Département des Technologie de l'information et de la communication (TIC)
Filière Télécommunications
Orientation Sécurité de l'information

Travail de Bachelor

Improving SymQEMU

Étudiant	**Damien Maier**
Enseignant responsable	Prof. Alexandre Duc
Entreprise mandante	Prof. Aurélien Francillon EURECOM 450 Route des Chappes 06904 Biot France
Année académique	2023-2024

Yverdon-les-Bains, le December 23, 2023

Département des Technologie de l'information et de la communication (TIC)
Filière Télécommunications
Orientation Sécurité de l'information
Étudiant : Damien Maier
Enseignant responsable : Prof. Alexandre Duc

Travail de Bachelor 2023-2024

Improving SymQEMU

Nom de l'entreprise/institution

EURECOM

Résumé publiable

SymQEMU is a software testing tool based on QEMU, that automatically discovers interesting inputs for a target program, using symbolic execution. Its design makes it orders of magnitude faster than traditional symbolic execution tools like KLEE, which allows to use it as an efficient support for a fuzzer like AFL++ to increase code coverage when testing real-world software.

In this bachelor thesis, we first present a technical analysis of how SymQEMU works, and how it modifies QEMU to enable it to perform both a normal and a symbolic execution at the same time. In particular, we give a detailed explanation of some internal aspects of QEMU that SymQEMU modifies, which were previously not or only poorly documented.

We then describe two improvements that we made to SymQEMU.

The first contribution is a port of SymQEMU to QEMU 8.1, the most recent version of QEMU. While realizing this work, we also added a useful debugging feature to SymQEMU, that helps to analyze its symbolic execution process and to compare two different versions of the software.

The second improvement is the added support for SIMD style instructions. The original SymQEMU lacked support for SIMD instructions, which was a non-negligible limitation as those instructions are common in real-world software. Thanks to our contribution, SymQEMU is now able to correctly test software that uses SSE X86 instructions.

Étudiant :	Date et lieu :	Signature :
Damien Maier	.....	.....
Enseignant responsable :	Date et lieu :	Signature :
Prof. Alexandre Duc	.....	.....
Nom de l'entreprise/institution :	Date et lieu :	Signature :
Prof. Aurélien Francillon EURECOM	.....	.....

Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus détudes, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit dauteur.

HEIG-VD

Vincent Peiris
Chef de département TIC

Yverdon-les-Bains, le December 23, 2023

Authentification

Le soussigné, Damien Maier, atteste par la présente avoir réalisé ce travail et navoir utilisé aucune autre source que celles expressément mentionnées.

Yverdon-les-bains, le December 23, 2023

Damien Maier

Specifications

Context

SymQEMU is a concolic execution engine based on SymCC and QEMU. SymQEMU, together with a fuzzer like AFL++ can be used to perform fuzzing of Linux applications.

Currently, SymQEMU only supports testing x86-64 Linux programs (user mode). However, due to its construction, SymQEMU should be able to be used for the simulation of full virtual machines (system mode) with all architectures supported by QEMU.

An internal project (SyX) already supports full system testing, but the code that adds system mode support is currently mixed with other elements of the SyX project.

In addition, preliminary work was performed to support ARM and MIPS architectures in user mode. In particular, the support for ARM 64 bits programs is probably already good.

Goals

The goal of the project is to obtain a standalone (not SyX dependent) version of SymQEMU that:

- Is based on the latest stable version of QEMU
- Supports testing ARM 64 bits, ARM 32 bits and MIPS programs.
- Supports testing in system mode.

Such a version of SymQEMU would allow to test embedded firmwares.

The tasks to perform are the following:

- Study the relevant parts of the source code of QEMU and SymQEMU.
- Merge the code of SymQEMU (the version that includes some support for ARM and MIPS) with the latest stable version of QEMU.

- Evaluate and improve the support for ARM 32 bits, ARM 64 bits and MIPS architectures. The goal is for SymQEMU to be able to correctly test binaries obtained from simple C programs.
- Integrate the work done in the SyX project to add system mode support. The goal is to get a version of SymQEMU able to correctly test minimal firmwares.

In addition, if time permits, some work on the following projects could be done:

- Using this new version of SymQEMU to test Bluetooth firmwares from commercial devices.
- Using SymQEMU together with Avatar^2. Avatar^2 is a testing orchestration framework created at EURECOM. It provides interoperability between different dynamic binary analysis frameworks, debuggers, emulators, and real physical devices.

Deliverables

The deliverables are the following :

- A report that includes :
  - A description of the modifications and improvements done to SymQEMU
  - An explanation of some relevant internal aspects of SymQEMU
- The source code of a version of SymQEMU able to test software in system mode with ARM and MIPS support.
- An analysis of the results obtained.

Note about the modification of this specification

During the project, it was decided to deviate from those initial objectives. The reasons for this and the new objectives are explained in 1.1.

Contents

Chapter 1

Introduction

Low level software vulnerabilities such as memory bugs are a major cybersecurity issue, as they can have devastating consequences like allowing an attacker to execute arbitrary code on a victim computer.

Due to the complexity and size of modern software, manually analysing program code to search for such vulnerabilities is costly, and often impracticable. As a consequence, automated software testing is a highly active area of research, both in the academical world and in the industry.

Fuzzing is a popular technique for automated memory bugs discovery, that consists of running the program under test on randomly generated inputs [14]. This approach is widely used by security researchers and allowed to discover a large number of critical vulnerabilities in real-world software. However, it stills suffers from some limitations, including the difficulty to discover appropriate program inputs for covering the whole code of the target program [6].

Symbolic execution is an orthogonal testing technique which, among other applications, can be used to automatically discover memory bugs. This approach is based on the idea of executing the program under test on a "symbolic" input rather than a concrete one, which allows to systematically discover all possible execution paths and the inputs that lead to them [18]. Although some testing tools use symbolic execution alone, this technique can also be used as a support for a fuzzer, to help it discover new interesting inputs and increase code coverage [25].

Historically, tools like KLEE or Angr performed symbolic execution by interpreting (instead of executing) the program under test, which strongly limited their efficiency due to the cost of the interpretation [26]. In recent years, researchers proposed a novel approach which consists of directly executing the program under test, while instrumenting it to track computations performed on the inputs and to perform a simultaneous symbolic execution [26].

This execution based technique is orders of magnitude faster than the traditional simulation based approach. It gave promising results, as it allowed, while being used together with a fuzzer, to discover previously unknown critical vulnerabilities in popular programs that had already been heavily tested with state-of-the-art tools [26, 18, 19].

SymQEMU is a software testing tool created at EURECOM that implements this new technique [19]. It is built on top of QEMU, and enables it to perform a symbolic execution of the target program, in addition to the normal execution.

In this thesis, we present the contribution we made to SymQEMU during our internship at EURECOM.

1.1 Contribution

The initial goal of this bachelor work was to improve SymQEMU to make it able to test embedded device firmwares. However, during our internship, this objective was modified in order to investigate unexpected results that we encountered with SymQEMU.

1.1.1 Tasks

During our internship, we first studied the technical implementation of SymQEMU. We familiarized ourselves with the components of QEMU that SymQEMU modifies, and we analyzed how SymQEMU adds to QEMU the ability to perform symbolic execution.

Then, we ported SymQEMU to QEMU 8.1, which is the most recent stable version of QEMU. The original SymQEMU is based on QEMU 4.1, dating from 2019. Updating SymQEMU to QEMU 8.1 allowed to benefit from the various bug fixes and improvements that this version offers.

After realizing this port, we discovered that SymQEMU based on QEMU 8 gave significantly lower quality results than SymQEMU based on QEMU 4 on some target test programs. In agreement with the professors that supervised this project, it was decided to not carry on with the tasks related to firmware testing, and instead to investigate the causes of this change in behavior.

To perform this task, we added a debugging feature to SymQEMU that helps analyze the symbolic execution process and highlight the differences in behavior between two different versions of SymQEMU. As a result of this investigation, we discovered that the root cause of the problem was a change in the way SSE x86 instructions were handled in QEMU 8 compared to QEMU 4.

Finally, we added support for SIMD-style instructions in the new version of SymQEMU. While the original motivation for performing this task was to make the new SymQEMU

perform as good as the original SymQEMU, this improvement actually allowed the new SymQEMU to significantly outperform the old one when testing software that contains x86 SSE instructions.

1.1.2 Organisation of this bachelor thesis

Let us now present the content of the different chapters of this thesis.

In the remaining of the present chapter, we will introduce several concepts related to software testing and symbolic execution.

In 1.2, we will discuss memory errors and the most popular approach for discovering them, which is *fuzzing*. We will also present the concept of *tight branch conditions*, which is an important obstacle for fuzzers.

In 1.3, we will introduce an alternative approach for automatic discovery of memory errors : *symbolic execution*. We will also see why symbolic execution is not affected by tight branch conditions.

In 1.4 we will see how testing tools that use symbolic execution are implemented in practice. We will also discuss the limits of symbolic execution and see that using it to test real-world software is often infeasible due to the problem of *path explosion*.

In 1.6 we will introduce a special flavor of symbolic execution named *concolic execution*. Concolic execution has several advantages that make it suitable for testing real-world software. In particular, it is the approach used in SymQEMU.

Finally, in 1.7 we will go back to fuzzing and introduce an approach named *hybrid fuzzing* that consists of using a symbolic (or concolic) executor as a support for a fuzzer to help it overcome tight branch conditions.

Let us now present the content of the next chapters.

Chapter 2 is a state of the art, where we present an overview of other tools that use symbolic execution for automatic discovery of memory bugs.

Chapter 3 is a technical presentation of several parts of the source code and design aspects of QEMU that need to be understood in order to understand how SymQEMU is implemented. As most of the code of QEMU is not or only poorly documented, we hope this chapter could be a useful resource for future contributors to SymQEMU.

Chapter 4 is a technical description of the implementation of SymQEMU. As for the previous chapter, we hope it could be a useful resource for future contributors as the implementation of SymQEMU is currently little documented.

Chapter 5 presents the work we realized for porting SymQEMU to QEMU 8, and for analyzing its differences in behavior with the original SymQEMU. In particular, we describe a

debugging feature that we added to SymQEMU and we explain how we used it to discover that SSE instructions were not handled identically in the old and the new version.

Chapter 6 presents the work we realized for adding support of SIMD-style instructions to SymQEMU.

In Chapter 7 we analyze how our new version of SymQEMU performs compared to the original one. We show that, even on simple programs, it is able to generate significantly better results when the tested programs contain x86 SSE instructions.

Finally, in Chapter 8 we conclude this thesis and discuss possible future work.

1.2 Memory errors and fuzzing

In this section, we will discuss memory errors and present how fuzzing can be used to discover them.

In 1.2.1, we will first discuss memory errors and their consequences.

Then in 1.2.2 we will present fuzzing, a popular technique for discovering memory bugs. In 1.2.3 we will discuss code sanitizers, which can be combined with fuzzing to increase the probability of discovering bugs. In 1.2.4 we will discuss the problem of code coverage and we will present a fuzzing technique for maximising it named coverage guided fuzzing.

Finally, in 1.2.6 and 1.2.5, we will discuss the main obstacle to code coverage that fuzzers face.

1.2.1 Memory errors

Memory errors are bugs related to low level memory access that can happen in so-called *memory-unsafe* programming languages, like C or C++. Some examples of memory errors are buffer overflows, buffer over-reads, format string bugs, user after free, double free and wild pointers.

Memory errors can have devastating consequences. When an input that triggers such an error is accidentally provided to the vulnerable program, it usually makes the program crash. Furthermore, it can lead to an unexpected behavior of the program before the crash.

More importantly, in some situations a memory bug can allow an attacker to feed the vulnerable program with an input that is especially crafted to trigger the bug and make the program have some unwanted behavior, for example leaking internal data. In most serious cases, memory bugs allow the attacker to hijack the execution flow of the vulnerable program and make it execute malicious code.

1.2.2 Fuzzing

Fuzzing is an automated technique for testing software and finding bugs, especially memory bugs.

The principle of fuzzing is to execute the target program a large number of times, each time with a different randomly generated input, in the hope of discovering an input that triggers a bug [14].

Usually, the criteria to determine that a given input triggers a bug is the fact that the target program crashes when executed on this input.

As memory errors usually lead to a crash when triggered, fuzzing is a popular approach for discovering such bugs.

1.2.3 Code sanitizers

Although memory errors usually lead to crashes, this is not always the case. Triggering a memory error may simply lead to an unexpected behavior of the program, or even have no effect at all, in which case the fuzzer will not detect it.

To increase the probability of discovering memory errors, fuzzing can be combined with *code sanitizers*. A code sanitizer is a tool that instruments a program under test at compilation time, in order to detect memory errors that happen at run time [10]. Note that a code sanitizer only detects memory errors when they actually happen.

A popular code sanitizer is AddressSanitizer [23], created by Google. It can detect memory errors like buffer overflows or use-after-free.

1.2.4 Coverage-guided fuzzing

A central concern with fuzzing is *code coverage*.

To maximize the probability of discovering bugs, it is important to test the target program on inputs that will collectively lead to the execution of as most diverse program code areas as possible. This is necessary because if a bug is present in the target program, but the test inputs all lead to an execution path that does not cover the area of this bug, then the bug will never be discovered.

A popular approach for increasing code coverage is *coverage-guided fuzzing*.

With coverage-guided fuzzing, the fuzzer monitors the execution path taken when the program is executed on a test input, in order to know if an input leads to the execution of new code or not. It then generates random variations of the interesting inputs.

Coverage guided fuzzing typically follows those steps [6]:

1. The input pool of the fuzzer is initialized with some provided input examples.
2. The fuzzer picks an input i of the pool and applies some random mutation on it to obtain a new input i'.
3. The fuzzer executes the target program on i' and monitors the execution path.
4. If the execution of i' leads to the execution of a code area not yet covered by the inputs of the pool, i' is added to the pool.
5. Go to step 2 and repeat the process.

1.2.5 Limits of fuzzing for tight branch conditions

We will now discuss the main obstacle that limits code coverage when testing a program with a fuzzer.

Let us introduce this concept with an example. Consider a simple program that displays images in the fictional *JPIG* format.

The header of a *JPIG* file simply consists of its first 8 bytes, that represent a single signed number. To be a valid *JPIG* image, the header value must be positive. Furthermore, there exists a special extension of the *JPIG* format, called *JPIG++*. If a file is in the *JPIG++* format, then its header must have the exact value `0x13374242deadbeef`.

Our *JPIG* reader program would have the following structure :

Code Source : JPIGReader.c

```
1  void main(){
2
3      int64_t header = 0;
4
5      fgets((char*) &header, 8, stdin);
6
7      if (header > 0) {
8          if (header == 0x13374242deadbeef){
9              /*
10                 process JPIG++ image
11             */
12         } else {
13             /*
14                 process JPIG image
15             */
16         }
17     } else {
18         /*
19             log error
```

```
20          */
21      }
22 }
```

For simplicity, here we assume that the image file content is provided in *stdin*.

Suppose now that the *process JPIG image* code has a memory bug A, that the *process JPIG++ image* has another memory bug B and that we want to discover those bugs with a fuzzer. Furthermore, let's assume that the initial input example that is given to the fuzzer is simply a file filled with bytes having the value 0.

In this situation, the fuzzer will generate new inputs by applying random variations on the provided initial input. Although this initial file only leads to the execution of the *log error* code, the fuzzer will apply some random mutations on it and will easily discover example of files that lead to the execution of the *process JPIG code*. This will allow the fuzzer to discover memory bug A.

It is however very unlikely for the fuzzer to discover an input that triggers the execution of the *process JPIG++ code* just by applying random mutations on the initial input file. Assuming the fuzzer generates random value for the first 8 bytes of the input, there is only one chance over 2^{64} that it will get the exact value `0x13374242deadbeef`. As a result, the code for processing *JPIG++* is never tested and the fuzzer is not able to discover memory bug B.

`if (header > 0)` is an example of a *loose branch condition*, i.e., a branch condition that can be both satisfied and not satisfied by a large number of different values. On the other hand, `if (header == 0x13374242deadbeef)` is a *tight branch condition* because the vast majority of inputs will lead to the same execution path.

1.2.6 Tight branch conditions in real world software

As we have seen, fuzzing is good at discovering inputs that lead to an execution path with loose branch conditions, but it often fails to test paths linked to tight branch conditions [26].

Tight branch conditions may appear early in the execution of real world software, which tends to make large parts of the program code difficult to test for the fuzzer [6]. This happens for example when a program compares parts of the input with magic numbers [6], or when it processes input commands and compares it to a list of possible commands [25]. In such situations, the fuzzer may only test the code that is executed when an invalid input is given to the program, and most of the actual program may not be covered.

As a consequence, fuzzing tools tend to easily discover shallow bugs, i.e., bugs that happen at the early stages of the program execution, but it is more difficult for them to find bugs located deeply in the execution paths tree [25].

1.3 Presentation of symbolic execution

We will now present an orthogonal approach for automatically testing and analyzing software : *symbolic execution*.

Symbolic execution allows to systematically explore the execution paths of a software, and to discover program inputs that will lead the program execution to those paths. The principle of this approach is to consider that the inputs of the program are unknown *symbolic* values, to simulate the execution of the program while tracking the operations performed on symbolic values, and to compute the conditions applied on the program inputs when a branch occurs.

In this section, we will first give an informal introduction to symbolic execution by describing a symbolic execution of the *JPIG* reader program. In 1.3.1 we describe the beginning of the symbolic execution, in 1.3.2 we describe the execution of the first branch condition, and in 1.3.3 we describe the execution of the second branch condition.

In 1.3.4 we show how symbolic execution allows to pass the tight branch conditions and discover program inputs.

Then in 1.3.5 we are going to give precise definitions related to symbolic execution.

Finally, in 1.3.6 we introduce the concept of *execution tree*, which is the tree of all possible execution paths of a program.

1.3.1 Symbolic execution of the JPIG reader program

To illustrate the functioning of symbolic execution, let see how a symbolic execution engine would analyze our *JPIG* reader program.

Code Source : JPIGReader.c

```
1  void main(){
2
3      int64_t header = 0;
4
5      fgets((char*) &header, 8, stdin);
6
7      if (header > 0) {
8          if (header == 0x13374242deadbeef){
9              /*
10                 process JPIG++ image
11             */
12         } else {
13             /*
14                 process JPIG image
15             */
16         }
```

```
17      } else {
18          /*
19              log error
20          */
21      }
22  }
```

In a symbolic execution engine, any value in memory is either *concrete* or *symbolic*. A concrete memory area represents a normal area of the program memory, that contains a known value. A symbolic memory area is a memory area whose value depends on the input of the program.

When the symbolic engine starts the analysis of our *JPIG* reader program, the only symbolic values are the bytes of *stdin*, as they are the input of our program. All other values are concrete.

When the execution engine reaches line 5 of the program, it puts the first 8 bytes of *stdin* into the memory area for the *header* variable. As *stdin* is symbolic, the memory area for the *header* variable does not contain an actual value anymore. Instead, it contains a symbolic value that indicates that it is bytes 0 to 7 of *stdin*.

1.3.2 First branch condition

When the execution engine reaches line 7, it needs to simulate the execution of the *if* statement. However, as the condition depends on the *header* variable, whose value is symbolic, the symbolic engine will check if one or both paths can be taken. As *header* contains the first 8 bytes of *stdin*, and those bytes can contain any value, both branches can be reached.

Suppose that the symbolic engine analyzes the *else* branch first. It will determine that, in order to reach this branch, the bytes 0 to 7 of *stdin*, when interpreted as a signed 8 bytes integer, must represent a number smaller or equal to 0. Let us represent this constraint as `int(stdin[0:7]) <= 0`.

While simulating the execution of the code starting at line 18, the symbolic engine will remember that this constraint holds.

Now let's go back to the *if* statement of line 7 and analyze the branch for the case where the condition is true. The symbolic engine determines that *stdin* must respect the condition `int(stdin[0:7]) > 0` for this branch to be taken.

1.3.3 Second branch condition

When it reaches the *if* statement of line 8, the symbolic execution engine will again check if one or both paths can be reached, as the condition depends on a symbolic value. The path corresponding to the case where the condition of the *if* statement is true is reachable because the constraint `int(stdin[0:7]) > 0 && int(stdin[0:7]) == 0x13374242deadbeef` is satisfiable.

The symbolic engine can then analyze the branch starting on line 9, and it will remember that in this branch, the condition `int(stdin[0:7]) > 0 && int(stdin[0:7]) == 0x13374242deadbeef` is true. This condition can of course be simplified to `int(stdin[0:7]) == 0x13374242deadbeef`.

