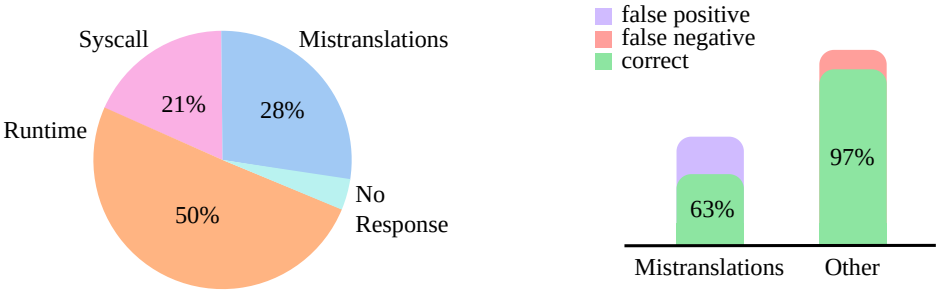
By looking at the distribution of the bugs across the categories in Figure 6.4 and Figure 6.5, we can see small differences between the ratios. What stands out in the pie



(a) Distribution of error categories.

(b) Quality of classification based on false positives and false negatives.

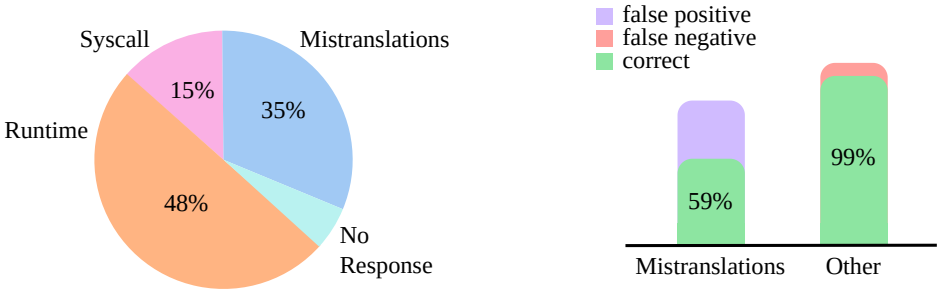
Figure 6.3: Results of the classification using the Qwen3 LLM.



(a) Distribution of error categories.

(b) Quality of classification based on false positives and false negatives.

Figure 6.4: Results of the classification using DeepSeek-R1 (32b) with thinking mode enabled.



(a) Distribution of error categories.

(b) Quality of classification based on false positives and false negatives.

Figure 6.5: Results of the classification using DeepSeek-R1 (32b) with thinking mode disabled.

chart is that some bug reports could not be classified by the model, even after reviewing the responses manually.

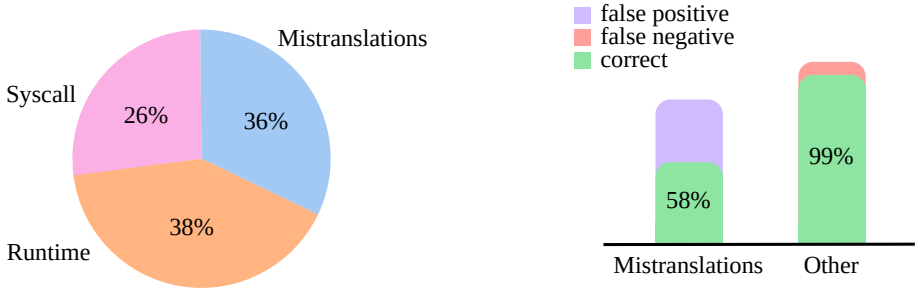
By taking a look at the accuracies, we observe that with the thinking mode disabled, the results are slightly better, as the amount of false negatives is smaller. But this gets overshadowed by the fact that this mode is much more inclusive regarding the mistranslation category, as we also have significantly more false positives. The quality of both results - even if better than the results of the *BART-large* model - is insufficient due to having false negatives.

Heavier model

Figure 6.6 shows the results of the classification using the 70b model of DeepSeek-R1. We observe that all bug reports could be classified by the model. However, the overall accuracy of the classification is similar to the smaller models. The fact that we still have false negatives in our classification makes this model insufficient for our needs.