The path corresponding to the case where the condition of the *if* statement of line 8 is false is also reachable, because there exists some *stdin* bytes such that `int(stdin[0:7]) > 0 && int(stdin[0:7]) != 0x13374242deadbeef`. The symbolic engine can then analyze the branch starting at line 13, and it will remember that in this branch the condition `int(stdin[0:7]) > 0 && int(stdin[0:7]) != 0x13374242deadbeef` is true.

1.3.4 Discovering inputs

For each branch that the symbolic engine explores, it is able to tell exactly which constraints the program input should satisfy to make the execution reach this branch. Using those constraints, it can compute actual examples of inputs that will make the program reach each branch.

Here we see how the symbolic engine can easily discover that the execution of the *process JPIG++ image* will be triggered by giving the value `0x13374242deadbeef` as input.

1.3.5 Definitions

Now that we have seen a concrete examples of symbolic execution, let us introduce some useful definitions.

When performing symbolic execution, we call a *symbol* the undetermined value that represents the program input.

At any point of the execution, a memory area (or a register, a variable, etc.) whose value depends on the program input is called *symbolic*. A memory area whose value does not depend on the program input is called *concrete*.

A symbolic memory area is associated to a *symbol expression* (or sometimes just *expression*). A symbol expression represents either :

- A symbol.

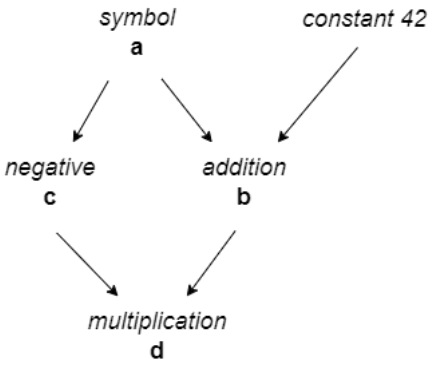

Figure 1.1: Symbol expressions example.

- A constant value (sometimes called a "constant symbol").
- A computation on others symbol expressions. In this case the expression is determined by the type of operation and the other expressions it directly depends on.

If an expression b represents a computation on an expression a, then we say that a is a *parent* of b. More generally, we say that an expression a is an *ancestor* of an expression z if z represents a computation that directly or indirectly depends on a.

To illustrate this, consider the following code :

Code Source :

```
1 void main(){
2
3     char a = getchar();
4
5     char b = a + 42;
6
7     char c = -a;
8
9     char d = b * c;
10 }
```

Figure 1.1 represents the symbol expressions associated to each variable of the program. In this example, c is a parent of d and both c and a are ancestors of d.

We can easily see that this construction allows to express any symbolic value in terms of an expression that depends only on the program input (and constant values). For example, here using the graph we can reconstruct the fact that $d = -a\dot{(}a + 42)$

1.3.6 The execution tree

To conclude this introducing section on symbolic execution, let us now introduce a last useful concept.

One can easily see that the different possible execution paths of a target program form a binary tree. This tree is built the following way:

- The root represents the first program state where the program is about to execute a branch that depends on a symbolic value (i.e., that depends on the input).
- Each child node represents a subsequent program state where the program is about to execute a branch that depends on a symbolic value.
- Each leaf represents a final state of the program.

For any input value, there is exactly one associated path going from the root to a leaf, that represents the execution of the program on this input.

Let us call this tree the *execution tree*. Symbolic execution consists of exploring this execution tree.

As an illustration, Figure 1.2 represents the execution tree of our *JPIG* image reader program.

1.4 Symbolic execution in practice

Now that we have introduced symbolic execution and the base concepts associated to it, we will discuss in details how a symbolic execution engine works.

A symbolic execution engine can be seen as the combination of three components [18]. Although the actual implementation of symbolic executors does not necessarily clearly makes a distinction between the three components presented below [18], this separation is useful conceptually to discuss the functioning of symbolic execution and the research work on it.

The three components are :

- An *execution component* that executes – or simulates the execution of – the target program. The execution component is responsible for tracking the operations performed on symbolic values and for producing new symbolic values as necessary.
- A *scheduling component* that applies a strategy for exploring the execution tree and tracks constraints associated to the different execution paths.

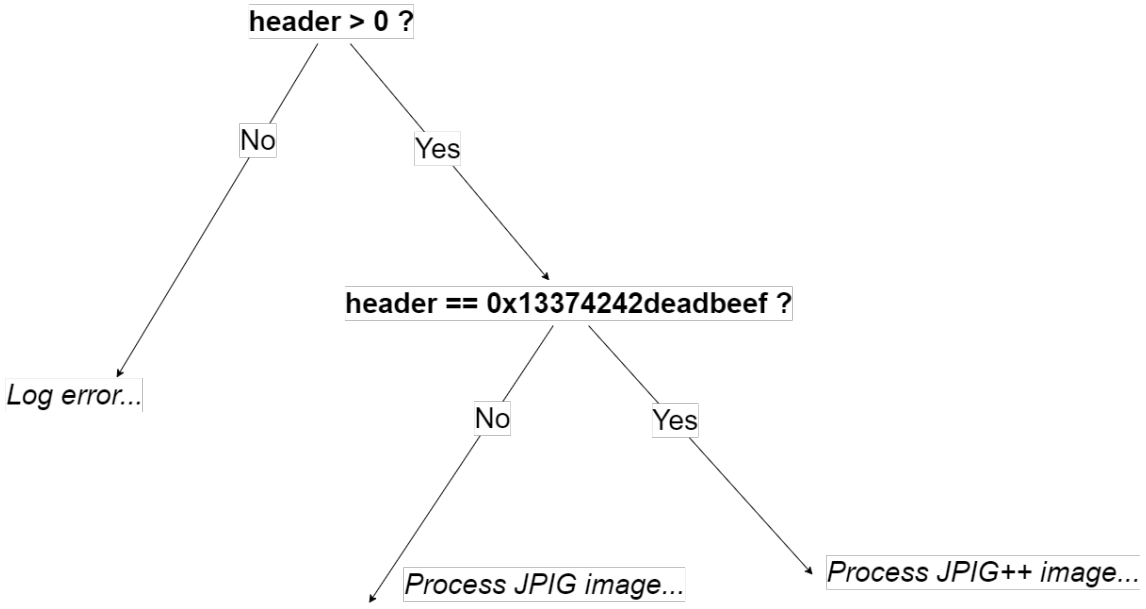

Figure 1.2: JPIG reader execution tree.

- A *constraint solving component* that reasons about path constraints to determine which paths are reachable and generates actual examples of inputs.

The execution component communicates with the scheduling component to transmit information about branch conditions on symbolic values. The scheduling component tells the execution component which part of the execution tree should be executed. Additionally, the scheduling components makes calls to the constraint solving component to solve path constraints in order to determine which paths are reachable and to generate examples of inputs that make the execution reach those paths. Figure 1.3 illustrates the communication between the components.

Most improvements on symbolic execution techniques can be seen as a specific improvement to one of the three components. To increase the efficiency of a symbolic executor, we can either try to execute the program faster, to find better exploration strategies, or to resolve symbolic constraints faster.

In this section, we will discuss each of those components in details.

We will start with the execution component in 1.4.1. As the main interest of SymQEMU is a significant improvement on the execution component, this component will be especially detailed. We will see how computations are performed on concrete and symbolic values, and how execution components are implemented in practice. We will also discuss the problematic of program state saving and restoration.

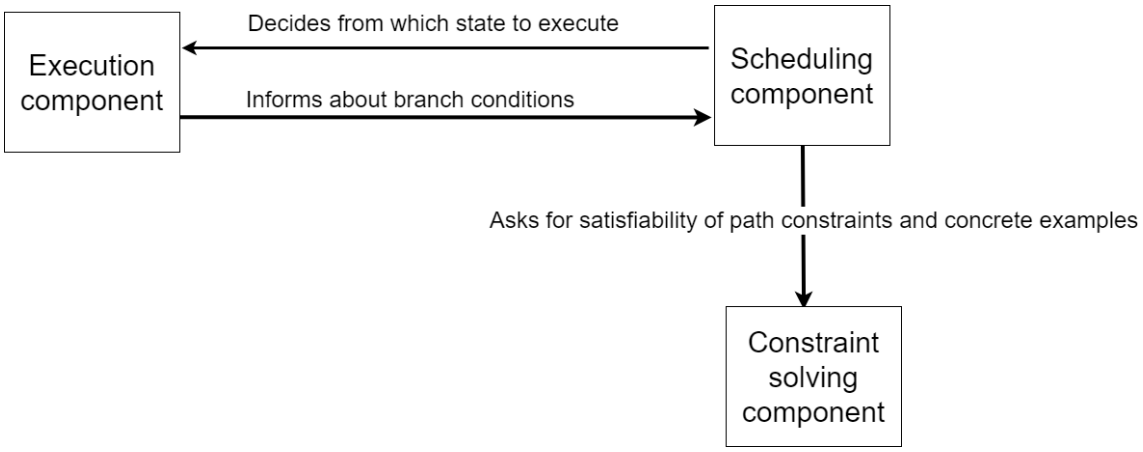

Figure 1.3: Symbolic executor components.

Then we will discuss the scheduling component in 1.4.2. We will see how path constraints are collected along execution forks, and how those constraints are used to generate concrete input values that lead to those paths.

Finally, in 1.4.3, we will discuss the scheduling component. We will explain how the satisfiability of path constraints are determined and how actual examples of inputs are generated.

1.4.1 Execution component

Let us first discuss the execution component.

Given a program state designated by the scheduling component, the job of the execution component is to continue the execution of the program from this state [6].

A program state describes the content of memory, registers, program counter, etc. at a given point in the execution flow. For simplicity, below we use the word "memory" to refer to the whole program state.

To discuss the execution component, we will first explain how it manages operations on symbolic values and how new symbolic values are created. We will then discuss how execution components are actually implemented in symbolic executors. Finally, we will explain the problem of state saving and restoring, which is necessary to be able to execute the target program from an intermediate node of the execution tree.

1.4.1.1 Concrete and symbolic values

In the program state manipulated by the execution component, each memory area is either associated to a concrete or a symbolic value.

As the execution progresses, the program performs operations on its state. Each operation can be seen as a computation that takes as input one or several values stored in memory, and that outputs the results into a memory area o. When an operation is performed, one of the two following situations happens :

- If all the inputs of the operation are concrete, the execution component simply performs the operation normally and stores the result as a concrete value in o.
- If at least one input of the operation is symbolic, the execution component does not perform the operation. Instead, it generates a new symbolic value and stores it in o. The new symbolic value contains references to the input values of the operation, and the description of the operation (e.g., an addition).

We can see that a symbol value will "propagate" into the program memory each time an operation that depends on it is performed.

When the execution reaches a branch condition that depends on a symbolic value, the condition is sent to the scheduling component.

1.4.1.2 Implementation of the execution component

Having seen how the execution component should manipulate concrete and symbolic values, let us discuss now how it is implemented in testing tools.

For the sake of simplicity, the example of symbolic execution that was presented on our *JPIG* reader program was performed directly at the level of the C code. In reality, modern symbolic executors analyze the target program in a lower level representation [18].

Some symbolic executors require the target program to be translated into an *intermediate representation (IR)* [7, 24, 18, 26]. An IR is a language used to describe a program at a lower level than source code but at a higher level than machine code assembly. IR typically have a smaller and simpler instruction set than classical instruction set architectures (e.g., x86). An example of widely used IR is LLVM [15].

Other symbolic executors directly analyze compiled programs at the machine code level [14, 20, 26].

Overall, most symbolic executors fall into one of the three following categories [18]:

IR interpretation. The executor component receives the program translated into an IR

and simulates its execution inside a virtual environment that allows to track the symbolic values [7, 24].

Using an IR makes the implementation of the symbolic executor simpler than if it had to be done on the machine code level. Another advantage is that the symbolic executor can analyze programs written in various programming languages, as long as they can be translated to the IR.

The main downside of this approach is that the emulation of the target program is slow.

Runtime instrumentation. The executor component receives the program in machine code. It executes it directly while performing dynamic run time instrumentation to track the operations performed on symbolic values. The symbolic executor holds a representation of the target program memory for storing the symbolic values, and updates it according to the information received from the instrumentation [14, 20, 26].

This approach is fast, as the target program is executed directly and not interpreted. The downside is that the symbolic executor implementation requires a lot of work, as classical architectures like x86 or arm are more complex than an IR. Additionally, this work has to be repeated for each architecture that the symbolic executor must support.

IR instrumentation. The symbolic executor receives the program translated into an IR. It instruments the IR code by adding instructions for tracking the operations performed on symbolic values. It then compiles the instrumented program and executes it directly [18, 19].

Like for the runtime instrumentation approach, the symbolic executor holds a representation of the target program memory for storing the symbolic values. As the program runs, the added tracking instructions make the program perform calls to the symbolic executor to signal operations performed on symbolic values, and the symbolic executor updates its representation of the program state accordingly.

This is probably the most promising approach, as it combines the advantages of the two previous ones.

1.4.1.3 Saving and restoring program states

Now that we have seen what the execution component does and how it is implemented, let us discuss a last aspect of this component : saving and restoring program states.

Depending on the exploration strategy applied by the scheduling component, the execution component may need to be able to save and restore program states, typically to be able to explore one branch after a fork in the execution tree, and then go back to this fork and explore the other branch [18].

For an execution component that simulates the execution of the program in a virtual environment, this task is quite simple as the execution component can easily save and restore the program state inside the virtual environment [7].

This restoration capability is more complex to implement for an execution component that relies on the actual execution of an instrumented executable, as a state restoration requires to recreate a process and restore its environment with respect to the OS (e.g., the PID of the process, the open file descriptors, etc.) [9].

1.4.2 Scheduling component

Having covered the execution component, we will now discuss the scheduling component of symbolic executors.

The scheduling component is responsible for deciding which area of the execution tree to explore, to collect constraints associated to the execution paths and to generate actual examples of inputs that lead the execution of the target program to those paths.

To discuss the scheduling component, we will describe how it associates path constraints to the explored nodes of the execution tree and how those path constraints are used to generate examples of inputs. We will also briefly discuss the need for a strategy for exploring the execution tree.

1.4.2.1 Forks and path constraints collection

For any explored node in the execution tree, the scheduling component stores an associated list π of conditions that the input must satisfy for the program to reach this point of the tree. Those conditions are called *path constraints*.

The list π associated to the root of the execution tree is empty, as any input value makes the execution go through the root of the execution tree.

When the execution reaches a branch statement whose condition depends on a symbolic value, the scheduling component can use the symbolic value to translate this condition into a condition b that depends on the input value [17].

In this situation, the scheduling component checks if $\pi \wedge b$ is satisfiable, i.e., if there exists an input such that all the accumulated constraints and the new constraint b are satisfied. Similarly, it checks if $\pi \wedge \neg b$ is satisfiable. Those checks are performed by doing a call to the constraint solving component.

Let us first consider the case where only one of the two conditions is satisfiable. Without loss of generality, let us assume that $\pi \wedge b$ is satisfiable while $\pi \wedge \neg b$ is not satisfiable. Then we have $\pi \implies b$. In this situation, the execution can only continue down the branch

associated to b and there is no need to update π, as it already implies b. As a consequence, there is no fork in the execution tree.

Now consider the case where both $\pi \wedge b$ and $\pi \wedge \neg b$ are satisfiable. In this situation we have discovered a fork in the execution tree as, depending on the input, the execution of the program could continue to either direction. The scheduling component associates the node corresponding to the case where the condition is true to a new set of constraints $\pi' := \pi \wedge b$. Similarly, the scheduling component associates the node corresponding to the case where the condition is false to a new set of constraints $\pi'' := \pi \wedge \neg b$.

Note that, as the execution component acts either at the IR or machine code level, it does not deal with high level control flow statements like *while* loops or *switch* statements. At the IR or machine code level, those statements are all translated to conditional branch instructions. As a consequence, all forks in the execution tree are associated to a conditional branch instruction.

1.4.2.2 Input generation

Thanks to the path constraint collected, for any explored node of the execution tree the symbolic engine can compute an actual example of an input value that leads the execution to this node.

It does so by performing a call to the constraint solving component, in order to get an input value that satisfies the set π of path constraints associated the requested node [6].

1.4.2.3 Exploration strategy

Another aspect of the scheduling component is that it needs to decide in which order the nodes of the execution tree should be explored.

Ideally, we would like the symbolic engine to explore all the execution paths of the target program. However, such an exhaustive exploration ot the target program is often impossible in practice because of the path explosion problem, discussed in 1.5.1.

To address this issue, symbolic engines use various heuristics in order to explore in priority execution paths that seem interesting [7].

1.4.3 Constraint solving component

We have discussed the execution and the scheduling components of a symbolic execution engine. Let us now discuss the final component : the constraints solving component.

We have seen that a symbolic engine needs to be able to know if a set π of path constraints

is satisfiable, and to produce actual examples of input values that satisfy π.

In mathematics, this problem is known as *Satisfiability Modulo Theories (SMT)*, and an *SMT solver* is a tool that tries to perform this task. Although the SMT problem is generally hard, modern SMT solvers can often solve a given set π of constraints in an acceptable time [17].

The constraint solving component uses an SMT solver to resolve path constraints request. It may have additional features to make the resolutions faster, like pre-processing the constraints to simplify them, or caching mechanisms [18].

1.5 Limits of symbolic execution

Now that we have described in details how symbolic execution works and what it is used for, in this section we will discuss some important problems that arise when using it to analyze real world software.

Although symbolic execution could theoretically explore all the execution paths of a target program, it is often not possible in practice because of the so called path explosion problem. We discuss this in 1.5.1

Another important difficulty for symbolic execution is the interaction between the target program and its environment. We discuss this in 1.5.2.

1.5.1 Path explosion problem

The *path explosion problem* is the main limitation of symbolic execution.

To illustrate it, consider this slightly expanded version of our *JPIG* reader program :

Code Source : JPIGReader.c

```
1  void main(){
2
3      int64_t header = 0;
4
5      fgets((char*) &header, 8, stdin);
6
7      if (header > 0) {
8          if (header == 0x13374242deadbeef){
9              /*
10                 process JPIG++ image
11             */
12         } else {
13
14             if (getchar() == 0x0) {
```

```
15                  /* do something */
16              } else {
17                  /* do something else */
18              }
19
20              if (getchar() == 0x0) {
21                  /* do something */
22              } else {
23                  /* do something else */
24              }
25
26              if (getchar() == 0x0) {
27                  /* do something */
28              } else {
29                  /* do something else */
30              }
31              /*
32                  ...
33              */
34          }
35      } else {
36          /*
37              log error
38          */
39      }
40  }
```

Starting from line 14, the program will repeatedly read a byte from *stdin* and perform a different task depending on whether this byte is equal to zero or not.

When reaching the first *if* statement, the symbolic engine will fork its state to explore both execution paths. Then, in each execution path, the second *if* statement will be reached, resulting in new forks and the presence of 4 execution paths to explore. Similarly, the third *if* statement results in 8 execution paths and so on. Figure 1.4 represents the execution tree of this new version of the *JPIG* image reader.

We see that the number of execution paths tends to increase exponentially with respect to the size of the program. As a consequence, the number of execution paths in real world programs if often very large. Given the fact that the exploration of each new path is slow because it requires to perform costly calls to the SMT solver, it is often infeasible for a symbolic engine to perform an exhaustive exploration [6].