6.2.5 Gemma3

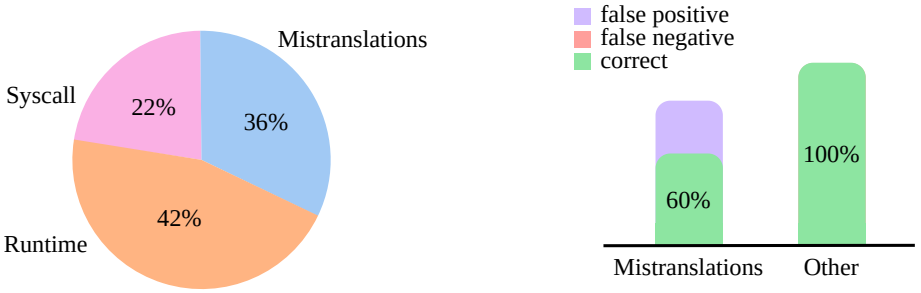
The LLM *Gemma3* is a model by `Google`, based on *Gemini*. It is a lightweight, open-source and open-weight alternative to *Gemini*. The results of the classification can be seen in Figure 6.7. This is the best-performing model we benchmarked, due to having no false negatives whatsoever. However, it has produced around 40% of false positives again.



(a) Distribution of error categories.

(b) Quality of classification based on false positives and false negatives.

Figure 6.6: Results of the classification using DeepSeek-R1 (70b) with thinking mode disabled.



(a) Distribution of error categories.

(b) Quality of classification based on false positives and false negatives.

Figure 6.7: Results of the classification using the Gemma3 (27b) LLM.

With this model, we reduced an initial set of user-mode bug reports to a significantly smaller subset, which still includes all of the actual mistranslation bug reports. While we should aim for making this subset even smaller, this result can be considered a success. A manual review of the mistranslation bugs is still needed.

6.2.6 Summary

We can summarize the following as a result of QEMU's evaluation:

- Heavier models do get better results in a classification.
- The thinking mode of the DeepSeek-R1 LLM doesn't impact the quality of the classification significantly.
- Models by different companies do indeed show differences, though not significant.

We suspect that none of the benchmarked models were trained on either emulation bugs or assembly, which explains the quality of the results.

Next, we will extend the classification to the user-mode emulators Box64 and FEX.

6.3 Box64

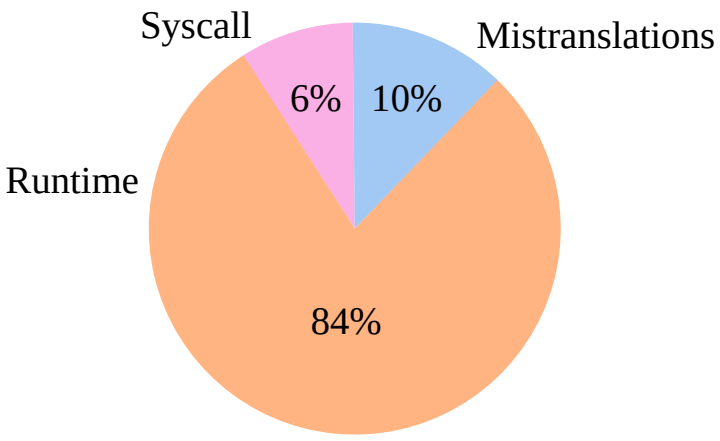


Figure 6.8: Classification results of Box64 issues using an LLM and a manual review afterwards. *10% of all user-mode related issues are mistranslations. The classification exposes a lower quality of the issues than for* QEMU *reports.*

After downloading all of the Box64 issues, we got 1265 issues in total. As Box64 is a user-mode only emulator, we don't have any system-mode bug reports. However, we

have feature requests, documentation-related issues and building-related issues, which we need to eliminate. For that, we introduce the category `other` which catches these kinds of issues automatically. As a result, we have 1093 user-mode related issues. After the classification of the *Gemma3* model, we manually review all of the bugs, which were categorized as mistranslations.

The end result of this classification can be seen in Figure 6.8. We observe a share of 10% of mistranslation bugs. The ratio is quite smaller compared to QEMU, which could stem from the fact that there is no user debugging happening for Box64. This hides potentially mistranslation bugs, as the root cause of those issues is not discussed.