1.5.2 Interaction with the environment

Another difficulty for symbolic execution is the interaction between the target program and its environment, when symbolic values are involved in those interactions [8, 7]. Typically,

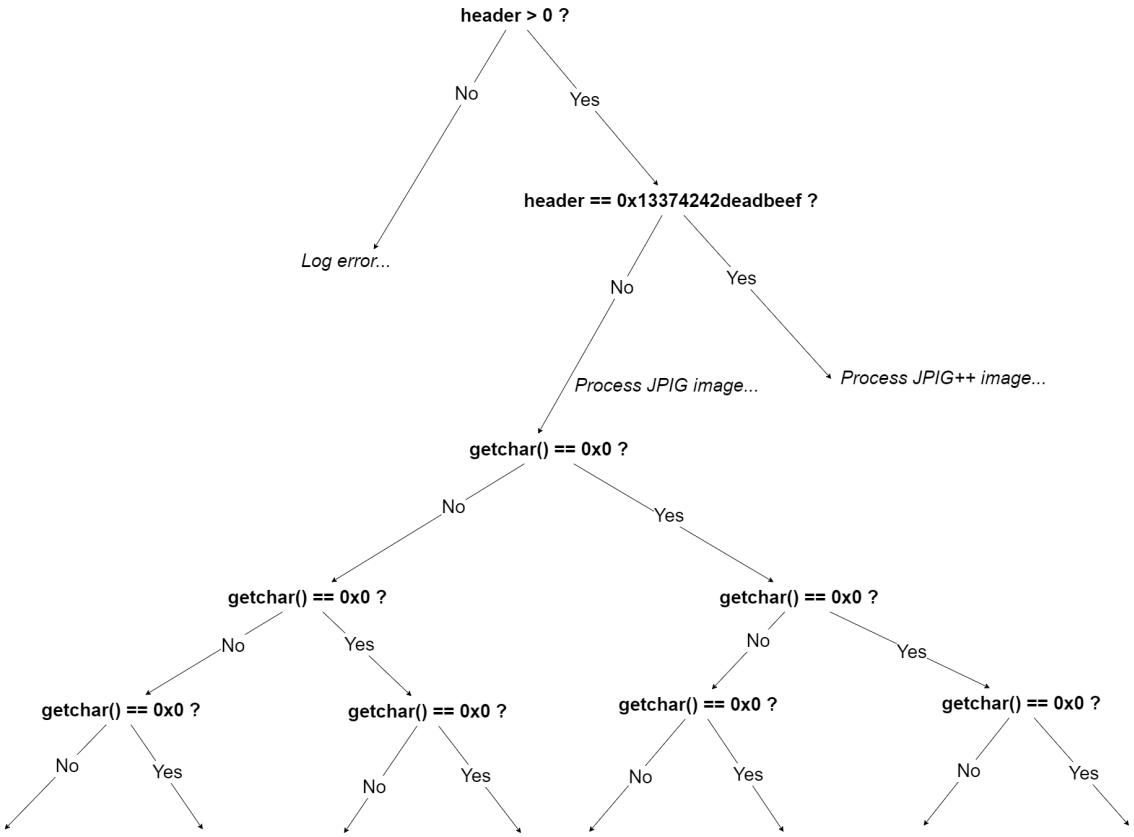

Figure 1.4: Execution tree of JPIG reader with path explosion.

calls to external libraries functions or system calls are problematic when performed with symbolic arguments.

Those interactions with the program environment can be seen as calls to functions that are external to the target program. Some solutions that symbolic executors use to perform those calls to external functions are :

Symbolic value concretization. The symbolic value is concretized (i.e., a call to the SMT solver is performed to generate a concrete value) and the call to the external environment is done with this concrete value as argument [7].

The disadvantage of this approach is that the external code is not executed symbolically, which stops the propagation of symbolic values [8]. Additionally, concretization requires an additional costly call to the SMT solver.

Environment modeling. The calls to external functions are redirected to a *model* of the program environment, that is able to simulate the effect of the external functions on symbolic values [24, 7]. For instance, if the target program calls the function *strcmp* from the C standard library, the symbolic executor could instead call a substitute function that simulates the behavior of *strcmp* on symbolic values.

The disadvantage of this approach is that substitute functions must be manually written for each external function that we want the symbolic executor to support.

Modeling the side effects of these functions on the system is also complicated. For example, if the target program performs a system call to write a symbolic value into a file, the environment model should be able to provide back this symbolic value if the target program subsequently reads the file.

Symbolic execution of the environment. The external function is executed symbolically [8].

The disadvantage of this solution is that it is prone to path explosion, as the code of external libraries and operating systems is often very large and complex.

1.6 Concolic execution

We will now present a special flavor of symbolic execution named *concolic execution*. We will see that concolic execution has several advantages over classical symbolic execution. Concolic execution particularly suits an execution engine that consists of instrumenting the program under test as described in 1.4.1.2.

The word *concolic* comes from a combination of the words “concrete” and “symbolic”.

Note that the scientific literature is not entirely consistent on the definition of the word concolic. The definition that we propose in this thesis is the one that, to the best of our

knowledge, is the most consensual, at least among the researchers working on techniques for finding low level vulnerabilities in real world software.

However, some authors use the expression "concolic execution" to designate what we have defined in the preceding section as "symbolic execution". In 2.1 we will present several symbolic and concolic executors, and we will discuss the terminology used by theirs authors.

In this section, we will first propose our definition of concolic execution in 1.6.1. We will then develop it further in 1.6.2 and in 1.6.3, by describing the execution component and the scheduling component of a concolic executor, in order to understand in which aspects it is different from classical symbolic execution.

Then, in 1.6.4, we will also explain how a concolic executor can be iteratively rerun on its outputs.

Finally, in 1.6.5 we will discuss the advantages of concolic execution compared to classical symbolic execution.

1.6.1 Definition

In this thesis, we use the following definition for concolic execution :

Concolic execution is a technique that consists of running the target program both on a symbolic and concrete version of the input, at the same time. The target program is executed symbolically, but it follows the execution path dictated by the concrete version of the input. Using the path constraints accumulated during the execution, concolic execution allows to discover input values that would make the execution of the program take new paths [17, 6, 21].

Concolic execution can be seen as a special case of symbolic execution, where the execution component and the scheduling component have the specific behaviors described in 1.6.2 and 1.6.3.

1.6.2 Execution component of a concolic executor

To clarify this definition, let us describe how the execution component of a concolic executor is different from the execution component of a symbolic executor.

The execution component of a concolic executor has the same behavior as the execution component of a symbolic executor, except for the following difference: With concolic execution, all symbolic values also have an associated concrete value. In other words, when performing concolic execution, any memory area of a program state is either associated to :

- A concrete value, or

- both a concrete and a symbolic value.

The concrete value associated to the initial symbolic value, i.e., the program input, needs to be provided to the concolic executor, like for a fuzzer.

Operations that take as input some memory area that is only associated to a concrete value are performed normally, as described in 1.4.1.1 for symbolic execution.

Let us decribe what happens when the execution component executes an operation that takes as input a memory area associated to both a concrete and a symbolic value and that outputs its result into a memory area o. In this situation, the execution component will :

- Perform the operation using the concrete version of the inputs and store the result into the concrete version of o.
- Build a new symbolic value, like a symbolic executor would have done, using the symbolic version of the inputs. The new symbolic value is then stored into the symbolic version of the memory area o.

In other words, the executor component performs both a normal and a symbolic execution at the same time. The normal execution is dictated by the concrete value provided for the program input.

1.6.3 Scheduling component of a concolic executor

We will now describe how the scheduling component of a concolic executor works. To discuss this, we will cover the same points that were discussed in the section related to symbolic execution, i.e., the exploration strategy, the collection of path constraints and the generation of new inputs.

1.6.3.1 Exploration strategy

The exploration strategy of a concolic executor is very simple : it performs one traversal of the execution tree, from the root to a leaf, while following the path dictated by the concrete version of the input value.

1.6.3.2 Path constraints collection

Like for symbolic execution, the concolic executor stores a set π of path constraints, that is initially empty.

When the program reaches a branch condition that depends on a memory area associated to a symbolic value, the concolic executor executes the branch condition like in a normal

execution, using the concrete values, to determine which of the two possible paths is taken. Let's call the path that is taken p.

Additionally, the concolic executor uses the symbolic version of the variables to translate the condition into a constraint b on the input of the program. b is the condition that the input of the program must satisfy for the program to take the path p when reaching this branch condition. The concolic executor updates π to $\pi \wedge b$ and continues the execution down the path p.

1.6.3.3 Generation of new inputs

For a given concolic execution, let us call b_1 the constraint generated when reaching the first branch condition, b_2 the constraint generated when reaching the second branch condition, etc. Then at the end of the concolic execution we have $\pi = b_1 \wedge b_2 \wedge \cdots \wedge b_n$.

The concolic executor generates new input values using the following method :

- It first requests the SMT solver to give a concrete example of input value i_1 that satisfies $\neg b_1$. Now i_1 is an input value that would make the program execution go through the other path than the one taken during the concolic execution, on the first branch condition.
- Then it requests the SMT solver to give a value i_2 satisfying $b_1 \wedge \neg b_2$. When executed on i_2, the target program would follow the same path as during the concolic execution, until the second branch condition. On the second branch condition, the program executed with input i_2 would take the other path than the one taken during the concolic execution.
- The concolic executor continues generating inputs by making requests for $b_1 \wedge b_2 \wedge \neg b_3$, and so on. More generally, for each path constraint b_m, $1 \leq m \leq n$, the concolic executor generates an input i_m that satisfies $b_1 \wedge b_2 \wedge \cdots \wedge b_{m-1} \wedge \neg b_m$. When executed on i_m, the target program will follow the same path as the concolic execution, until the mth branch condition, where it will take a different path [6, 21].

Figure 1.5 represents the traversal of the execution tree by a concolic execution, along with the collected path constraints.

With symbolic execution we have seen that, if the program reaches a branch condition that depends on a symbolic value, it does not necessarily mean that both paths are reachable, as the constraints associated to one of the path may be unsatisfiable. Equivalently, with concolic execution some of the conditions generated with the method described above will be unsatisfiable.

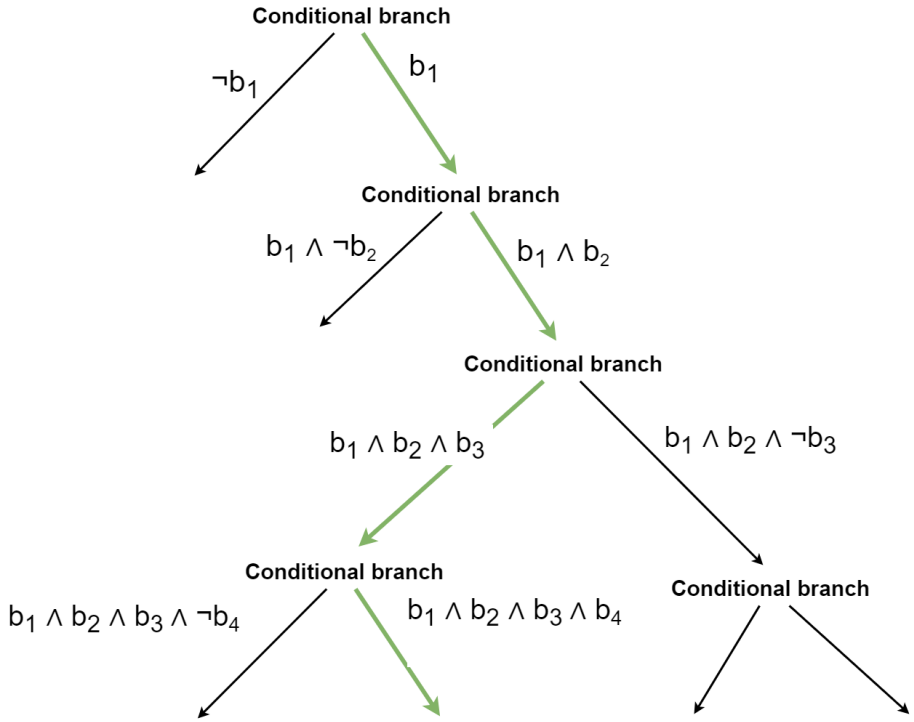

Figure 1.5: Concolic exploration of the execution tree.

1.6.4 Iterative testing on new inputs

Now that we have seen how concolic execution works, we will discuss how a concolic executor can be iteratively rerun on its outputs, to explore the execution tree.

After generating the new inputs $i_1 \dots i_n$, the concolic executor could be iteratively rerun on each of these inputs, in order to discover some new inputs covering new paths of the target program. By doing it systematically, the concolic executor could theoretically discover an exhaustive list of inputs that collectively cover all the possible execution paths of the target program [14, 21].

In this respect, this strategy of systematically rerunning the concolic executor on each generated input is equivalent to a depth-first exploration of the execution tree [21].

Note that, as a consequence, an attempt to cover all possible execution paths of a real world software using concolic execution would face the same path explosion problem as classical symbolic execution[14].

1.6.5 Advantages of concolic execution

Now that we have seen how concolic execution is different from symbolic execution, we will present the motivations behind this approach.

Compared to classical symbolic execution, concolic execution requires fewer calls to the SMT solver, as some needed concrete values can be obtained from the concrete side of the execution directly. This is discussed in 1.6.5.1 and 1.6.5.2.

Additionally, we concolic execution there is no need for state restoration as the target program is executed from start to end at each run. This is discussed in 1.6.5.3

1.6.5.1 Lower number of queries to the SMT solver

We have seen that with classical symbolic execution, when a branch condition that depends on a symbolic value is reached, the symbolic executor needs to check if the path constraints associated with both paths are satisfiable or not. As a consequence, two calls to the SMT solver must be performed.

With concolic execution, the executor uses the concrete version of the values to know which path to follow. It only performs one call to the SMT solver instead of two, to know if the other path is reachable (and to generate an actual input example if the answer is yes) [6].

1.6.5.2 Free concretization

We have seen that in some situations, symbolic executors need to perform calls to the SMT solver in order to concretize a symbolic value before giving it as argument to some external function.

With concolic execution this concretization does not consume any resource, as the concolic executor already has a concrete value associated to each symbolic value at any time during the execution [6].

1.6.5.3 No state restoration

Depending on the execution tree exploration strategy, symbolic executors may need to be able so save and restore program states.

As concolic execution performs one execution at a time, where each execution goes from the start of the program to its end, there is no need for state restoration [18].

This makes concolic execution especially suitable for an execution component that actually executes an instrumented version of the compiled program.

1.7 Hybrid fuzzing

We have seen that fuzzing is an efficient technique for testing software automatically, but that it is difficult for a fuzzer to explore execution paths behind tight branch conditions.

On the other hand, symbolic and concolic testing can discover those hard to reach paths in a systematic way, but this technique is much slower, which makes real world software testing often impossible because of the path explosion problem.

Hybrid fuzzing is an approach where a fuzzer and a concolic executor are used together to test the target program, in a way that aims to combine the strengths of each tool [17].

Typically, the fuzzer will use new input values discovered by the concolic executor and rapidly try to further explore the execution tree by applying random mutations on them. Similarly, interesting inputs discovered by the fuzzer will be provided to the concolic executor, in order to try to discover new paths that would have been difficult to reach for the fuzzer.

This approach has given promising results. Hybrid fuzzing allowed to discover vulnerabilities in real world software, that neither fuzzing alone nor symbolic execution alone were able to find [25, 26, 18].

The setup of hybrid fuzzing is in itself a whole subject of research [6]. Some of this work is related to finding a good strategy for deciding how much resource to allocate to the fuzzer

and how much resource to allocate the concolic executor during testing. Another subject of research is the problem of defining criteria for selecting interesting inputs discovered by one tool and giving them to the other.

Chapter 2

State of the art

In this chapter, we present an overview of existing tools for automatic discovery of memory errors. We will especially concentrate on tools that use symbolic execution.

In 2.1 we present different existing tools. Our goal is not only to show which tools exist but also to show their relationship to each other and which novel ideas they bring.

In 2.2, we present a summary of tools that use symbolic execution for finding memory errors.

2.1 Existing automated testing tools

As this thesis focuses on techniques for finding memory bugs, we will only cover symbolic tools designed to test programs written in languages prone to memory bugs, and to discover such bugs. Symbolic execution can however be used for a lot of other purposes.

When not stated otherwise, the authors of the tools presented below use the same terminology as the definitions of symbolic execution, concolic execution and hybrid fuzzing that we gave in Chapter 1. When the authors of a tool use a different terminology, we will mention it.

In 2.1.1 we will first present AFL, which is the most popular fuzzer.

We will then present the widely used symbolic execution tools KLEE, S2E and Angr, in 2.1.2, 2.1.3 and 2.1.4.

For a historical perspective, in 2.1.5 and 2.1.6 we will cover CUTE and SAGE whose respective authors were among the first to propose the idea of concolic execution, and the hybrid fuzzing tool Driller in 2.1.7, whose authors present an interesting discussion about hybrid fuzzing.

Finally, in 2.1.8, 2.1.9 2.1.10 and 2.1.11, we will present the more recent concolic executors

Triton, QSYM, SymCC and SymQEMU.

2.1.1 AFL

Before presenting tools based on symbolic execution, let us briefly discuss AFL, the most popular fuzzer.

American Fuzzy Lop (AFL) [1] is a fuzzer created by Google. AFL itself is not updated anymore, but AFL++ [13], a popular fork created by the AFL community, is actively maintained.

AFL and its derivatives are actively used by researchers and companies to test real world software.

AFL can perform coverage guided fuzzing, by instrumenting the target program and monitoring the execution path it takes [13].

2.1.2 KLEE

KLEE [7] is a symbolic execution tool presented in 2008. It is the successor of EXE, an older symbolic execution tool created by the same authors. KLEE is still actively maintained.

KLEE performs symbolic execution on programs in the LLVM IR, by simulating the execution of the LLVM bitcode.

It is typically used to test programs written in C or C++, whose source code is available and that are then translated to LLVM.

KLEE allows to symbolize the program arguments or the content of a file red by the program. It is also possible for the program itself to require KLEE to symbolize values.

KLEE has a model of the program environment and intercepts some calls to external functions to simulate their effect on symbolic values. For the other calls to external code, the symbolic arguments are concretized.

KLEE allows to use various strategies for the exploration of the execution tree, and it is capable of saving and restoring program states.

While exploring the execution graphe, KLEE is capable of automatically detecting several categories of memory bugs.

2.1.3 S2E

S2E [8] is a symbolic execution framework presented in 2011, that allows to perform symbolic execution on a whole system. It is still actively maintained.

S2E allows to perform symbolic execution inside a virtual machine that runs a complete modern operating system, like a GNU/Linux distribution or Windows. It is built on top of QEMU and KLEE. S2E dynamically translates TCG, the IR used by QEMU, to LLVM, and uses KLEE to perform symbolic execution on it.

The software to execute symbolically is provided to S2E as compiled machine code.

2.1.4 Angr

Angr [24] is a symbolic execution framework written in Python. It was created in 2013 and is still actively maintained.

The software to analyze is provided to Angr as a compiled executable, like ELF or PE binaries, there is no need to have the source code available.

Angr provides a flexible Python API for managing the symbolic execution, that allows to freely manipulate program states, symbolic values and path constraints, and to perform symbolic execution on them using various exploration techniques. Because Angr allows to freely control and customize the symbolic execution process, it can easily be used to perform concolic execution.

Internally, Angr translates the machine code assembly to VEX, which is an IR originally used by the Valgrind profiling tool, and simulates the execution of the VEX code. As an optimization, Angr uses QEMU for executing code that does not depend on symbolic values.

Angr includes a number of "simprocedures" for various functions of the standard C library and system calls. By default, when the program performs a call to an external function for which a simprocedure exists, the simprocedure will be called instead and simulate the behavior of the orginial function, while propagating the symbolic values.

2.1.5 CUTE

CUTE [22] is a concolic executor that was presented in 2005. It is not maintained anymore but it is interesting in a historical point of view, as its authors described several concepts that happened to become important in more recent years.

CUTE is one of the first testing tool that executes the target program both on a symbolic and concrete version of the input at the same time and. The authors of CUTE call this approach "concolic testing" and, to the best of our knowledge, they were the first to use this expression in the field of automated software testing.

To do concolic execution, CUTE adds instrumentation to the C code in order to include support of symbolic execution directly into the target program. The program is then compiled and executed directly. Interestingly, this is a similar approach that allowed more

modern concolic executors like QSYM to achieve significant progress in symbolic execution performance, many years later [18].

Additionally, already in 2007, the authors of CUTE proposed the idea of combining a concolic executor with a fuzzer and call this approach hybrid fuzzing [16].

2.1.6 SAGE

SAGE [14] is a concolic execution tool created by Microsoft. It is not publicly available, but its authors published a paper describing it in 2012.

Instead of using the expression "concolic execution", the authors of SAGE call the technique that their tool implements "whitebox fuzzing". They give the following definition of whitebox fuzzing:

"Starting with a well-formed input, whitebox fuzzing consists of symbolically executing the program under test dynamically, gathering constraints on inputs from conditional branches encountered along the execution. The collected constraints are then systematically negated and solved with a constraint solver, whose solutions are mapped to new inputs that exercise different program execution paths." [14]

As we can see, the process they describe is exactly the approach that we have defined as "concolic execution".

2.1.7 Driller

Driller [25] is a hybrid fuzzing tool that was presented in 2016 by the authors of Angr. It is not maintained anymore.

Driller performs hybrid fuzzing by using AFL for the fuzzing and Angr for the concolic execution. The principle of the functioning of Driller is the following :

1. Driller runs AFL on the target program.
2. When, according to some criteria, the fuzzer stops making progress, some inputs discovered by the fuzzer are selected to be given to the concolic executor.
3. Angr is used to perform concolic execution on those inputs.
4. This process is started again by running the fuzzer on the inputs generated by the concolic execution.

The Driller paper presents an interesting discussion about the fact that the execution tree of a tested program can be seen as a set of connected "compartments". A compartment is

a subset of the execution tree that a fuzzer can easily explore, given an original input that makes the program reach a node of the compartment. Compartments are separated by tight branch conditions, that require usage of symbolic execution to be passed.

The authors of Driller use the expression "concolic execution" to refer to what we have defined as symbolic execution. They use the expression "whitebox testing" to designate fuzzing augmented by inputs discovered with symbolic execution, i.e., what we have defined as hybrid fuzzing.

2.1.8 Triton

Triton [20] is a concolic execution framework that was presented in 2015 and is still actively maintained.

Triton performs symbolic execution on a compiled executable by running it directly and performing dynamic runtime instrumentation using Intel PIN to track the operations performed, and update its symbolic model of the program state accordingly.

It provides a python API for controlling the concolic execution.

Triton can automatically detect some kind of memory bugs like buffer overflows or format string vulnerabilites. It is also able to save and restore process states.

2.1.9 QSYM

QSYM [26] is a concolic executor that was presented in 2018. It is not maintained anymore but some of its components have been incorporated in SymCC and SymQEMU, presented in 2.1.10 and 2.1.11.

QSYM allows to test x86 executables, using an approach similar to Triton. It uses Intel PIN to perform dynamic runtime instrumentation on the target program, in order to track operations performed on the program state, and update its symbolic representation of the program accordingly.

This approach that consists of directly executing the target program, instead of simulating the execution of an IR, allows QSYM to be orders of magnitudes faster than predecessor tools like KLEE or Angr. According to the authors of QSYM, the research community used to wrongly believe that the main bottleneck for symbolic execution was the cost of path constraint resolutions with the SMT solver, while actually one of the main limitation was the fact that most symbolic executors perform IR interpretation, which is slow.

The authors of QSYM used it to perform hybrid fuzzing with AFL in a simple setup, that consists of running the fuzzer and the concolic executor in parallel, while sharing all the inputs discovered by each tool in a common pool.

This simple setup allowed them to discover several previously unknown security bugs in popular real world programs that were already heavily tested by state-of-the art tools.

2.1.10 SymCC

SymCC [18] is a concolic executor that was presented in 2019 and is still actively maintained.

One of the motivations behind SymCC is the following : while the approach proposed by QSYM or Triton is very efficient, its downside is that adapting it to new processor architectures other than x86 would require a lot of work, as the instrumentation is performed at the machine code level. A better approach would be to perform static instrumentation at an IR level, and then compiling and executing the resulting program.

SymCC performs instrumentation in the LLVM IR. It modifies the behavior of the Clang compiler, such that the compiler adds instructions into the compiled program for tracking operations performed on symbolic values and performing calls to a "symbolic backend" that tracks constraints and generates new inputs. SymCC reuses the symbolic backend of QSYM.

In other words, SymCC "compiles symbolic execution capabilities right into the binary" [18]. A program compiled with SymCC will have exactly the same behavior as if it was compiled normally, except that it incorporates the following new feature : each time the program reaches a branch in the execution flow that depends on the input value, the program outputs in a dedicated directory an example of input that would have made it take the other path.

Thus, SymCC can easily be used to test any program whose source code is available and that is written in a language for which an LLVM front-end exists.

The authors of SymCC used it to perform hybrid fuzzing with a similar setup than the one used for QSYM. They discovered two critical previously unknown vulnerabilities in the already heavily tested OpenJPEG project.

2.1.11 SymQEMU

Finally, let us present SymQEMU.

SymQEMU [19] is a concolic executor that was presented in 2021 by the authors of SymCC.

SymQEMU can be seen as a symmetrical idea to SymCC : while SymCC instruments an IR code that has been obtained from the source code of the program, SymQEMU instruments an IR code that has been obtained from a compiled executable. The instrumented IR is then translated back to machine code and executed directly.

SymQEMU is built on the top of QEMU and it instruments TCG, which is the IR used by QEMU. SymQEMU reuses components of SymCC.

The design of SymQEMU allows it to easily test programs compiled for the various architectures supported by QEMU. If an architecture is supported by QEMU, low effort is required to make SymQEMU support it.

2.2 Tools summary

Table 2.1 presents a summary of tools for automated finding of memory errors that use symbolic execution.

The column *tested program* indicates the form in which the program under test must be provided to the tool. We consider a tool to be still maintained if its last update (last commit on GitHub) is not older than 6 months at the time of writing.

Note that as SAGE is note publicly available, little information is known about it.

Name	**Year**	**Technique**	**Tested program**	**Maintained**
CUTE	2005	Concolic execution	C source code	No
SAGE	2007	Concolic execution	Unknown	Unknown
KLEE	2008	Symbolic execution	LLVM	Yes
S2E	2011	Symbolic execution	Compiled binary	Yes
Angr	2013	Symbolic execution	Compiled binary	Yes
Triton	2015	Concolic execution	Compiled binary	Yes
Driller	2016	Hybrid fuzzing	Compiled binary	No
QSYM	2018	Concolic execution	Compiled binary	No
SymCC	2019	Concolic execution	C/C++ source code	Yes
SymQEMU	2021	Concolic execution	Compiled binary	Yes

Table 2.1: Testing tools for find memory erros that use symbolic execution.

Chapter 3

QEMU

In this chapter, we are going to explain and describe several internal aspects of QEMU.

The motivation for this is that, in order to understand how SymQEMU modifies QEMU, we need first to understand in details how the relevant parts of QEMU work. In particular, we need to understand the code that performs the translation from guest architecture instructions to TCG ops (the QEMU IR), because this is during this translation that SymQEMU puts its instrumentation.

Unfortunately, the source code of QEMU is poorly documented and few resources are available on the web to help understand it. A precious source of knowledge about some internal aspects of QEMU, although a bit outdated, is a series of blog articles by Stephane Duverger [11]. The source of the information that we present in this chapter is a mix of those blog articles, some content of the documentation of QEMU [3, 4] and, mainly, our own reverse-engineering of the source code.

In Section 3.1, we will present some definitions and conventions that will be used in this chapter.

Section 3.2 presents a high level view of the code that manages and executes the translated instructions. We will see that QEMU generates so-called *translation blocks* which are blocks of translated instructions, and that those translation blocks are stored in a caching system.

Then in Section 3.3, we will zoom on the QEMU code that performs the translation from guest to host architecture. In particular, we will show the code involved in the translation of a block of guest instructions into TCG code.

In Section 3.4, we will discuss TCG in details. We will see that TCG provides a set of TCG instructions whose operands are TCG variables. We will present the *TCG API* which is the set of functions that allow to programmatically generate TCG code.

Finally, in Section 3.5, we will describe how the TCG backend internally generates and

manages TCG variables.

3.1 QEMU introduction

QEMU is an emulator that allows to execute programs compiled for a given architecture on a computer whose CPU uses a different architecture. It is written in C.

In this section, we will first give an overview of the functioning of QEMU and the associated terminology. Then, we will give some explanations about the conventions that we follow when presenting QEMU code.

3.1.1 Overview and definitions

We call *target program* the program that QEMU executes. Furthermore, we call *guest architecture* the architecture for which the target program was compiled, and *host architecture* the architecture of the computer that runs QEMU. More generally, we use the words *guest* and *host* to distinguish between what relates to the computer QEMU is running on and what relates to the simulated environment inside which the target program runs.

QEMU uses an intermediate representation named *TCG ops* (for TCG operations). TCG stands for "tiny code generator". When executing the target program, QEMU first translates the guest architecture instructions to TCG ops, then translates the TCG ops to host architecture instructions, and finally executes those instructions.

This translation is performed *just in time*: while the execution progresses, QEMU repetitively fetches a small number of guest instructions, translates them, executes them, and starts over.

The series of instructions that QEMU translates and executes during one step of this process is called a *translation block* (sometimes we also simply call it a block of instructions). We will see that the structure that QEMU uses to represent a translation block holds a pointer to a block of guest instructions, and a pointer to a corresponding block of translated host instructions.

During the execution, QEMU maintains a representation of the guest CPU registers, whose values are modified with respect to the target program instructions.

When discussing QEMU, it is important to have a clear understanding of when a particular code is executed. We call *translation time* the moment when guest instructions are translated to TCG ops or when TCG ops are translated to host instructions. We call *run time* the moment where the translated instructions are executed.

QEMU supports a large number of guest and host architectures. The specific guest and host architectures used are determined at compilation time, the host architecture is the tar-

get architecture of the compiler, and the guest architecture is chosen when configuring the project. QEMU contains alternative implementations of several functions and structures for the different architectures that is supports. The appropriate function and structure implementations are selected at compilation time, according to which guest and host architecture the project was configured for.

3.1.2 User mode and system mode

QEMU has a *user mode* and a *system mode*.

The user mode allows to run an executable that is intended to be executed on the operating system of the host. In user mode, QEMU simply executes the translated program code and forwards the system calls to the host OS.

In system mode, QEMU simulates a whole computer. System mode is used for instance to execute a device firmware.

3.1.3 Conventions for presenting QEMU code

In the following sections, we will regularly present some extracts of the QEMU code. As we focus on some specific aspects of QEMU, we will systematically elude all lines of code that are not relevant to what we want to explain.

Beware that, as a consequence, most of the time we will show a very small subset of the content of the QEMU functions and structures. We encourage the interested reader to read the original code in order to see the actual complexity of the QEMU code.

We will stick to the following convention when presenting QEMU code :

- The header of the code block indicates the path and name of the file that contains this code.
- Elision of lines of code is indicated with the comment `/* [...] */`.
- When a line of code has the form `var = foo();`, and we are interested on the side effect of `foo` but not on its return value, we simplify the line to `foo();`.
- Except for those changes, all code presented is the exact actual code of QEMU.
- All comments are original from QEMU (except for the `/* [...] */` comments).

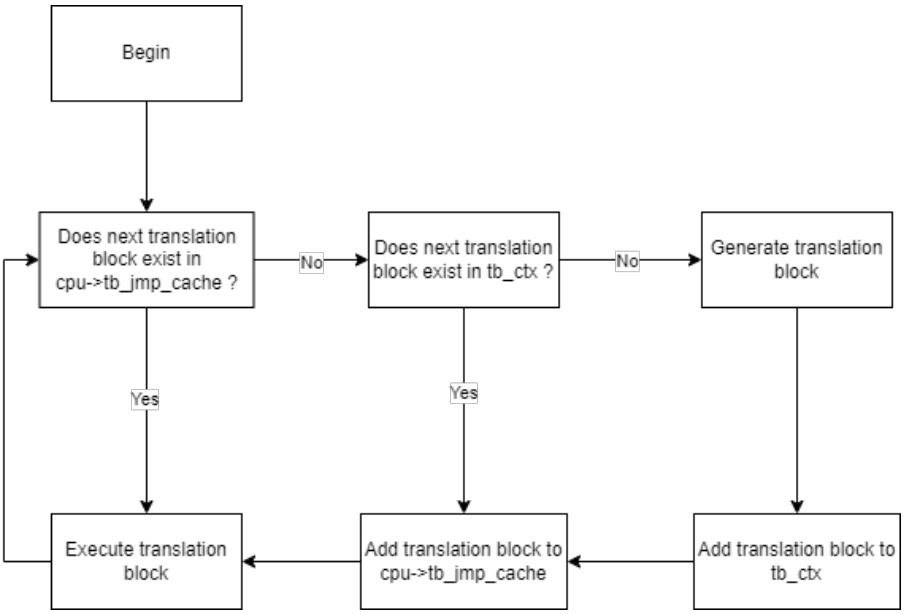

Figure 3.1: Algorithm for generating, caching and executing translation blocks.

3.2 Execution of translation blocks

During the execution of the target program, QEMU repeatedly obtains translation blocks and executes the host instructions associated to them. QEMU uses the state of the simulated guest register, in particular its *PC* register, to determine which block of instructions should be executed next.

When a block of instructions is executed for the first time, QEMU translates it to host instructions and stores the result of this translation into a new translation block. Previously created translation blocks are stored in a caching system, which allows to re-execute them with little overhead.

Figure 3.1 represents the algorithm implemented by QEMU for caching and executing translation blocks. We can see that there are two levels of caching. The first cache is named `tb_ctx` and is common to all simulated guest threads. The second cache is `cpu->tb_jmp_cache` and is specific to each simulated guest thread. As we are going to see, `cpu` is the name commonly used in QEMU to refer to a C structure instance that represents a guest thread.

Figure 3.2 represents the graph of functions that are involved in the creation, caching and execution of translation blocks (functions at the same row are executed from left to right). In this section, we will give an overview of the code of those functions.

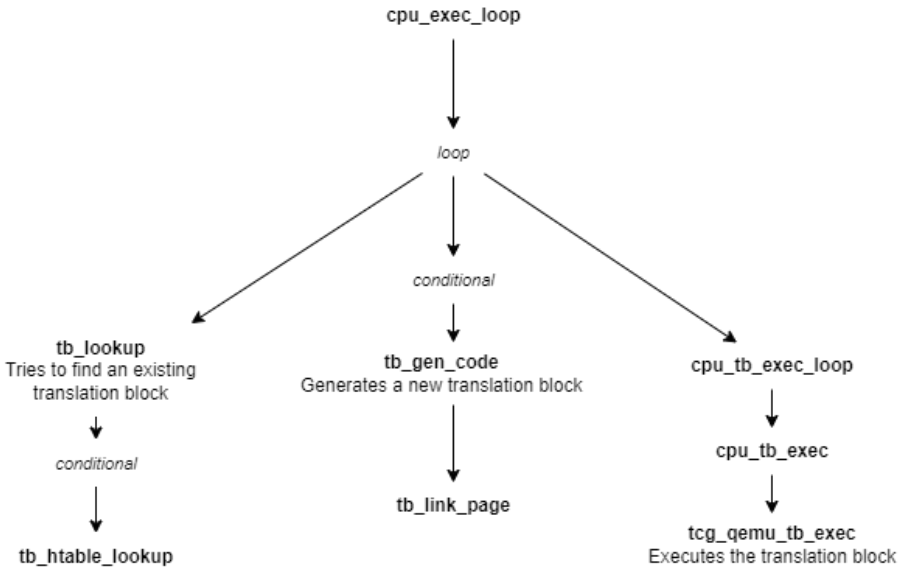

Figure 3.2: Function call graph for the execution of translation blocks.

We will first present several C structures that those functions use. In 3.2.1, we will present the structure that represents a translation block. In 3.2.2, we will present the structure that represents the first cache of generated translation blocks. In 3.2.3, we will present the structure that represents the simulated guest CPU, and in particular its registers. Finally, in 3.2.4, we will present the structure that represents a guest thread.

Having seen those structures, in 3.2.5, 3.2.6, 3.2.7 and 3.2.8 we will then discuss the different functions represented in Figure 3.2.

3.2.1 struct TranslationBlock

A `TranslationBlock` instance represents a block of guest architecture instructions of the target program, that have been translated to host architecture instructions.

In the QEMU source code, variables holding a `TranslationBlock` pointer are typically named `tb`.

`TranslationBlock` is defined in the following way :

Code Source : include/exec/translation-block.h

```
1 struct tb_tc {
2     const void *ptr;    /* pointer to the translated code */
```

```
3     size_t size;
4 };
5
6 struct TranslationBlock {
7
8     /* [...] */
9
10    vaddr pc;
11
12    /* [...] */
13
14    uint16_t size;
15
16    /* [...] */
17
18    struct tb_tc tc;
19
20    /* [...] */
21
22 };
```

- `tb->pc` is the address of the first target program instruction associated to this `TranslationBlock`.
- `tb->size` is the size in bytes of the target program instructions associated to this `TranslationBlock`.
- `tb->tc.ptr` points to a buffer that holds host architecture instructions, which are the result of the translation.
- `tb->tc.size` is the size of this buffer.

3.2.2 struct TBContext

A `TBContext` instance represents a hash table that stores pointers to `TranslationBlock` instances.

There is exactly one instance of `TBContext`, which is a global variable named `tb_ctx`. `tb_ctx` is a cache of already generated translation blocks. When a new translation block is generated, it is added to `tb_ctx`.

Here is an extract of the code defining it:

Code Source : accel/tcg/tb-context.h

```
1 typedef struct TBContext TBContext;
2
```

```
3 struct TBContext {
4
5     struct qht htable;
6
7     /* [...] */
8 };
9
10 extern TBContext tb_ctx;
```

3.2.3 struct CPUArchState

A `CPUArchState` instance represents the state of a guest CPU simulated by QEMU.

An instance of `CPUArchState` stores, among other things, the values of all registers of the simulated guest CPU. As the execution progresses, the register values stored in the `CPUArchState` instance are updated according to the guest instructions whose execution is simulated.

For each guest architecture supported by QEMU, there is a corresponding specific definition of `CPUArchState`.

In the QEMU source code, variables holding a pointer to a `CPUArchState` instance are typically named `env`.

As an illustration, here is an (extremely reduced) extract of the definition of `CPUArchState` for the x86 guest architecture :

Code Source : target/i386/cpu.h

```
1 typedef struct CPUArchState {
2     /* standard registers */
3     target_ulong regs[CPU_NB_REGS];
4     target_ulong eip;
5     target_ulong eflags;
6
7     /* [...] */
8
9     ZMMReg xmm_regs[CPU_NB_REGS == 8 ? 8 : 32] QEMU_ALIGNED(16);
10
11    /* [...] */
12 };
```

- `regs` is an array that stores the values of each standard X68 register (RAX, RBX, RDI, RSI, etc.).
- `eip` stores the value of the EIP register.

- `eflags` stores the state of the x86 status register.
- `xmm_regs` is an array that stores the value of each x86 SSE XMM register.

3.2.4 struct CPUState

A `CPUState` instance mainly contains information that describes a thread state. QEMU creates an instance of `CPUState` for each guest thread, and uses those instances to store information about the state of each thread.

Although the name `CPUState` suggests that it represents a CPU, we find it more natural to think about a `CPUState` instance as representing a simulated guest thread.

In a normal execution (outside of QEMU), each thread has its own vision of the state of the CPU. For instance, writing to a regular register in a thread does not affect the register value in other threads. To simulate this behavior, QEMU associates a separate `CPUArchState` instance to each `CPUState` instance.

In the QEMU source code, variables holding a pointer to a `CPUState` instance are typically named `cpu`. We advise the reader to keep in mind that a `cpu` mostly stores data about a simulated guest thread, and that the actual CPU state data is stored in the associated `CPUArchState` instance.

Here is an (extremely reduced) extract of the definition of `CPUState` :

Code Source : include/hw/core/cpu.h

```
1  struct CPUState {
2
3      /* [...] */
4
5      CPUArchState *env_ptr;
6
7      /* [...] */
8
9      CPUJumpCache *tb_jmp_cache;
10
11     /* [...] */
12
13 };
```

- `env_ptr` is a pointer to the `CPUArchState` instance associated to this thread.
- `tb_jmp_cache` points to a hashtable that stores pointers to `TranslationBlock` instances. When a translation block is executed by the thread, it is added to this cache.

3.2.5 cpu_exec_loop function

Now that we have discussed several structures involved in the management of translation blocks, let us discuss the different functions represented in Figure 3.2.

The `cpu_exec_loop` function is the root function that triggers the translation and execution of target program instructions.

Given a `CPUState` instance, `cpu_exec_loop` will repetitively

- obtain a `TranslationBlock` instance that corresponds to the next instructions that the guest thread should execute, and
- execute the translated host architecture code that is associated to this translation block.

Here is an extract of the definition of `cpu_exec_loop`:

Code Source : accel/tcg/cpu-exec.c

```
1  cpu_exec_loop(CPUState *cpu, SyncClocks *sc)
2  {
3      /* [...] */
4
5      while (!cpu_handle_exception(cpu, &ret)) {
6
7          /* [...] */
8
9          while (!cpu_handle_interrupt(cpu, &last_tb)) {
10             TranslationBlock *tb;
11
12             /* [...] */
13
14             tb = tb_lookup(cpu, pc, cs_base, flags, cflags);
15             if (tb == NULL) {
16
17                 /* [...] */
18
19                 tb = tb_gen_code(cpu, pc, cs_base, flags, cflags);
20
21                 /* [...] */
22
23                 h = tb_jmp_cache_hash_func(pc);
24                 jc = cpu->tb_jmp_cache;
25
26                 /* [...] */
27
28                 jc->array[h].pc = pc;
29                 qatomic_store_release(&jc->array[h].tb, tb);
```

```
30
31                  /* [...] */
32
33              }
34
35              cpu_loop_exec_tb(cpu, tb, pc, &last_tb, &tb_exit);
36
37              /* [...] */
38
39          }
40      }
41
42      /* [...] */
43  }
```

At line 14, the function `tb_lookup` is called in order to try to find an already existing translation block that corresponds to the next code to execute.

If no such translation block exists, `tb_gen_code` is called at line 19 to generate a translation block. The translation block obtained contains the result of the translation of the next guest architecture instructions to execute, translated to the guest architecture.