6.4 FEX

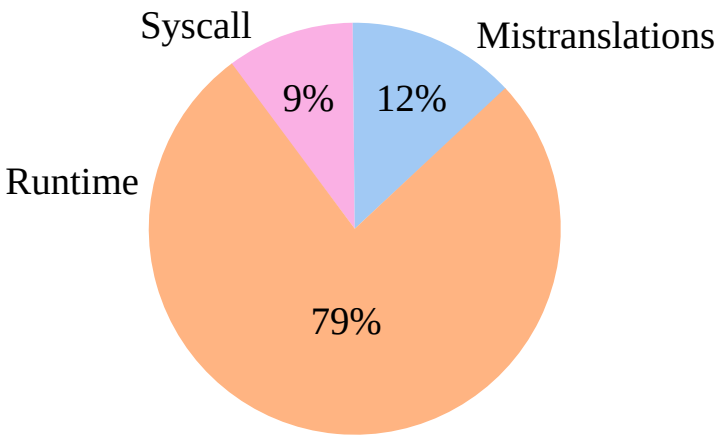


Figure 6.9: Classification results of FEX issues using an LLM and a manual review afterwards. *8% of all user-mode related issues are mistranslation. The quality of the bug reports are unreliable for our purpose.*

We gathered 856 issues for the user-mode emulator FEX- 601 after removing non-runtime issues. The results can be seen in Figure 6.9. While the share of 12% of mistranslation is higher than for Box64, FEX suffers from being Work-in-Progress and therefore not being used that widely. Around half of the issues are written by the maintainers, which leads to a lower quality of the issues. Nearly all of the bug reports, which we consider a mistranslation in Figure 6.9, are actually translation-optimization issues: FEX produces apparently a lot of redundant instructions in the translation process. It is debatable if these kinds of issues are mistranslations. However, we can say that FEX's issues do not show meaningful results for the bug study.

6.5 Focaccia

Table 6.2: Overview of faulty bugs in user-mode emulators.

Instruction	Guest	Host	Emulator	Supported in Miasm
`BLSI` [Yon]	x86-64	any	QEMU-7.2	no
`BLR` [Sij]	Arm	any	QEMU-2.0	no
`CMPXCHG` [Leo]	x86-64	any	QEMU-6.0	yes
`LDSMAX` [Lhu]	Arm	any	QEMU-6.0	no
`CMP` [Coe]	x86-64	RISC-V/Arm	Box64-0.3.0	yes

As a result of the bug study, we found various bugs related to mistranslations. Table 6.2 shows some examples for incorrect instructions. We implemented the `BLSI` instruction, such that it is supported by Miasm and therefore the bug [Yon] reproducible by Focaccia.

```
capture−transforms −o oracle.trace blsi.out
qemu−x86_64 −g 12345 blsi.out &
validate−qemu −−symb−trace oracle.trace localhost 12345
```

Figure 6.10: The testing process of Focaccia. *We use utilities, which are provided by* Focaccia *to (1) create a symbolic trace of the program and (2) compare it to the trace provided by* QEMU.

We prepare a statically linked x86-64 executable that contains the `BLSI` instruction and link Miasm to our fork that supports the instruction. Additionally, we fetch the appropriate QEMU version using the Nix package manager [Nix]. Figure 6.10 describes the steps we need to take to test the emulation of our x86-64 executable by QEMU.

Figure 6.11 shows the output of Focaccia when validating QEMU's execution of the `BLSI` instruction. We observe that the output mismatch, described in the bug report, was successfully found.

With the addition of the VEX-prefix, we can easily add various instructions that use this prefix. We expect the addition of other extensions and ISAs to be similarly uncomplicated.

In general, we can say that the addition of new instructions to Miasm is not complex. While you have to understand the logic of the architecture on a bit level, the actual implementation of the logic in Miasm is simple. Additionally, the addition of the semantics of an instruction is straightforward in a higher-level language, given that the

instruction operation is already described in most ISA manuals [Int25; ARM21].

1. [ERROR] Content of register CF is false. Expected value: 0x0, actual value
in the translation: 0x1.

Expected transformation: Symbolic state transformation 0x401040 –> 0x401045:
[Symbols]
ZF = (RAX & (–RAX))?(0x0,0x1)
SF = (RAX & (–RAX))[63:64]
OF = 0x0
CF = RAX?(0x1,0x0)
RBX = RAX & (–RAX)
RIP = 0x401045
[Instructions]
BLSI RBX, RAX
Actual difference: Snapshot (x86_64): {'RSP': '0x0', 'RIP': '0x5'}

`For PC=0x401040`

Figure 6.11: Validator output for QEMU's execution of the faulty `BLSI` instruction. *The error of the inverted carry flag that was described in the issue is found successfully.*

7 Related Work

In our work, we focused on multiple areas, which are all relevant bodies of research. In the following, we will explore all of those areas on their own.