Then at lines 23 - 29, the new generated translation block is stored into the `cpu->tb_jmp_cache` cache.

Finally, `cpu_loop_exec_tb` is called at line 35 to execute the host architecture instructions associated to the translation block.

3.2.6 tb_lookup and tb_htable_lookup functions

Given some information about the current state of the simulated execution, the `tb_lookup` function tries to find an already generated translation block that corresponds to the next target program code to execute.

Here is an extract of the definition of `tb_lookup`, and of `tb_htable_lookup` which is called by `tb_lookup` :

Code Source : accel/tcg/cpu-exec.c

```
1 static inline TranslationBlock *tb_lookup(CPUState *cpu, vaddr pc,
2                                           uint64_t cs_base, uint32_t flags,
3                                           uint32_t cflags)
4 {
5     TranslationBlock *tb;
6     CPUJumpCache *jc;
7     uint32_t hash;
```

```
8
9
10      hash = tb_jmp_cache_hash_func(pc);
11      jc = cpu->tb_jmp_cache;
12
13      /* Use acquire to ensure current load of pc from jc. */
14      tb = qatomic_load_acquire(&jc->array[hash].tb);
15
16      if (likely(tb &&
17                 jc->array[hash].pc == pc &&
18                 tb->cs_base == cs_base &&
19                 tb->flags == flags &&
20                 tb_cflags(tb) == cflags)) {
21          return tb;
22      }
23      tb = tb_htable_lookup(cpu, pc, cs_base, flags, cflags);
24      if (tb == NULL) {
25          return NULL;
26      }
27      jc->array[hash].pc = pc;
28      /* Ensure pc is written first. */
29      qatomic_store_release(&jc->array[hash].tb, tb);
30
31
32      return tb;
33  }
```

Code Source : accel/tcg/cpu-exec.c

```
1  static TranslationBlock *tb_htable_lookup(CPUState *cpu, vaddr pc,
2                                            uint64_t cs_base, uint32_t flags,
3                                            uint32_t cflags)
4  {
5      tb_page_addr_t phys_pc;
6      struct tb_desc desc;
7      uint32_t h;
8
9      desc.env = cpu->env_ptr;
10     desc.cs_base = cs_base;
11     desc.flags = flags;
12     desc.cflags = cflags;
13     desc.pc = pc;
14     phys_pc = get_page_addr_code(desc.env, pc);
15     if (phys_pc == -1) {
16         return NULL;
17     }
18     desc.page_addr0 = phys_pc;
19     h = tb_hash_func(phys_pc, (cflags & CF_PCREL ? 0 : pc),
```

```
20                      flags, cs_base, cflags);
21     return qht_lookup_custom(&tb_ctx.htable, &desc, h, tb_lookup_cmp);
22 }
```

At lines 10 - 22 of `tb_lookup`, the function checks if a translation block that matches the current execution state exists in the `cpu->tb_jmp_cache` cache, and if so, returns it.

If the translation block is not found, at line 23 of `tb_lookup`, the function `tb_htable_lookup` is called. This function checks if a translation block that matches the current execution state exists in the `tb_ctx` cache.

If the translation block is found in the `tb_ctx` cache, it is added to the `cpu->tb_jmp_cache` cache at lines 27 - 29 of `tb_lookup`.

3.2.7 Function tb_link_page

We have seen that the function `cpu_exec_loop` calls the function `tb_gen_code` when it needs to create a new translation block. We will discuss in details how `tb_gen_code` generates a translation block in the next section. For now, what we need to know is that, after creating the new translation block, `tb_gen_code` calls the function `tb_link_page` and passes the new translation block as argument.

Here is an extract of the definition of `tb_link_page` :

Code Source : accel/tcg/tb-maint.c

```
1 TranslationBlock *tb_link_page(TranslationBlock *tb)
2 {
3     /* [...] */
4
5     uint32_t h;
6
7     /* [...] */
8
9     /* add in the hash table */
10    h = tb_hash_func(tb_page_addr0(tb), (tb->cflags & CF_PCREL ? 0 : tb->pc),
11                     tb->flags, tb->cs_base, tb->cflags);
12    qht_insert(&tb_ctx.htable, tb, h, &existing_tb);
13
14    /* [...] */
15
16    return tb;
17 }
```

As we can see, this function adds the translation block to the `tb_ctx` cache.

3.2.8 Function cpu_loop_exec

Given a translation block `tb`, the `cpu_loop_exec` function is responsible for executing the host instructions stored at `tb->tc.ptr`.

`cpu_loop_exec` internally calls `cpu_tb_exec` with the translation block as argument. This function itself calls the function `tcg_qemu_tb_exec` with `tb->tc.ptr` as argument.

`tcg_qemu_tb_exec` is responsible for passing control to the code stored at `tb->tc.ptr`. However, before doing so, `tcg_qemu_tb_exec` must perform a function prologue, in particular it must save on the stack the callee saved registers according to the architecture calling convention. Similarly, after the code stored at `tb->tc.ptr` has finished executing, `tcg_qemu_tb_exec` must do a proper function epilogue, in particular it must restore callee saved registers previously stored on the stack.

As a consequence, `tcg_qemu_tb_exec` can not be implemented in standard C code, as it needs to perform low level register manipulation. Instead, `tcg_qemu_tb_exec` is "defined" at the beginning of the execution of QEMU, by assigning the function `tcg_qemu_tb_exec` to a buffer, and injecting appropriate host architecture instructions into this buffer.

We encourage the interested reader to read the code of the functions `tcg_prologue_init` in *tcg/tcg.c* and `tcg_target_qemu_prologue` in *tcg/i386/tcg-target.c.inc* to see how this is done.

3.3 Generation of translation blocks

In the previous section, we have presented the code of QEMU that is responsible for obtaining and executing translation blocks. We have seen that the function `tb_gen_code` is responsible for generating a new translation block by translating a block of guest instructions to a block of host instructions.

In this section, we will zoom into the function `tb_gen_code` in order to see how the instructions are translated from guest to host architecture. We will specifically look at the functions involved in the translation from guest instructions to TCG ops, as this is at this translation step that SymQEMU modifies QEMU.

Figure 3.3 gives an overview of the functions called by `tb_gen_code`.

Like for the previous section, we will first present some structures used by those functions. In 3.3.1, we will present a structure used to represent the generated TCG ops. In 3.3.2, we will present a structure used internally to store information about the progress of the translation.

Then, in 3.3.3, 3.3.4 and 3.3.5, we will discuss the different functions represented in Figure 3.3.

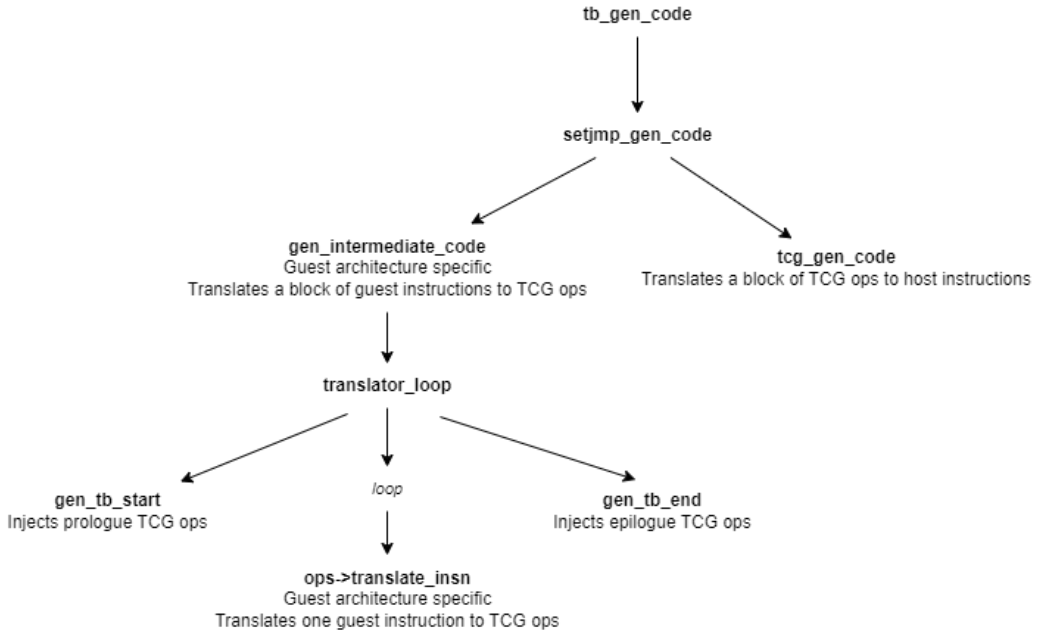

Figure 3.3: Function call graph for the generation of translation blocks.

3.3.1 struct TCGContext

A `TCGContext` instance represents the result of the translation of a block of host architecture instructions to TCG ops. In particular, it contains the list of generated TCG ops.

As the guest instructions are translated, the resulting TCG ops are progressively accumulated into the `TCGContext` instance. Instead of passing along the `TCGContext` instance as argument to all functions involved in the translation process, the developers of QEMU have chosen to create one `TCGContext` instance and store its address in a global pointer named `tcg_ctx`. Several functions involved in the translation directly manipulate the `TCGContext` instance through `tcg_ctx`.

To be more precise, there is actually one separate `TCGContext` instance for each thread of the host. `tcg_ctx` is declared using the GCC *thread-local storage* feature.

Note that some functions still take a pointer to the `TCGContext` instance as argument, instead of using the global `tcg_ctx`. In this situation, this local pointer to the `TCGContext` instance is usually named `s`.

Here is an extract of the definition of `TCGContext` and `tcg_ctx`

Code Source : include/tcg/tcg.c

```
1  #define TCG_MAX_TEMPS 512
2
3  /* [...] */
4
5  struct TCGContext {
6
7      /* [...] */
8
9      int nb_globals;
10     int nb_temps;
11
12     /* [...] */
13
14     QTAILQ_HEAD(, TCGOp) ops;
15
16     /* [...] */
17
18     TCGTemp temps[TCG_MAX_TEMPS];
19
20     /* [...] */
21
22 };
23
24 /* [...] */
25
26 typedef struct TCGContext TCGContext;
27
28 /* [...] */
29
30 extern __thread TCGContext *tcg_ctx;
```

- `nb_globals`, `nb_temps`, `temps` store data about *TCG variables*. We will discuss TCG variables in Section 3.4.

- `ops` is a doubly linked list that stores the generated TCG ops.

3.3.2 struct DisasContext and struct DisasContextBase

A `DisasContext` instance is used to store information about the current state of the translation when performing guest architecture instructions to TCG ops translation. When the translation of a new block of guest instructions is started, a new instance of `DisasContext` is created and is then passed along as argument to the different functions involved in the translation process.

The purpose of `DisasContext` may seem redundant with `TCGContext`. The difference between them is that `TCGContext` is used to store the result of the translation, i.e., it stores information that will be transmitted to the code responsible for TCG ops to host architecture translation. `DisasContext`, on the other hand, only stores information that is internal to the process of translating guest architecture instructions to TCG ops.

The definition of `DisasContext` is specific to the guest architecture that QEMU has been compiled for. For each guest architecture, there is a different definition of `DisasContext`, among which the appropriate one is chosen at compilation time.

The first member of `DisasContext` is an instance of `DisasContextBase`. The definition of `DisasContextBase` does not depend on the guest architecture QEMU is compiled for, it represents information that is common to all guest architectures.

This design can be seen as something similar to class inheritance in object-oriented programming : `DisasContextBase` is the "base class" and `DisasContext` "extends" it with members that are specific to the guest architecture. A pointer to a `DisasContext` can be safely casted to a pointer of type `DisasContextBase`, in a similar way as upacsting is performed in object-oriented programming.

Remarkably, `DisasContextBase` has some accompanying documenting comments. Here is an extract if its definition :

Code Source : include/exec/translator.h

```
/**
 * DisasContextBase:
    [...]
 * @is_jmp: What instruction to disassemble next.
 * @num_insns: Number of translated instructions (including current).
 * @max_insns: Maximum number of instructions to be translated in this TB.
    [...]
 * Architecture-agnostic disassembly context.
 */
typedef struct DisasContextBase {
    /* [...] */
    DisasJumpType is_jmp;
    int num_insns;
    int max_insns;
    /* [...] */
} DisasContextBase;
```

3.3.3 Functions tb_gen_code and setjmp_gen_code

Now that we have discussed the structures involved in the translation of the target program instructions, let us discuss the functions that use them.

`tb_gen_code` is the root funtion that triggers the translation of a block of guest architecture instructions to TCG ops, and then the translation of these TCG ops to a block of host architecture instructions.

Given information about the current state of the simulated execution, the `tb_gen_code` function is responsible for generating a new translation block for the next guest instructions that should be executed.

`tb_gen_code` internally calls the `setjmp_gen_code` function.

Here is an extract of the definition of both functions :

Code Source : accel/tcg/translate-all.c

```
1  TranslationBlock *tb_gen_code(CPUState *cpu,
2                                vaddr pc, uint64_t cs_base,
3                                uint32_t flags, int cflags)
4  {
5      CPUArchState *env = cpu->env_ptr;
6      TranslationBlock *tb;
7      int gen_code_size;
8
9      /* [...] */
10
11     void *host_pc;
12
13     /* [...] */
14
15     tb = tcg_tb_alloc(tcg_ctx);
16
17     /* [...] */
18
19     setjmp_gen_code(env, tb, pc, host_pc, &max_insns, &ti);
20
21     /* [...] */
22
23     tb_link_page(tb);
24
25     /* [...] */
26
27     return tb;
28 }
```

At line 15 of `tb_gen_code`, a new empty translation block is allocated. Then at line 19 this translation block is passed as argument to the function `setjmp_gen_code`, along with information about the current state of the simulated execution.

Now let us see what `setjmp_gen_code` does.

Code Source : accel/tcg/translate-all.c

```
1 static int setjmp_gen_code(CPUArchState *env, TranslationBlock *tb,
2                             vaddr pc, void *host_pc,
3                             int *max_insns, int64_t *ti)
4 {
5     /* [...] */
6
7     gen_intermediate_code(env_cpu(env), tb, max_insns, pc, host_pc);
8
9     /* [...] */
10
11    return tcg_gen_code(tcg_ctx, tb, pc);
12 }
```

First, `setjmp_gen_code` calls the function `gen_intermediate_code`, which translates the next block of guest architecture instructions to TCG ops, and stores those TCG ops in the global variable `*tcg_ctx`.

Then, `setjmp_gen_code` calls the function `tcg_gen_code` with `tcg_ctx` and the translation block as argument. `tcg_gen_code` translates the TCG ops stored by `*tcg_ctx` to host architecture instructions, and stores the result into the translation block.

Finally, going back to `tb_gen_code`, at line 23 the function `tb_link_page` is called with the translation block as argument. As discussed in 3.2.7, `tb_link_page` adds the translation block to a cache of generated translation blocks.

3.3.4 Function gen__intermediate__code

`gen_intermediate_code` is the root function that triggers the translation of a block of guest architecture instructions to a list of TCG ops.

This translation task is different depending on which guest architecture QEMU is compiled for. Let us explain the modular design used by QEMU.

The implementation of `gen_intermediate_code` depends on the guest architecture QEMU is compiled for.

`gen_intermediate_code` internally calls the function `translator_loop`. The definition of this function does not depend on which guest architecture QEMU is compiled for, as `translator_loop` only contains high level translation code logic that is common to all guest architectures. `translator_loop` receives as argument a collection of pointers to guest architecture dependant functions. While translating, `translator_loop` uses those functions to perform translation tasks that are specific to the guest architecture.

As an illustration, let us look at the definitions of `gen_intermediate_code` for the x86

guest architecture and ARM guest architecture :

Code Source : target/i386/tcg/translate.c

```
1  static const TranslatorOps i386_tr_ops = {
2      .init_disas_context = i386_tr_init_disas_context,
3      .tb_start           = i386_tr_tb_start,
4      .insn_start         = i386_tr_insn_start,
5      .translate_insn     = i386_tr_translate_insn,
6      .tb_stop            = i386_tr_tb_stop,
7      .disas_log          = i386_tr_disas_log,
8  };
9
10 void gen_intermediate_code(CPUState *cpu, TranslationBlock *tb, int *max_insns,
11                             target_ulong pc, void *host_pc)
12 {
13     DisasContext dc;
14
15     translator_loop(cpu, tb, max_insns, pc, host_pc, &i386_tr_ops, &dc.base);
16 }
```

Code Source : target/arm/tcg/translate.c

```
1  const TranslatorOps aarch64_translator_ops = {
2      .init_disas_context = aarch64_tr_init_disas_context,
3      .tb_start           = aarch64_tr_tb_start,
4      .insn_start         = aarch64_tr_insn_start,
5      .translate_insn     = aarch64_tr_translate_insn,
6      .tb_stop            = aarch64_tr_tb_stop,
7      .disas_log          = aarch64_tr_disas_log,
8  };
9
10 static const TranslatorOps arm_translator_ops = {
11     .init_disas_context = arm_tr_init_disas_context,
12     .tb_start           = arm_tr_tb_start,
13     .insn_start         = arm_tr_insn_start,
14     .translate_insn     = arm_tr_translate_insn,
15     .tb_stop            = arm_tr_tb_stop,
16     .disas_log          = arm_tr_disas_log,
17 };
18
19 static const TranslatorOps thumb_translator_ops = {
20     .init_disas_context = arm_tr_init_disas_context,
21     .tb_start           = arm_tr_tb_start,
22     .insn_start         = arm_tr_insn_start,
23     .translate_insn     = thumb_tr_translate_insn,
24     .tb_stop            = arm_tr_tb_stop,
25     .disas_log          = arm_tr_disas_log,
26 };
```

```
27
28 void gen_intermediate_code(CPUState *cpu, TranslationBlock *tb, int *max_insns,
29                             target_ulong pc, void *host_pc)
30 {
31     DisasContext dc = { };
32     const TranslatorOps *ops = &arm_translator_ops;
33     CPUARMTBFlags tb_flags = arm_tbflags_from_tb(tb);
34
35     if (EX_TBFLAG_AM32(tb_flags, THUMB)) {
36         ops = &thumb_translator_ops;
37     }
38 #ifdef TARGET_AARCH64
39     if (EX_TBFLAG_ANY(tb_flags, AARCH64_STATE)) {
40         ops = &aarch64_translator_ops;
41     }
42 #endif
43
44     translator_loop(cpu, tb, max_insns, pc, host_pc, ops, &dc.base);
45 }
```

As we can see, `translator_loop` receives as argument a `TranslatorOps` structure instance, which is populated with pointers to functions that are specific to the guest architecture.

We can additionally observe that, for ARM, the functions passed to `translator_loop` are different depending on whether the guest uses basic ARM, the Thumb extension or the Aarch64 extension.

3.3.5 Function translator_loop

Given information about the current state of the simulated execution, a pointer `op` to a `TranslatorOps` instance and a pointer `db` to a `DisasContext` instance, the function `translator_loop` is responsible for repetitively fetching a guest architecture instruction and translating it to TCG ops.

As the translation progresses, the resulting TCG ops are accumulated in the `TCGContext` instance whose address is stored in the global pointer `tcg_ctx`.

Here is an extract of the definition of `translator_loop` :

Code Source : accel/tcg/translator.c

```
1 void translator_loop(CPUState *cpu, TranslationBlock *tb, int *max_insns,
2                       vaddr pc, void *host_pc, const TranslatorOps *ops,
3                       DisasContextBase *db)
4 {
5
```

```
6     /* [...] */
7
8     /* Initialize DisasContext */
9
10    /* [...] */
11
12    db->is_jmp = DISAS_NEXT;
13    db->num_insns = 0;
14    db->max_insns = *max_insns;
15
16    /* [...] */
17
18    gen_tb_start(db, cflags);
19    ops->tb_start(db, cpu);
20
21    /* [...] */
22
23    while (true) {
24
25        /* [...] */
26
27        /* Disassemble one instruction.  The translate_insn hook should
28           update db->pc_next and db->is_jmp to indicate what should be
29           done next - either exiting this loop or locate the start of
30           the next instruction.  */
31
32        /* [...] */
33
34        ops->translate_insn(db, cpu);
35
36        /* [...] */
37
38        /* Stop translation if translate_insn so indicated.  */
39        if (db->is_jmp != DISAS_NEXT) {
40            break;
41        }
42
43        /* Stop translation if the output buffer is full,
44           or we have executed all of the allowed instructions.  */
45        if (tcg_op_buf_full() || db->num_insns >= db->max_insns) {
46            db->is_jmp = DISAS_TOO_MANY;
47            break;
48        }
49    }
50
51    /* Emit code to exit the TB, as indicated by db->is_jmp.  */
52    ops->tb_stop(db, cpu);
53    gen_tb_end(tb, cflags, icount_start_insn, db->num_insns);
54
```

```
55      /* [...] */
56
57  }
```

The `translator_loop` function is composed of some initialization code, followed by a main loop, followed by some finalization. Let us discuss the initialization and finalization code before looking at the main loop.