Text Classification using LLMs

Classification of text in general is an upcoming field of research, since the advent of LLMs. One example is CARP [Sun+23], which splits the classification process into multiple requests, to achieve a more fine-tuned results, making it a more sophisticated approach compared to our bug study. CARP works on low-resource setups and achieves remarkable results in benchmarks.

Lenzmann describes an approach of comparing multiple models based on their classification performances [Len]. In general, this approach is quite similar to ours, but with the important difference that the prompt did not include any categories. In this work, we had fixed categories that we gathered from previous iterations.

Bug study

There exists no bug study of emulators, so we will focus on bug studies for other software systems.

A bug study of bugs in open source software [Tan+14] answers the question of how to reduce errors in software. For that, an understanding of occurring bugs is needed, which is a similar goal to that of our bug study. The questions of the distribution, root causes, impacts and correlation of bugs are answered in that study by analyzing an impressive amount of 100,000 bugs using machine learning techniques. Among many other things, the bug study points out semantic bugs, described as inconsistencies with the requirements, as a dominant root cause.

Validation of Emulators

PokeEmu [Mar+12] is a fuzzer that performs high-coverage testing and cross-validation of processor emulators. It differs between high-fidelity emulators and low-fidelity

emulators. An emulator is a high-fidelity one if it models real CPU behaviour on a low level, focusing on correctness. It is a low-fidelity one, if it uses complex strategies to emulate instructions on a high level, focusing on performance. A high-fidelity emulator is then used to validate a low-fidelity one.

EmuFuzzer [Mar+09] uses fuzzing with an automated test-case generator. Focaccia is similar in the way it compares the emulated guest state with the oracle. It differs from EmuFuzzer by focusing on large-scale non-deterministic applications and the semantic equivalence of their translation.

In a work by Kim et al. [Kim+17], symbolic execution is used to differentially test multiple binary lifters. While it effectively finds mistranslations happening in binary lifters by applying systematic high-coverage testing techniques, it doesn't test the entire emulation process, other than Focaccia.

8 Conclusion

This thesis deals with the process of the analysis of emulator bug reports, with a focus on the design and implementation of the extension of Miasm.

We collected bug reports for different CPU emulators and analyzed user-mode errors by using an LLM. Additionally, we compare multiple models with each other in order to determine which of them provides the best classification of user-mode emulator bug reports. Unfortunately, no model could give results that remove the necessity of a manual review.

After conducting the bug study, we reproduced reported mistranslation errors in Focaccia. For one bug, which occurred due to the faulty `BLSI` instruction, we couldn't reproduce the error, because the instruction was not supported by Miasm. To remedy that, we extended the symbolic execution backend and made the bug recognizable and reproducible for Focaccia.

9 Future Work

In a subsequent work, one can investigate the effect of using fine-tuned models, compared to the base models in this work. This can be done by using a Low Rank Adaptation (LoRA) [Mic] in LLMs. LoRA is designed to fine-tune heavy LLMs by reducing the number of trainable parameters. Therefore, LoRA is resource-efficient and scalable, without discarding the initial base model. We can use the already gathered dataset from our bug study for fine-tuning a model. As the results of Google's gemma3 (27b) model were the most promising ones, we would start with adapting this model. In the end, we might get a purely automated classification of user-mode emulator bug reports regarding mistranslations.

Another idea for future work is the extension of Miasm to support even more bug-related instructions. Now that the VEX-prefix for x86-64 is implemented, we can start by adding support for the AVX extension, which additionally introduces new registers. Additionally, we will focus on the Arm and RISC-V architectures, as the set of implemented instructions is even smaller than for x86-64. This includes going through the gathered mistranslation bug list of the user-mode emulators and enabling support for the faulty instructions. In general, the main goal is to make Miasm as applicable as possible, and with the preliminary work done, we are able to tackle and add new instructions for various ISAs.