The functions called at lines 18 - 19 and 52 - 53 inject additional TCG ops at the beginning and at the end of the translated block. Those added TCG ops allow QEMU to control what will happen before and after the execution of the block of instructions. For instance, those instructions allow to directly jump to another block of instructions without the need to go back the main loop in `cpu_exec_loop` (a mechanism named *direct block chaining[3]*), to give back control to QEMU if an interruption happens, etc.

Now let us discuss the main loop. At line 34, `ops->translate_insn` is called. This function is responsible for fetching one guest instruction and for translating it to TCG ops. The definition of this function is specific to the guest architecture. It is the heart of the guest architecture to TCG ops translation process, and we will discuss in details how it generates TCG ops in the next section.

Additionally, as described in the comment, this function updates the value of `db->is_jmp` to communicate if the translation should continue or if we should terminate the instruction block.

At lines 39 - 47, we exit the translation loop and finalize the block either if `ops->translate_insn` has requested so, if we have reached the maximum number of generated TCG ops or if we have translated the maximum number of guest instructions.

3.4 The TCG API

In the previous sections, we have discussed the code of QEMU that generates and executes translation blocks. We have progressively zoomed into the code responsible for translating from guest architecture to TCG ops, and we have finally seen that the function responsible for translating one guest architecture to TCG ops is `ops->translate_insn`.

The process of translating a guest architecture instruction to TCG ops can be conceptually separated into three tasks :

- Fetching and disassembling the guest instruction.
- Determining which TCG ops to generate with respect to the semantic of the fetched instruction.

- Actually generating the TCG ops.

This section explains in details how the third task is performed by QEMU.

QEMU contains a series of functions that allow to programmatically "write" TCG code. When one of those functions is called, it modifies the content of the `TCGContext` instance to which `tcg_ctx` points. In particular, those functions typically add a TCG op to the list `tcg_ctx->ops` of currently generated ops.

Let us call this collection of functions for writing TCG code the *TCG API*. Furthermore, let us call a TCG API *client* any code that calls functions of the TCG API. In contrast, let us call *TCG backend* the internal code that makes this API work.

The code in `ops->translate_insn` is a client of the TCG API. Additionally, some initialization code that is run at the beginning of the execution of QEMU is also a client of the TCG API, as it uses the API to create *global TCG variables*. We will explain what a TCG variable is in 3.4.1.

In this section, we will discuss the TCG API from the point of view of the client.

In 3.4.1, we will present an overview of TCG, and we will show an example of TCG client code. We will see that TCG provides a set of TCG instructions whose operands are TCG variables.

In 3.4.2 we will discuss TCG variable types. We will see that each TCG variable has a type and that TCG instructions expect their operands to be of specific types.

In 3.4.3, 3.4.4 and 3.4.5, we will see that TCG variables can be either temporary, constant or globals, and we will see how the TCG client creates each kind of TCG variable. We will see that global TCG variables are used to represent guest registers.

In 3.4.6 we will discuss TCG instructions. In 3.4.7 we will discuss vector TCG instructions, that allow to perform SIMD-style computations.

In 3.4.8 we will discuss the TCG instructions that allow to access memory. We will see that there exist instructions for accessing the guest memory and other instructions for accessing the host memory.

In 3.4.9 we will discuss how the TCG client manages the simulated guest register.

In 3.4.10 we will present a mechanism called helper functions, that allows authors of client code to write C functions that will be executed at run time. This feature is used to allow the translation of complex guest architecture instructions that can not easily be translated to TCG ops. Finally, in 3.4.11 we will discuss some technical aspects of how helper functions are implemented in QEMU.

3.4.1 Overview of TCG

In many aspects, the TCG IR is similar to a RISC instruction set architecture, like ARM. A TCG op is composed of an *instruction* (like add, sub, xor, ...) and of a number of *operands* that designate the input and the output of the operation.

One important difference is that, in TCG, the operands of an instruction designate *TCG variables*, that the TCG client creates using the TCG API. This is an important difference from instruction set architectures, where instruction operands typically designate registers whose number and name is fixed.

To illustrate the use of the TCG API, we will now present an example of TCG API client code and explain what it does.

Consider the following code :

Code Source :

```
1 TCGv_i64 temp_variable1 = tcg_temp_new_i64();
2 TCGv_ptr address1 = tcg_constant_ptr(0xdeadbeef);
3 tcg_gen_ld_i64(temp_variable1, address1, 0);
4
5 TCGv_i64 temp_variable2 = tcg_temp_new_i64();
6 TCGv_ptr address2 = tcg_constant_ptr(0x42421337);
7 tcg_gen_ld_i64(temp_variable2, address2, 0);
8
9 tcg_gen_add_i64(temp_variable1, temp_variable1, temp_variable2);
10
11 tcg_gen_st_i64(temp_variable1, address1, 0);
```

This code creates TCG ops that will, at run time, read the value stored in memory at address 0xdeadbeef, read the value stored in memory at address 0x42421337, compute the sum of those two values and then store the result in memory at address 0xdeadbeef.

At line 1, a new temporary TCG variable of type *i64* is created. Then at line 2, a constant TCG variable of type *i64* is created, with value 0xdeadbeef.

At line 3, a TCG *ld* instruction is added, with the temporary TCG variable as first operand, the constant TCG variable as second operand and the immediate value 0 as third operand. In TCG, the first operand of a *ld* instruction is the destination TCG variable, the second is the load address and the third is an immediate value that represents an offset from the load address.

The lines 5 - 7 are similar to lines 1 - 3.

Then at line 9, a TCG *add* instruction is added, that will compute the sum of the loaded values. In TCG, the first operand of an *add* instruction is the destination. Finally, at line 11, a TCG *st* instruction is added to store the result in memory.

Here it is important to understand that, when the above C code is executed, the loads, the addition and the store are not actually performed. Instead, the effect of executing this C code is simply to add new TCG ops that represent those operations into `tcg_ctx->ops`. Only later, when those TCG ops will be translated to host architecture instructions and executed, the actual memory reads, addition and memory write will occur.

In particular, the C variables `temp_variable1` and `temp_variable2` never contain the loaded values. What they store is a unique ID that identifies the TCG variable that the TCG API created when `tcg_temp_new_i64` was called. When calling `tcg_gen_add_i64(temp_variable1, temp_variable1, temp_variable2)`, what we actually do is passing the unique IDs of the previously created TCG variables to the TCG API, in order to indicate that those TCG variables must be the operands of the newly created *add* instruction.

Note that later, when the TCG ops will be translated to host architecture instructions, each TCG variable will be assigned either to a host register or to a location in host memory. However, the client TCG API code does not need to care about this.

This distinction between a TCG variable and the C variable that represents it can be a source of ambiguity when discussing client code for the TCG API. We will do our best to avoid such confusions in the explanations below.

3.4.2 TCG variable types

Each TCG variable has a type, and each TCG instruction expects its operands to have a specific type. For instance, in the example of 3.4.1 `tcg_temp_new_i64` generates a TCG variable of type i64, and `tcg_gen_add_i64` generates an *add_i64* instruction that specifically requires its operands to have type i64. There exists another add instruction, *add_i32*, that acts on operands of type i32.

For each possible TCG variable type, there is a corresponding associated C type in QEMU, for a C variable that represents a TCG variable. Even though a C variable representing a TCG variable actually simply stores the ID of the TCG variable, having different C types for the different TCG variable types is an elegant way of preventing the programmer that writes TCG API client code to accidentally create a TCG op with an operand having a type that does not match the instruction.

For instance, the C type for a C variable that represents an i64 TCG variable is `TCGv_i64`. As we can see in the example of 3.4.1, `tcg_temp_new_i64` returns a value of type `TCGv_i64`, and `tcg_gen_add_i64` expects arguments of C type `TCGv_i64`. Calling `tcg_gen_add_i64` with the value returned by a call to `tcg_temp_new_i32` would trigger a compilation error.

Table 3.1 lists the different TCG variable types and their corresponding C types.

i32 and i64 TCG variables are used as operands for standard TCG instructions, like arithmetic operations, logical operations, shifts, etc. Each of those instructions have an i32 and

C type for a C variable that represents this TCG variable	Description of the TCG variable type
`TCGv_i32`	Standard TCG variable of size 32 bits
`TCGv_i64`	Standard TCG variable of size 64 bits
`TCGv_i128`	Standard TCG variable of size 128 bits
`TCGv_ptr`	TCG variable for holding a pointer to host memory. Has size 32 or 64 bits depending on whether the host architecture is 32 or 64 bits.
`TCGv_vec`	Vector TCG variable. Can have size 64, 128 or 256 bits.

Table 3.1: TCG variable types.

an i64 version.

i128 TCG variables are not widely used. As far as we know, the only instructions that use them are *qemu_ld_i128* and *qemu_st_i128*, which are discussed in 3.4.8.

For load / store instructions that target host memory, the operand that designates the memory address has type ptr.

Vec TCG variables are used as operands of special vector instructions that we will discuss in 3.4.7.

3.4.3 Temporary TCG variables

We will now present different TCG API functions that allow to create TCG variables, and explain the specific characteristics of the variables they create.

The functions `tcg_temp_new_<i32 / i64 / i128 / ptr / vec>` create a new *temporary* TCG variable of type i32, i64, i128, ptr or vec.

Temporary variables do not persist beyond the current block of translated instructions. Those variables are used in situations like the example of 3.4.1, for storing short term intermediate computation results.

3.4.4 Constant TCG variables

The functions `tcg_constant_<i32 / i64 / ptr / vec>` create a new *constant* TCG variable of type i32, i64, ptr or vec. Constant TCG variables can not have type i128.

The value of the TCG variable is directly given as argument to `tcg_constant_<i32 / i64 / ptr / vec>` and it never changes.

Constant TCG variables are convenient when we want to pass an "immediate" value as operand of an instruction that expects a TCG variable. For example, we used constant TCG variables to store the memory addresses in the example of 3.4.1, as those addresses were already known at the time of generating the TCG ops. Without constant TCG variables, in such a situation it would be necessary to first create a new standard TCG variable and then add a *movi* TCG instruction to move the appropriate variable into the value.

Like for temporary TCG variables, constant TCG variables do not persist beyond the current block of translated instructions.

3.4.5 Global TCG variables

Global TCG variables are TCG variables that are created at the beginning of the execution of QEMU, and that persist during the whole lifetime of the execution. Global TCG variables are typically used to represent registers of the guest CPU.

To understand the functioning of TCG variables, let us illustrate it with a concrete example.

If QEMU is compiled for the x86 guest architecture, the following line of code will be executed during the initialization phase, before any translation happens :

Code Source :

```
1 TCGv_i64 rax_tcg_variable = tcg_global_mem_new_i64(cpu_env, offsetof(CPUArchState,
  ↪  regs[0]), "rax");
```

Note that the code above is a bit of a simplification. We invite the intersted reader to read the actual implementation of the function `tcg_x86_init` in *target/i386/tcg/translate.c*.

This line of code creates a new global TCG variable, which is associated to the RAX register of the simulated guest CPU. Let us detail it :

`tcg_global_mem_new_i64` is a function of the TCG API that creates a new global TCG variable of type i64.

`cpu_env` is a global C variable that is initialized at the beginning of the execution of QEMU. It represents a special TCG variable that designates the address of the `CPUArchState` instance. Recall that, as explained in 3.2.3, `CPUArchState` is a struct that represents the state of the simulated guest CPU, and in particular stores the values of the different guest CPU registers.

`offsetof(CPUArchState, regs[0])` is the offset, in bytes, of the RAX register in the `CPUArchState` instance. `offsetof(A, B)` is a macro that computes the offset of the member B of the struct A, relatively to the start of A. As shown in 3.2.3, `regs` is a member array of `CPUArchState` that represents the standard x86 registers. Its first element is RAX.

Now, during the translation phase, the global TCG variable created above will be used to manipulate the simulated RAX guest register. For example, suppose that the code that performs the translation from guest architecture to TCG ops makes the following API call :

Code Source :

```
1 tcg_gen_add_i64(rax_tcg_variable, rax_tcg_variable, rax_tcg_variable);
```

This creates a TCG op that doubles the value of the RAX register of the simulated guest CPU.

When the translation from TCG ops to host architecture will happen, this specific TCG op will be translated to host instructions that access the host memory and double the value of `cpu_arch_state->regs[0]` (where `cpu_arch_state` is the `CPUArchState` instance).

Let us now present a more formal discussion on global TCG variables.

The function `tcg_global_mem_new_<i32 / i64 / ptr>` creates a new global TCG variable of type i32, i64 or ptr.

This function can be called in the following way :

`tcg_global_mem_new_<i32 / i64 / ptr>(cpu_env, <offset>, <name>)`, where :

- `cpu_env` is the global QEMU C variable that represents a special TCG variable designating the `CPUArchState` instance.
- `<offset>` is the offset in bytes of the `CPUArchState` member to which the new TCG global variable should be associated.
- `<name>` is a name associated to the global TCG variable.

TCG global variables created this way are called *direct globals* [4].

The function can also be called in the following way : `tcg_global_mem_new_<i32 / i64 / ptr>(<global TCG variable>, <offset>, <name>)`, where :

- `<global TCG variable>` is an already existing direct global TCG variable. Let call A the address `<global TCG variable>` is associated to.
- `<offset>` is the offset in bytes, relatively to A, of the `CPUArchState` member to which the new TCG global variable should be associated.

TCG global variables created this way are called *indirect globals* [4].

As far as we know, the `<name>` argument does not have any effect on a normal execution of QEMU. However, QEMU has a debugging feature that allows to print the TCG ops during

the execution; when doing this printing, QEMU conveniently uses the name assigned to the global variables to designate them.

All global TCG variables must be created before any temporary or constant TCG variable is created [4].

Global TCG variables can not have type i128 or vec.

3.4.6 TCG instructions

TCG provides instructions for performing all the usual computations, like arithmetic operations, logical operations, shifts and rotates, conversion between i32 and i64, etc. The list of TCG instructions is provided in the documentation of QEMU [4].

For each TCG instruction, there is a corresponding function `tcg_gen_<instruction name>` provided in the TCG API, that takes operand TCG variables as arguments and adds a new TCG op. For example, we have seen that an *add_i64* instruction is created by calling the function `tcg_gen_add_i64`.

Some instructions have immediate value operands, in which case the immediate value is given as argument to the `tcg_gen_<instruction name>` in the form of a C integer.

3.4.7 Vector instructions

Vector TCG instructions and vector TCG variables are a special feature of TCG that allow to perform SIMD (single instruction, multiple data) computations.

A vector TCG variables can have a length of 64, 128 or 256 bits. When creating a new TCG variable with `tcg_temp_new_vec` or `tcg_constant_new_vec`, an argument of type `TCGType` must be passed to the API function to indicate the vector size.

When an operation on vector TCG variables is performed, each vector operand is subdivided in several vector *elements*, and the computation is performed in parallel on the different elements.

Figure 3.4 gives a visual representation of a *vec_add* instruction with element size 64, where D, S1 and S2 are vector TCG variables of size 256 bits.

The complete list of vector instructions is provided in the QEMU documentation [4].

All TCG vector instructions take as first operand an immediate value named `vece` that controls the element size for this instruction. For a `vece` value `a`, the element size will be $8 \cdot 2^a$. For instance, in the example given in Figure 3.4, the value of `vece` was 3.

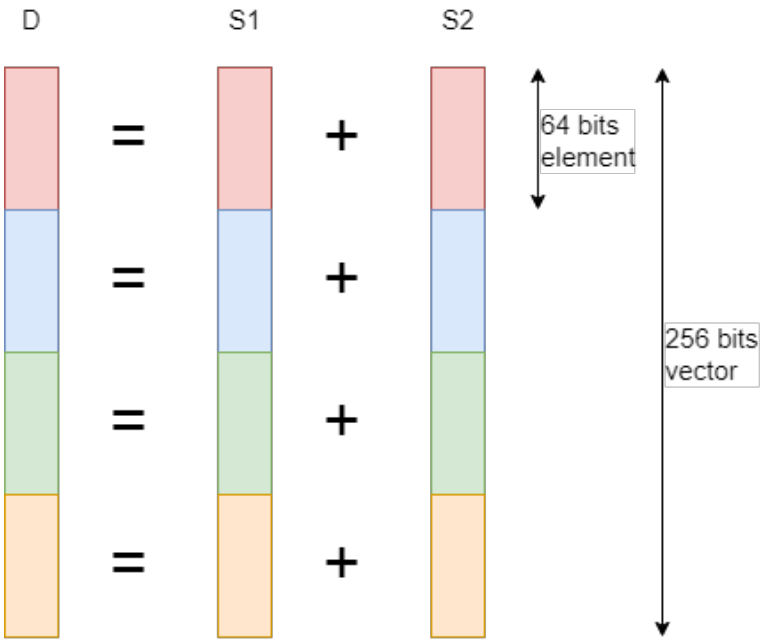

Figure 3.4: Execution of an add_vec instruction.

3.4.8 Host and guest memory

When executing the guest program, QEMU will dedicate space inside its own memory for the memory segments of the guest program, like the stack, the text segment, the data segment, etc. However, there is no guaranty that those segments will be located at the addresses expected by the guest program.

Typically, a non PIE (position independent executable) guest program performs memory access to the text, data and bss segments using absolute addresses. To solve this problem, QEMU implements a mechanism of guest to host address translation, that can be seen as something similar to the virtual to physical address translation performed by a MMU.

The TCG instructions *qemu_ld_i32/i64/i128* and *qemu_st_i32/i64/i128* perform load / store to the guest memory. When those instructions are executed, the target address is internally translated from guest to host by QEMU.

The TCG instructions *ld_i32/i64* and *st_i32/i64* (and their variants, see the documentation) perform load / store to the host memory. They are executed directly, without address translation. Those instructions are used when the TCG API client wants to create TCG code that will directly read or write to internal structures of QEMU.

3.4.9 Guest registers

We have seen that the registers of the simulated guest CPU are stored in an instance of the structure `CPUArchState`.

When translating guest instructions that read or write to the guest registers, the TCG API client needs to generate TCG code that will perform equivalent read or writes on the appropriate members of the `CPUArchState` instance.

We can now see that the TCG API client can do this in two different ways :

- The first option is to have a global TCG variable for the guest register, that is associated to the appropriate location inside the `CPUArchState` instance. Then, during translation, the TCG API client can use this global TCG variable as operand of TCG ops. As we have seen, those TCG ops will be translated to host instructions that will read and write the virtual register in the `CPUArchState` instance.
- The second option is to directly emit TCG ops that perform loads and stores at the address of the simulated register inside the `CPUArchState` instance. As we have seen in 3.4.8, TCG allows to emit load and store instructions that directly target host memory addresses.

The actual implementation of the translation code actually uses both of those approaches. For example, if QEMU is compiled for the x86 guest architecture, global TCG variables are created for the common registers like RAX, RBX, etc. However, the manipulations of other less commonly used registers like XMM1, XMM2, etc. is performed using direct host memory access, with *ld_i32/i64* and *st_i32/i64* TCG instructions.

3.4.10 Helper functions

We have seen that the code that perform translation from guest architecture instructions to TCG has to generate TCG ops that have an equivalent behavior as the original guest instructions.

TCG offers an alternative mechanism, called *helper functions*. A helper function is a C function, that is written by the author of the TCG API client code, and that is called from the TCG code at run time. Helper functions are typically used to implement complex guest architecture instructions that do not have any equivalent TCG instruction.

Let us present a concrete example. Suppose that you are a developer of some TCG API client code, and that you need a helper function for computing the sum of two TCG variables, and another helper function for computing the difference between two TCG variables.

Of course, in reality there is no need to have such helper functions as there exists TCG instructions for performing addition and subtraction, but we will use this simple example.