Abbreviations

ISA Instruction Set Architecture

LLM Large Language Model

TB Translation Block

IR Intermediate Representation

LoRA Low Rank Adaptation

URL Uniform Resource Locator

HTML Hypertext Markup Language

TOML Tom's Obvious, Minimal Language

List of Figures

2.1 Overview of the binary translation approach on the example of a RISC-V to ARM translation. *A sequence of instructions in a guest ISA gets lifted to an Intermediate Representation. The IR gets optimized and the code for a host ISA is generated.* 4

2.2 Overview of the training process of an LLM. *The dataset gets tokenized and the sequence of tokens learned by creating probabilities for a token prediction. Afterwards, the model gets fine-tuned manually, which results in a ready-to-use LLM.* 6

3.1 Overview of the classifier. *A bug report and fixed categories (along with explanations) get formed into a prompt that is presented to an LLM. The output has to be parsed so that the category can be extracted.* 9

3.2 Overview of Focaccia. *A program gets executed by a symbolic execution backend and by the emulator. The backend creates symbolic equations and an expected next state, using the equations and the current state of the emulator. The next state of the emulator is then compared to the expected state. If a mismatch occurs, a minimal reproducer program is generated.* 10

4.1 An example for a mistranslation of the `BLSI` instruction. *The flags set by the instruction are not set correctly by the translation. Therefore, the semantic equivalence is not given and a minimal reproducer is generated.* 14

4.2 An example for disassembling an x86-64 instruction in Miasm. *The instruction gets split into dedicated bitfields, implemented by Miasm. The opcode describes the mnemonic `add` and the order of operands. The operands are described by the fields mod, reg and r/m. Each bitfield executes its own encoding, and all modules get combined to form the instruction and its operands.* 15

6.1 Manual classification of QEMU user-mode bugs, focused on mistranslations. *Out of 551 user-mode bugs, 119 are mistranslation ones.* 21

6.2 Results of the classification using the BART-large LLM. 22

6.3 Results of the classification using the Qwen3 LLM. 23

6.4 Results of the classification using DeepSeek-R1 (32b) with thinking mode enabled. 23

6.5 Results of the classification using DeepSeek-R1 (32b) with thinking mode disabled. 24
6.6 Results of the classification using DeepSeek-R1 (70b) with thinking mode disabled. 25
6.7 Results of the classification using the Gemma3 (27b) LLM. 25
6.8 Classification results of Box64 issues using an LLM and a manual review afterwards. *10% of all user-mode related issues are mistranslations. The classification exposes a lower quality of the issues than for* QEMU *reports.* . . 26
6.9 Classification results of FEX issues using an LLM and a manual review afterwards. *8% of all user-mode related issues are mistranslation. The quality of the bug reports are unreliable for our purpose.* 27
6.10 The testing process of Focaccia. *We use utilities, which are provided by* Focaccia *to (1) create a symbolic trace of the program and (2) compare it to the trace provided by* QEMU. 28
6.11 Validator output for QEMU's execution of the faulty `BLSI` instruction. *The error of the inverted carry flag that was described in the issue is found successfully.* 29

List of Tables

5.1 Prompt templates used for the automated classification by LLMs. 17
5.2 REX encoding bits 18
5.3 VEX encoding bits 18

6.1 Table of benchmarked models 20
6.2 Overview of faulty bugs in user-mode emulators 28

Bibliography

[ARM21] ARM. *Armv7-M Architecture Reference Manual*. 2021.

[Bro+20] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[Coe] Coekjan. *Unexpected Dynarec-Interpreter Difference on cmp Instruction*. Accessed on: 2025-07-30. URL: https://github.com/ptitSeb/box64/issues/1661.

[deea] deepseek-ai. *DeepSeek-Online*. Accessed on: 2025-07-30. URL: https://chat.deepseek.com.

[deeb] deepseek-ai. *DeepSeek-R1*. Accessed on: 2025-07-30. URL: https://github.com/deepseek-ai/DeepSeek-R1.

[FEX] FEX-Emu. *FEX GitHub Issue tracker*. https://github.com/FEX-Emu/FEX/issues. Accessed: August 07, 2025.

[Gita] GitHub. *GitHub Issues API*. URL: https://docs.github.com/en/rest?apiVersion=2022-11-28.

[Gitb] GitLab. *GitLab Issues API*. URL: https://docs.gitlab.com/api/issues/.

[Haj] S. Hajnoczi. *Migrating custom qemu.org infrastructure to GitLab*. URL: https://mail.gnu.org/archive/html/qemu-devel/2020-07/msg02776.html.