In this situation, you would create two files *example.h* and *example.c* with the following content :

Code Source : example.h

```
1 DEF_HELPER_FLAGS_2(my_add, TCG_CALL_NO_RWG, i64, i64)
2 DEF_HELPER_FLAGS_2(my_sub, TCG_CALL_NO_RWG, i64, i64)
```

Code Source : example.c

```
1  #define HELPER_H "example.h"
2  #include "exec/helper-proto.h.inc"
3  #undef  HELPER_H
4
5  #define HELPER_H  "example.h"
6  #include "exec/helper-info.c.inc"
7  #undef  HELPER_H
8
9  uint64_t HELPER(my_add)(uint64_t a, uint64_t b){
10     return a + b;
11 }
12
13 uint64_t HELPER(my_sub)(uint64_t a, uint64_t b){
14     return a - b;
15 }
```

Then, thanks to code automatically generated by macro expansion, you can simply use your helper functions in your TCG API client code in the following way :

Code Source : client.c

```
1  #define HELPER_H  "example.h"
2  #include "exec/helper-gen.h.inc"
3  #undef  HELPER_H
4
5
6  void foo(){
7      TCGv_i64 tcg_variable_dst = /* [...] */;
8      TCGv_i64 tcg_variable_src1 = /* [...] */;
9      TCGv_i64 tcg_variable_src2 = /* [...] */;
10
11     /* [...] */
12
13     gen_helper_my_add(tcg_variable_dst, tcg_variable_src1, tcg_variable_src2);
14
15     /* [...] */
16
17     gen_helper_my_sub(tcg_variable_dst, tcg_variable_src1, tcg_variable_src2);
```

```
18 }
```

`gen_helper_my_add` is a TCG API function that is automatically created by the preprocessor from the content of your *example.h* and *example.c* files.

`gen_helper_my_add` adds TCG instructions that, at run time, will trigger a call to the `my_add` function. At run time, the `my_add` function will be called with the content of the TCG variables `tcg_variable_src1` and `tcg_variable_src2` as argument, and the result will be stored in the TCG variable `tcg_variable_dst`.

Similarly, `gen_helper_my_sub` is a TCG API function that creates TCG ops for calling the `my_sub` function at run time.

Now, let us develop what those macro actually do.

The preprocessor replaces lines 1 - 3 of *example.c* with the following function prototypes :

```
helper_my_add(uint64_t, uint64_t, uint64_t);
helper_my_sub(uint64_t, uint64_t, uint64_t);
```

Then it replaces lines 5 - 7 of *example.c* with the instantiation of two `TCGHelperInfo` instances, one for each helper function. `TCGHelperInfo` is a struct that QEMU uses to store metadata about helper functions.

At lines 9 and 13 of *example.c*, the expansion of the `HELPER` macros will result in `helper_my_add` and `helper_my_sub`, respectively.

Finally, lines 1 - 3 of *client.c* will be expanded to the declaration and definition of the two TCG API functions `gen_helper_my_add` and `gen_helper_my_sub`. Those functions internally call the function `tcg_gen_callN` with the `TCGHelperInfo` instance as argument. `tcg_gen_callN` is the internal TCG API function that is responsible for adding TCG ops that will call the helper function, and pass the source and destination TCG variable values.

3.4.11 Chains of includes for helper functions

Before continuing, let us precise some terminology.

We call *helper function* the function that is written by the developer and that will be called a run time. For example, `helper_my_add` is a helper function.

In contrast, we call *API function for calling a helper* the function that is automatically generated by the macro expansion and that is called at translation time. For example, `gen_helper_my_add` is an API function for calling a helper.

Now, we have seen that a C file that uses an API function for calling a helper has to include preprocessor directives that follow the same pattern as lines 1 - 3 of of *client.c*. To avoid

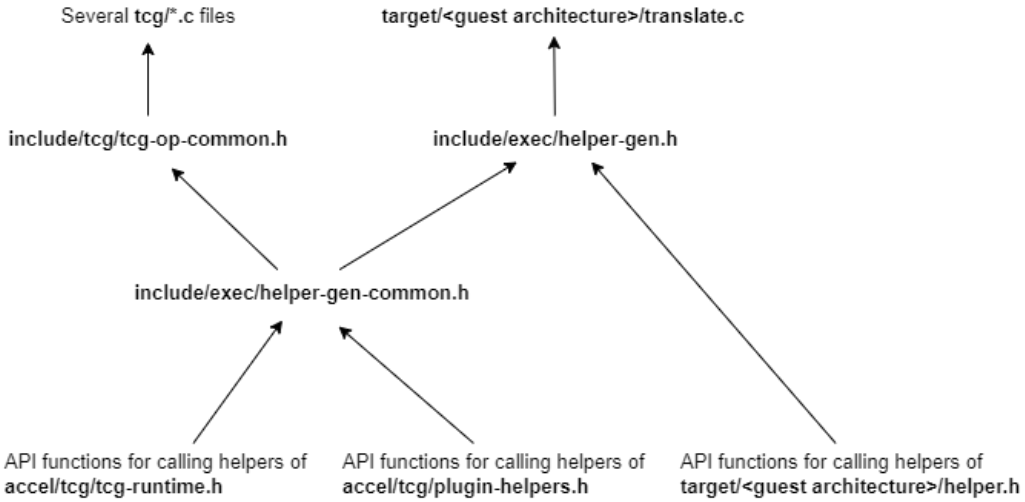

Figure 3.5: Graph of includes for API functions for calling helpers.

duplication of those directives into multiple C files, the QEMU source code actually has some header files whose purpose is to contain those directives.

This situation creates some quite complex chains of includes. Figure 3.5 represents the graph of those includes. Beware that, in this graph, an arrow from A to B means that A is included in B (and not that A includes B).

Similarly, C files that make direct calls to helper functions need to include preprocessor directives which follow the same pattern as lines 1 - 3 of *example.c*, and that will be replaced with the prototypes of the helper functions. It is not unusual for a helper function to make a call to another helper function, for example to avoid code repetition.

There are similar chains of includes for the helper function prototypes. Figure 3.6 represents the graph of those includes.

3.5 Internal implementation of TCG variables in the TCG backend

In the previous section, we have described the TCG API in the point of view of the client of this API. In this section, we are going to see the internals of how the TCG backend generates TCG variables.

Figure 3.7 represents the graph of functions that are involved in the generation of TCG variables. Functions in the green area are the API functions exposed to the client, that we have discussed in 3.4.3, 3.4.4 and 3.4.5. Functions in the red area are backend functions

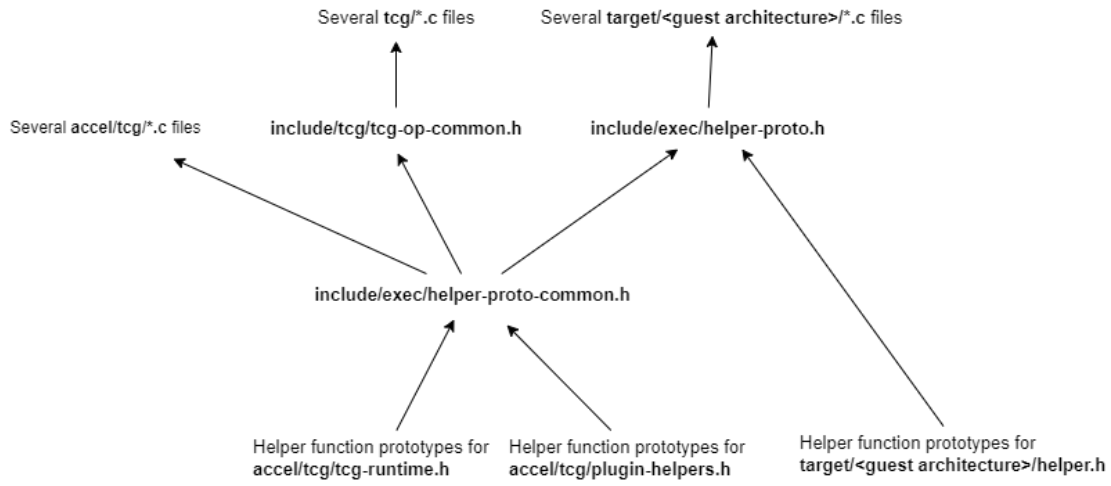

Figure 3.6: Graph of includes for helper functions.

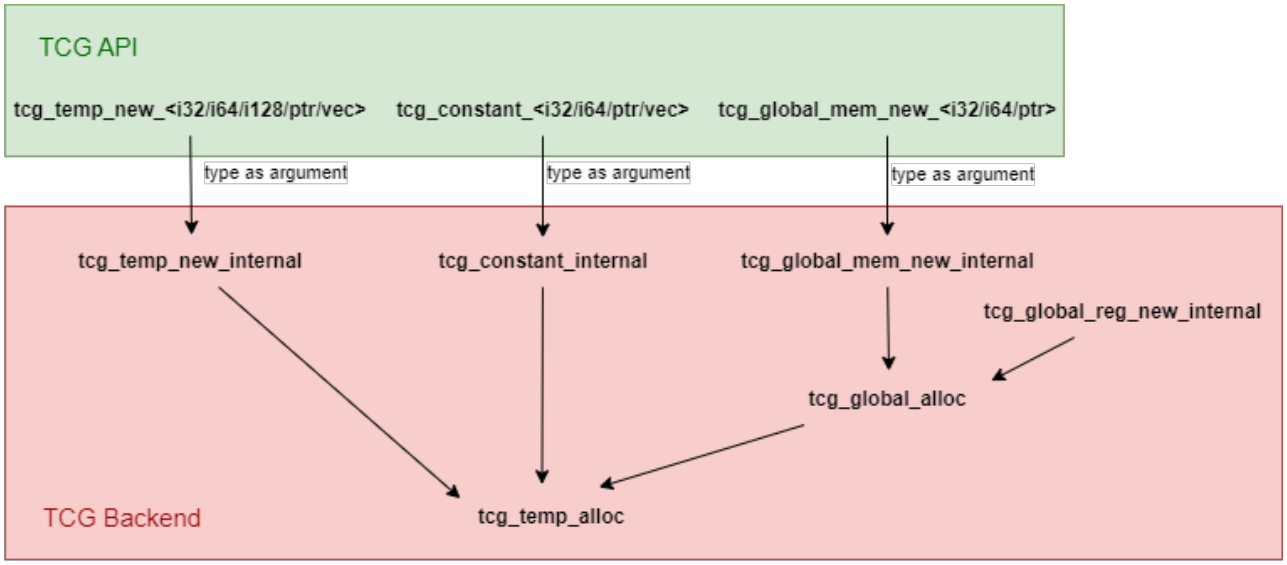

Figure 3.7: Function call graph for the generation of TCG variables.

that perform the actual work.

The API functions `tcg_temp_new_<TCG type>` are simple wrappers that call the function `tcg_temp_new_internal` with `<TCG type>` as argument. Similarly, `tcg_constant_<TCG type>` functions are wrappers for `tcg_constant_internal` and `tcg_global_mem_new_<TCG type>` are wrappers for `tcg_global_mem_new_internal`.

In 3.5.1, we will first present `TCGTemp`, which is the C struct that the backend uses to represent a TCG variable. Then in 3.5.2 we will see where the backend stores TCG variables. In 3.5.3, we will explain how types used by the API like `TCGv_i32` or `TCGv_i64` are internally converted to and from `TCGTemp` pointers.

We will then discuss each backend function that appears in Figure 3.7. We will do this with a bottom up approach, by first presenting `tcg_temp_alloc` in 3.5.5 and `tcg_global_alloc` in 3.5.6, then functions `tcg_temp_new_internal`, `tcg_constant_internal`, `tcg_global_reg_internal` and `tcg_global_mem_internal`, in 3.5.7, 3.5.8, 3.5.9 and 3.5.10 respectively.

3.5.1 Struct TCGTemp

A `TCGTemp` instance represents a TCG variable.

In 3.4.3, 3.4.4 and 3.4.5, we have seen that the TCG API allows to create temporary, constant and global TCG variables. Very counterintuitively, `TCGTemp` is used to represent any kind of TCG variable, not only temporary ones. We do not know the motivation behind this confusing naming choice, it may be an artifact of the history of QEMU.

Here is an extract of the definition of `TCGTemp` :

Code Source : include/tcg/tcg.h

```
1  typedef struct TCGTemp {
2
3      /* [...] */
4
5      TCGType type:8;
6      TCGTempKind kind:3;
7      unsigned int indirect_reg:1;
8      unsigned int indirect_base:1;
9
10     /* [...] */
11
12     struct TCGTemp *mem_base;
13     intptr_t mem_offset;
14     const char *name;
15
16     /* [...] */
17
```

```
18 } TCGTemp;
```

- `type` is an enum value that indicates the type of the variable. Possible values are :
  - `TCG_TYPE_I32`
  - `TCG_TYPE_I64`
  - `TCG_TYPE_I128`
  - `TCG_TYPE_PTR`, which is an alias for `TCG_TYPE_I32` or `TCG_TYPE_I64`, depending on the host architecture
  - `TCG_TYPE_V64`
  - `TCG_TYPE_V128`
  - `TCG_TYPE_V256`
- `kind` is an enum value that indicates whether the TCG variable is temporary, constant or global. Possible values are :
  - `TEMP_TB` for a variable created with `tcg_temp_new_<i32/i64/i128/ptr/vec>`.
  - `TEMP_CONST` for a variable created with `tcg_constant_<i32/i64/ptr/vec>`.
  - `TEMP_GLOBAL` for a variable created with `tcg_global_mem_new_<i32/i64/ptr>`.
  - `TEMP_FIXED` for the special `env` TCG variable.
- `indirect_reg` is used for global variables. It is set if the variable is an indirect global.
- `indirect_base` is used for global variables. It is set if this variable is the base of an indirect global.
- `mem_base` is used for global variables. It is the base TCG variable passed by the TCG API client.
- `mem_offset` is used for global variables. It is the offset passed by the TCG API client.
- `name` is used for global variables. It is the name chosen by the TCG API client.

As a remainder, in 3.4.5 we have explained the meaning of direct and indirect globals, the base, offset and name of global TCG variables, and the special `env` TCG variable.

3.5.2 Storage of TCG variables

TCG variables, and several other data related to them, are stored in `*tcg_ctx`, which is a global pointer to a `TCGContext` instance. We have presented it in 3.3.1.

`tcg_ctx->temps` is an array of `TCGTemps`. All created `TCGTemp` instances are allocated in this array; the first created `TCGTemp` is allocated in `tcg_ctx->temps[0]`, the second in `tcg_ctx->temps[1]`, and so on.

`tcg_ctx->nb_temps` contains the number of allocated `TCGTemp`. `tcg_ctx->nb_globals` contains the number of allocated `TCGTemp` that represent global TCG variables.

As all global TCG variables must be allocated before the first constant or temporary TCG variable, `tcg_ctx->temps[0]` to `tcg_ctx->temps[tcg_ctx->nb_globals]` are all global TCG variables, while `tcg_ctx->temps[tcg_ctx->nb_globals]` to `tcg_ctx->temps[tcg_ctx->nb_temps]` are all constant or temporary TCG variables.

3.5.3 Conversions between C types for representing TCG variables

In Section 3.4, we have seen that the functions of the TCG API use types like `TCGv_i32`, `TCGv_i64`, `TCGv_vec`, etc. to represent TCG variables.

A C variable of type `TCGv_<TCG type>` contains the offset, in bytes, of the corresponding `TCGTemp` in `tcg_ctx->temps`.

For example, let suppose that the TCG API client calls the function `tcg_temp_new_i64` to create a new TCG variable, and that this TCG variable is allocated at `tcg_ctx->temps[i]`, for some integer `i`. In this situation, the `TCGv_i64` returned by the function contains the value `i * sizeof(TCGTemp)`.

The file *include/tcg/tcg.h* contains several util functions for converting a `TCGv_<TCG type>` variable to a `TCGTemp` pointer and vice versa. Functions for converting a `TCGv_<TCG type>` variable to a `TCGTemp` pointer have a name of the form `TCGv_<TCG type>_temp`. Functions for converting a `TCGTemp` pointer to a `TCGv_<TCG type>` variable have a name of the form `temp_TCGv_<TCG type>`.

3.5.4 TCG variables splitted into multiple TCGTemp instances

When a TCG variable of type i64 or i128 is created, and the host register size is smaller than the size of the TCG variable, QEMU internally splits the TCG variable into multiple `TCGTemp` instances. This mechanism only exists for integer types, vector TCG variables are never split.

As SymQEMU is intended to be used on a 64 bit host, we will only consider the case where the host has a 64 bits architecture. In this situation, only i128 TCG variables are split.

As constant and global TCG variables can not have type i128, in the related functions presented below we will elude the code that is responsible for splitting the TCG variable, as this code is never executed on a 64 bits host architecture.

Even for the case of a temporary TCG variable of type i128, we will not detail how the split works, because SymQEMU does not support spit TCG variables.

3.5.5 Function tcg_temp_alloc

The function `tcg_temp_alloc` allocates a new TCG variable. Here is its code, which is straightforward to read :

Code Source : tcg/tcg.c

```
1 static TCGTemp *tcg_temp_alloc(TCGContext *s)
2 {
3     int n = s->nb_temps++;
4
5     if (n >= TCG_MAX_TEMPS) {
6         tcg_raise_tb_overflow(s);
7     }
8     return memset(&s->temps[n], 0, sizeof(TCGTemp));
9 }
```

3.5.6 Function tcg_global_alloc

The function `tcg_global_alloc` is used internally for allocating global TCG variables. Here is its implementation :

Code Source : tcg/tcg.c

```
1 static TCGTemp *tcg_global_alloc(TCGContext *s)
2 {
3     TCGTemp *ts;
4
5     tcg_debug_assert(s->nb_globals == s->nb_temps);
6     tcg_debug_assert(s->nb_globals < TCG_MAX_TEMPS);
7     s->nb_globals++;
8     ts = tcg_temp_alloc(s);
9     ts->kind = TEMP_GLOBAL;
10
11    return ts;
12 }
```

The assert at line 5 is here check that no constant or temporary TCG variables are allocated

before a global TCG variable.

The function simply increments `tcg_ctx->nb_globals`, allocates a `TCGTemp` by calling `tcg_temp_alloc`, and sets its `kind` attribute to `TEMP_GLOBAL`.

3.5.7 Function tcg__temp__new__internal

The function `tcg_temp_new_internal` is responsible for generating a new temporary TCG variable.

Here is an extract of its code :

Code Source : tcg/tcg.c

```
1  TCGTemp *tcg_temp_new_internal(TCGType type, TCGTempKind kind)
2  {
3      TCGContext *s = tcg_ctx;
4      TCGTemp *ts;
5      int n;
6
7      /* [...] */
8
9      switch (type) {
10     case TCG_TYPE_I32:
11     case TCG_TYPE_V64:
12     case TCG_TYPE_V128:
13     case TCG_TYPE_V256:
14         n = 1;
15         break;
16     case TCG_TYPE_I64:
17         n = 64 / TCG_TARGET_REG_BITS;
18         break;
19     case TCG_TYPE_I128:
20         n = 128 / TCG_TARGET_REG_BITS;
21         break;
22     default:
23         g_assert_not_reached();
24     }
25
26     ts = tcg_temp_alloc(s);
27
28     /* [...] */
29
30     ts->kind = kind;
31
32     if (n == 1) {
33         ts->type = type;
34     } else {
35         ts->type = TCG_TYPE_REG;
```

```
36
37          for (int i = 1; i < n; ++i) {
38              TCGTemp *ts2 = tcg_temp_alloc(s);
39
40              /* [...] */
41          }
42      }
43      return ts;
44  }
```

The variable `n`, whose value is assigned in the switch statement of lines 9 - 24, is the number of `TCGTemp` instances that will be created for representing the TCG variable. `TCG_TARGET_REG_BITS` is expanded to the value 32 or 64, depending on whether the host architecture is 32 or 64 bits.

At line 17, we can see that an i64 TCG variable will be represented by one `TCGTemp` instance if the host is 64 bits, or split into two `TCGTemp` instances if the host is 32 bits. Similarly, at line 20, an i128 TCG variable will be split into two or four `TCGTemp` instances depending on whether the host architecture is 32 or 64 bits.

At line 26, `tcg_temp_alloc` is called to allocate a new `TCGTemp` instance.

At lines 28 - 33, in the simple case where no split happens, the `TCGTemp` instance attributes get the following values :

- `type` : The variable type, as requested by the API client.
- `kind` : `TEMP_TB`. This value comes from the argument `kind`, but the functions `tcg_temp_new_<TCG type>` always set this argument to `TEMP_TB`.

3.5.8 Function tcg__constant__internal

The function `tcg_constant_internal` is responsible for generating a new constant TCG variable.