[Hug] HuggingFace. *Zero-Shot Classification*. URL: https://huggingface.co/tasks/zero-shot-classification.

[Int25] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual: Instruction Set Reference, A-Z, Volume 2 (2A, 2B, 2C, 2D)*. 2025.

[Kim+17] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha. "Testing intermediate representations for binary analysis." In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 353–364. DOI: 10.1109/ASE.2017.8115648.

[Lau] Launchpad. *Issues API*. URL: https://launchpad.net/+apidoc/.

[Len] O. Lenzmann. *Mastering LLMs for Complex Classification Tasks*. URL: https://medium.com/@olaf.lenzmann/mastering-llms-for-complex-classification-tasks-64f0bda2edf3.

[Leo] I. Leoshkevich. *x86_64 cmpxchg behavior in qemu tcg does not match the real CPU*. Accessed on: 2025-07-30. URL: https://gitlab.com/qemu-project/qemu/-/issues/508.

[Lhu] Y. Lhuillier. *qemu-aarch64 incorrect signed comparison in ldsmax instructions*. Accessed on: 2025-07-30. URL: https://gitlab.com/qemu-project/qemu/-/issues/364.

[Mar+09] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. "Testing CPU emulators." In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA '09. Chicago, IL, USA: Association for Computing Machinery, 2009, pp. 261–272. ISBN: 9781605583389. DOI: 10.1145/1572272.1572303.

[Mar+12] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. "Path-exploration lifting: hi-fi tests for lo-fi emulators." In: *SIGARCH Comput. Archit. News* 40.1 (Mar. 2012), pp. 337–348. ISSN: 0163-5964. DOI: 10.1145/2189750.2151012.

[Mic] Microsoft. *LoRA*. URL: https://github.com/microsoft/LoRA.

[Nix] NixOS. *nix*. URL: https://github.com/NixOS/nix.

[pti] ptitSeb. *Box64 GitHub Issue tracker*. https://github.com/ptitSeb/box64/issues. Accessed: August 07, 2025.

[QEMa] QEMU. *QEMU GitLab Issue tracker*. https://gitlab.com/qemu-project/qemu. Accessed: August 07, 2025.

[QEMb] QEMU. *QEMU Launchpad bug tracker*. https://launchpad.net/qemu. Accessed: August 07, 2025.

[QEMc] QEMU. *QEMU Mailing List*. https://lists.nongnu.org/archive/html/qemu-devel/. Accessed on: 2025-07-30.

[Sij] S. Sijaric. *AArch64 - blr x30 is handled incorrectly*. Accessed on: 2025-07-30. URL: https://bugs.launchpad.net/qemu/+bug/1328996.

[SN05] J. E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series in Computer Architecture and Design. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105.

[Sun+23] X. Sun, X. Li, J. Li, F. Wu, S. Guo, T. Zhang, and G. Wang. *Text Classification via Large Language Models*. 2023. arXiv: 2305.08377 [cs.CL].

[sym] sympa-community. *MHonArc*. https://github.com/sympa-community/MHonArc.

[Tan+14] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. "Bug characteristics in open source software." In: *Empirical Software Engineering* (2014). ISSN: 1573-7616. DOI: 10.1007/s10664-013-9258-8.

[TUM] TUM-DSE. *Focaccia*. URL: https://github.com/TUM-DSE/focaccia.

[Vas+23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].

[Yan+25] B. Yan, K. Li, M. Xu, Y. Dong, Y. Zhang, Z. Ren, and X. Cheng. "On protecting the data privacy of Large Language Models (LLMs) and LLM agents: A literature review." In: *High-Confidence Computing* 5.2 (2025), p. 100300. ISSN: 2667-2952. DOI: https://doi.org/10.1016/j.hcc.2025.100300.

[Yao+24] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang. "A survey on large language model (LLM) security and privacy: The Good, The Bad, and The Ugly." In: *High-Confidence Computing* 4.2 (2024), p. 100211. ISSN: 2667-2952. DOI: https://doi.org/10.1016/j.hcc.2024.100211.

[Yon] L. Yong-Woo. *x86 BLSI and BLSR semantic bug*. Accessed on: 2025-07-30. URL: https://gitlab.com/qemu-project/qemu/-/issues/1370.