Here is an extract of its code :

Code Source : tcg/tcg.c

```
1 TCGTemp *tcg_temp_new_internal(TCGType type, TCGTempKind kind)
2 {
3     TCGContext *s = tcg_ctx;
4     TCGTemp *ts;
5     int n;
6
7     /* [...] */
```

```
8
9       switch (type) {
10      case TCG_TYPE_I32:
11      case TCG_TYPE_V64:
12      case TCG_TYPE_V128:
13      case TCG_TYPE_V256:
14          n = 1;
15          break;
16      case TCG_TYPE_I64:
17          n = 64 / TCG_TARGET_REG_BITS;
18          break;
19      case TCG_TYPE_I128:
20          n = 128 / TCG_TARGET_REG_BITS;
21          break;
22      default:
23          g_assert_not_reached();
24      }
25
26      ts = tcg_temp_alloc(s);
27
28      /* [...] */
29
30      ts->kind = kind;
31
32      if (n == 1) {
33          ts->type = type;
34      } else {
35          ts->type = TCG_TYPE_REG;
36
37          for (int i = 1; i < n; ++i) {
38              TCGTemp *ts2 = tcg_temp_alloc(s);
39
40              /* [...] */
41          }
42      }
43      return ts;
44  }
```

`tcg_ctx->const_table` is an array of hash tables that store previously generated `TCGTemp` instances for constant TCG variables, keyed by the values of the TCG variables.

At lines 9 - 10, if a `TCGTemp` instance already exists for the requested value, this instance is returned instead of creating a new one.

Otherwise, the if statement at line 10 is entered and a new `TCGTemp` instance is created, whose attributes get the following values :

- `type` : The variable type, as requested by the API client.

- `kind` : `TEMP_CONST`
- `val` : the constant value

Finally, the newly created `TCGTemp` instance is inserted into the hash table and is then returned by the function.

3.5.9 Function tcg_global_reg_new_internal

Contrary to the other functions presented in this section, `tcg_global_reg_new_internal` is not called by a TCG API function.

The purpose of this function is to create the special `env` TCG variable, that is used to create global TCG variables, as explained in 3.4.5. This function is called at the beginning of the execution of QEMU.

Here is an extract of its code :

Code Source : tcg/tcg.c

```
1  static TCGTemp *tcg_global_reg_new_internal(TCGContext *s, TCGType type,
2                                              TCGReg reg, const char *name)
3  {
4      TCGTemp *ts;
5
6      /* [...] */
7
8      ts = tcg_global_alloc(s);
9      /* [...] */
10     ts->type = type;
11     ts->kind = TEMP_FIXED;
12     /* [...] */
13     ts->name = name;
14
15     /* [...] */
16
17     return ts;
18 }
```

This function simply allocates a new `TCGTemp` by calling `tcg_global_alloc`, and sets its attributes to the following values :

- `type` : `TCG_TYPE_PTR`. This value is determined by the argument `type` of the function, which in practice is always `TCG_TYPE_PTR`.
- `kind` : `TEMP_FIXED`

- `name` : `"env"`. This value is determined by the argument `name` of the function, which in practice is always `"env"`.

3.5.10 Function tcg_global_mem_new_internal

The function `tcg_global_mem_new_internal` is responsible for generating a new global TCG variable.

Here is an extract of its code :

Code Source : tcg/tcg.c

```
1  TCGTemp *tcg_global_mem_new_internal(TCGType type, TCGv_ptr base,
2                                       intptr_t offset, const char *name)
3  {
4      TCGContext *s = tcg_ctx;
5      TCGTemp *base_ts = tcgv_ptr_temp(base);
6      TCGTemp *ts = tcg_global_alloc(s);
7      int indirect_reg = 0;
8
9      switch (base_ts->kind) {
10     case TEMP_FIXED:
11         break;
12     case TEMP_GLOBAL:
13         /* We do not support double-indirect registers.  */
14         tcg_debug_assert(!base_ts->indirect_reg);
15         base_ts->indirect_base = 1;
16         /* [...] */
17         indirect_reg = 1;
18         break;
19     default:
20         g_assert_not_reached();
21     }
22
23     /* [...] */
24
25     ts->type = type;
26     ts->indirect_reg = indirect_reg;
27     /* [...] */
28     ts->mem_base = base_ts;
29     ts->mem_offset = offset;
30     ts->name = name;
31
32     /* [...] */
33
34     return ts;
35 }
```

At line 6, a call to `tcg_global_alloc` is performed to allocate the new `TCGTemp` instance.

In the switch statement of line 9, the functions checks if it must create a direct or an indirect global.

If the `base` TCG variable has kind `TEMP_FIXED`, this means that the base is the special `env` TCG variable and we are creating a direct global.

Otherwise, if the `base` TCG variable has kind `TEMP_GLOBAL`, the base is another global TCG variable and we are creating an indirect global. At line 15, the `indirect_reg` attribute of the base TCG variable is set, in order to indicate that this variable is the base of an indirect global.

The attributes of the new `TCGTemp` instance get the following values:

- `type` : The variable type, as requested by the API client.
- `kind` : `TEMP_GLOBAL` (this is set by the function `tcg_global_alloc`).
- `indirect_reg` : 1 if this TCG variable is an indirect global, 0 if it is a direct global.
- `mem_base` : the base TCG variable given by the API client (either the special variable `env` or another global TCG variable)
- `mem_offset` : the offset relative to the base, given by the API client.
- `name` : the name given as argument.

Chapter 4

SymQEMU

SymQEMU is a concolic executor based on QEMU. SymQEMU adds modifications to the source code of QEMU in order to make it perform a symbolic execution simultaneously to the normal concrete execution.

Those modifications mainly consists of instrumenting the TCG instructions, such that, for each TCG op created, additional TCG ops are added that perform an equivalent computation in the symbolic side of the computation.

The added instrumentation is entirely passive, i.e., it does not change the behavior of the normal execution performed by QEMU. The instrumentation simply observes the concrete execution in order to perform a parallel symbolic execution.

In this chapter, we will present several aspects of the implementation of SymQEMU.

In 4.1, we will explain how SymQEMU is used and we will give some definitions used in this chapter.

In 4.2, we will present an overview of the design of SymQEMU.

In 4.3, we will explain in details how TCG ops that perform computations on TCG variables are instrumented. For this, we will show as an example the code that SymQEMU adds to QEMU for instrumenting the *add_i32* TCG instruction.

In 4.4, we will explain how the instructions that perform access to memory are instrumented, and we will show extracts of the code of SymQEMU that handle this.

In 4.5, we will explain how the instructions that trigger the generation of path constraints are instrumented.

In 4.6, we will explain how SymQEMU creates and manages the symbolic version of TCG variables.

In 4.7, we will discuss some miscellaneous aspects of the implementation of SymQEMU.

Finally, in 4.8, we will briefly discuss the symbolic backend, which is a shared library used by SymQEMU, that provides several services for performing symbolic execution.

4.1 SymQEMU introduction

In the point of view of the user of SymQEMU, SymQEMU has the same behavior as QEMU, except that it generates examples of interesting program inputs during the execution. This is discussed in more details in 4.1.1.

In 4.1.2 we will present some definitions related to the implementation of SymQEMU.

4.1.1 SymQEMU usage

Let us discuss how SymQEMU is used from the point of view of its user.

In 3.1.2 we have seen that QEMU has a system mode and a user mode. Currently, SymQEMU only supports user mode.

In the point of view of its user, SymQEMU is run in the same way as QEMU. The command arguments are the same and running an executable with SymQEMU will produce the same outputs in the terminal as if it had been run with QEMU.

However, SymQEMU offers the following additional feature : during the execution, examples of program inputs that would have led to a different execution path are generated and saved in a dedicated directory. By default, the program input is considered to be *stdin*, i.e., the content of *stdin* is symbolized. It is also possible to consider as program input the content of a specific file. When the target program performs reads from this file, its content is symbolized.

The path of the directory for saving generated program inputs and the path of the file whose content should be symbolized are controlled with environment variables.

4.1.2 Definitions

Before discussing the details of the implementation of SymQEMU, let us introduce some definitions used in this chapter and the following ones.

We call *SymQEMU outputs* or *test cases* the examples of target program inputs generated by SymQEMU that would make the target program follow different execution paths.

We call an *instrumented instruction* an instruction whose corresponding TCG API function was modified in SymQEMU, such that additional TCG ops are generated for performing the symbolic execution.

We call *instrumentation TCG ops* the additional TCG ops generated for performing the symbolic execution.

We call *symbolic helper functions* a series of helper functions that SymQEMU adds to QEMU, and that are called at run time by the instrumentation TCG ops for performing the symbolic execution. Those helper functions are located in *accel/tcg/tcg-runtime-sym.c*

4.2 High level design of SymQEMU

In this section, we will give a high level overview of how SymQEMU is implemented.

SymQEMU adds several capabilities to QEMU, that allow it to store a symbolic version of any data stored in the guest memory or registers, and to perform symbolic computations on them simultaneously to the concrete computations. This is discussed in 4.2.1.

To implement those capabilities, SymQEMU uses a *symbolic backend*, that provides several useful services for performing a symbolic execution like the generation of symbol expressions, a symbolic representation of the program memory and the ability to add and solve path constraints. This is discussed in 4.2.2.

To implement those capabilities, QEMU is modified such that each TCG variable has a corresponding symbolic version, and TCG instructions are instrumented such that TCG ops are emitted to perform the symbolic side of the execution. This is discussed in 4.2.3.

At run time, the instrumentation TCG ops for an instruction will call a corresponding symbolic helper function, which itself performs appropriate calls to the symbolic backend. This is discussed in 4.2.4.

The instrumentation TCG ops do not check if the instruction operands are actually symbolic or not, the symbolic helper functions are responsible for performing those checks and create appropriate constant symbols when needed. This is discussed in 4.2.5.

4.2.1 Capabilities added to QEMU

Let us first discuss what capabilities SymQEMU must add to QEMU.

SymQEMU modifies QEMU to allow it to perform a symbolic execution simultaneously to the normal concrete execution. To do this, five main capabilities must be added to QEMU :

1. SymQEMU must be able to store a symbolic version of any data stored in guest memory.
2. SymQEMU must be able to store a symbolic version of any data stored in a TCG variable.

3. For any computation (like an add, sub, etc.) performed on a TCG variable associated to a symbolic value, a new symbolic value that represents the result of the computation must be generated and associated to the result TCG variable.

4. Any time a branch condition that depends on a symbolic data is encountered, the condition must be added to the set of path constraints.

5. Each time a condition is added to the set of path constraints, the current set of conditions must be used to perform a call to an SMT solver, in order to try to generate a new input that would lead the execution down a different path.

4.2.2 The symbolic backend

SymQEMU performs those actions with the help of the *symbolic backend*. The symbolic backend is a shared library that offers an API for performing tasks related to symbolic execution. It mainly offers the following services :

- A series of functions for generating symbol expressions.
- A function for pushing path constraints. When a path constraint is pushed, the backend performs a call to an SMT solver to try to generate a new input.
- A series of functions to store and load symbolic values from / to the *shadow pages*. The shadow pages are the symbolic version of the memory of the target program. It contains a mapping between memory addresses and symbolic values.

Beware that we also use the words *API* and *backend* to refer to the set of functions of QEMU that allow to generate TCG instructions, which is a totally different subject introduced in 3.4. In our explanations, we will do our best to make it clear when we are referring to the TCG API / backend or to the symbolic API / backend.

4.2.3 Implementation of the capabilities

The five capabilities listed in 4.2.1 are implemented in the following way :

For the first capability, each time a TCG variable is created, SymQEMU creates a second TCG variable for storing the symbolic version of the data. Note that, in SymQEMU, all concrete TCG variables have a corresponding symbolic TCG variable. The symbolic TCG variable may store a symbolic expression or may be empty (in which case its value is `NULL`). If the symbolic version of a TCG variable is `NULL`, we say that the TCG variable is concrete. Recall that in 1.6.2 we have seen that with concolic execution, a program data may or may

not be associated to a symbolic value. We will discuss in details the implementation of symbolic variables in 4.6.

For the second capability, for each store TCG op, SymQEMU adds instrumentation TCG ops that perform an equivalent store of the symbolic version of the source TCG variable to the shadow pages. Similarly, load TCG ops are instrumented to perform a load from the shadow pages to the symbolic version of the target TCG variable. We will discuss how this is implemented in 4.4

For the third capability, for each computation TCG op, SymQEMU adds instrumentation TCG ops that use the symbolic backend to generate a new symbolic expression that represents the result of the computation performed, and then store this new symbolic expression into the symbolic version of the TCG variable for the output of the computation. We will discuss how this is implemented in 4.3

Finally, for the fourth and fifth capabilities, for each branching TCG op SymQEMU adds instrumentation TCG ops that build a symbolic expression representing the branching condition and then pushes this expression as a path constraint to the backend.

4.2.4 Instrumentation of the TCG instructions

We see that a central aspect of the implementation of SymQEMU is the instrumentation of TCG instructions in order to trigger calls to the symbolic backend at run time. Let us give an overview of how this instrumentation is implemented.

For each instrumented TCG instruction, SymQEMU modifies the corresponding TCG API function such that, additionally to the normal TCG op, the API function generates instrumentation ops that will, at run time, perform a call to a *symbolic helper function*.

Symbolic helper functions are helper functions (as described in 3.4.10) that SymQEMU adds to QEMU, which are responsible for calling appropriate functions from the symbolic backend to perform symbolic execution. For (almost) each instrumented TCG instruction, there is a corresponding symbolic helper function that will be called at run time each time the instrumented TCG instruction is executed.

Figure 4.1 gives a high level overview of what happens at run time when an instrumented TCG op is executed.

- At step 1, the instrumentation TCG ops call the symbolic helper function that corresponds to the instrumented instruction. The helper function receives as argument the symbolic version of each input operand.
- At step 2, the helper function performs a call to the symbolic backend with the symbolic operands as argument.

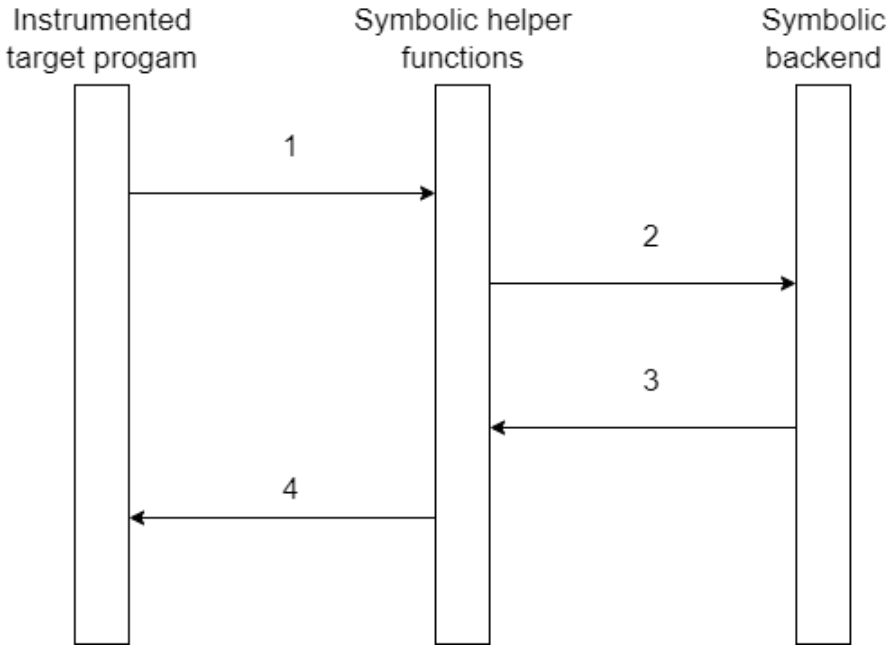

Figure 4.1: Sequence diagram for the execution of an instrumented intruction.

- At step 3, the symbolic backend returns a new child symbolic expression which represents the operation performed on the operands.
- Finally at step 4 the new symbolic expression is returned to the instrumentation TCG ops, and is saved into the symbolic version of the TCG variable for the output of the instrumented instruction.

4.2.5 Handling of concrete TCG variables

The overview that we have just presented is a bit of a simplification. As we have explained in 4.2.3, the symbolic version of a TCG variable can either contain a symbol expression or be `NULL`.

Let us present a more accurate version that explains how TCG variables that are not associated to a symbolic expression are managed.

Step 1 always happens, even if the symbolic version of TCG variables involved does not contain a symbolic expression. At run time, the TCG ops call the symbolic helper without checking whether the symbolic TCG vararіable has value `NULL` or not.

The symbolic helper is responsible for checking if the symbolic values received actually contain a symbolic expression or are `NULL`.

If all input operands of the instruction are concrete (i.e., their symbolic value is `NULL`), then the symbolic helper does not perform any call to the symbolic backend, i.e. steps 2 and 3 above are skipped. Instead, it simply returns `NULL`, which concretizes the output operand.

If the instruction has two input operands and one of them is concrete while the other is symbolic, then the symbolic helper function performs a call to the symbolic backend to create a new constant symbol for the concrete operand. To do this, the symbolic helper function needs to know the concrete value of the operand. For this reason, at step 1, the instrumentation TCG ops actually always passes the concrete value of the input operands to the helper function, additionally to their symbolic value.

After the creation of the concrete symbol, steps 2, 3 and 4 are performed normally.

4.3 Instrumentation of the add instruction

Now that we have seen the high level design of SymQEMU, let us show a concrete example of how a TCG instruction is instrumented.

In this section, we are going to show the code that SymQEMU adds to QEMU for the instrumentation of the *add_i32* TCG instruction.

To instrument this instruction, SymQEMU adds additional code to the corresponding TCG API function, such that an instrumentation TCG op is added at translation time that will call a symbolic helper function at run time. This is discussed in 4.3.1.

At run time, the symbolic helper function for the *add_i32* instruction is called. It performs the appropriate calls to the symbolic backend to generated a new symbol expression that represents the result of this addition. This is discussed in 4.3.2.

4.3.1 tcg_gen_add_i32

Let us first discuss the code added by SymQEMU that is executed at translation time.

`tcg_gen_add_i32` is the TCG API function that the TCG client calls at translation time to add an *add_i32* TCG op.

Here is its definition in SymQEMU

Code Source : include/tcg/tcg-op-common.h

```
1 static inline void tcg_gen_add_i32(TCGv_i32 ret, TCGv_i32 arg1, TCGv_i32 arg2)
2 {
3     gen_helper_sym_add_i32(tcgv_i32_expr(ret),
4                            arg1, tcgv_i32_expr(arg1),
```

```
5                           arg2, tcgv_i32_expr(arg2));
6
7     tcg_gen_op3_i32(INDEX_op_add_i32, ret, arg1, arg2);
8 }
```

The function call at lines 3 - 5 was added in SymQEMU. It adds a TCG op that will call the helper function `helper_sym_add_i32` at run time.

`tcgv_i32_expr` is a util function added in SymQEMU that receives a TCG variable as argument and returns the symbolic version of the TCG variable (we will discuss how this is implemented in 4.6.2). Note that in the actual source code this line is generated with the expansion of a macro created by the authors of SymQEMU. Here for simplification we show the code generated after the macro expansion.

As we can see, both the concrete and the symbolic version of each input operand is passed to the symbolic helper. The return value of the helper function will be written in the symbolic version of `ret`. Recall that, as discussed in 3.4.10, the first argument of `gen_helper_sym_add_i32` is the TCG variable that will receive the value returned by the helper function at run time.

4.3.2 helper__sym__add__i32

Now let us examine the helper function `helper_sym_add_i32`, which will be called at run time.

Code Source : accel/tcg/tcg-runtime-sym.c

```
1 void* helper_sym_add_i32(uint32_t arg1, void *arg1_expr, uint32_t arg2, void
  ↪  *arg2_expr) {
2         return helper_sym_add_i64(arg1, arg1_expr, arg2, arg2_expr);
3 }
4
5 void *helper_sym_add_i64(uint64_t arg1, void *arg1_expr, uint64_t arg2, void
  ↪  *arg2_expr) {
6     if (arg1_expr == NULL && arg2_expr == NULL) {
7         return NULL;
8     }
9
10    if (arg1_expr == NULL) {
11        arg1_expr = _sym_build_integer(arg1, _sym_bits_helper(arg2_expr));
12    }
13
14    if (arg2_expr == NULL) {
15        arg2_expr = _sym_build_integer(arg2, _sym_bits_helper(arg1_expr));
16    }
17
```

```
18     return _sym_build_add(arg1_expr, arg2_expr);
19 }
```

Here again, the above code is actually generated by a series of expansions of macros created by the authors of SymQEMU, in order to factorize repetitive code, as many symbolic helper functions are very similar.

`arg1` and `arg2` are the concrete version of the instruction operands. `arg1_expr` and `arg2_expr` contain their respective symbolic versions. In SymQEMU, symbol expressions are represented as `void*` pointers. A symbol expression pointer points to a C++ object created and managed by the symbolic backend. The code that SymQEMU adds to QEMU has no need to care about the meaning of those pointers, they are just received and sent from / to the symbolic backend and stored in the symbolic version of TCG variables.

As we can see, `helper_sym_add_i32` actually delegates to `helper_sym_add_i64`. It is only here to perform casting of `arg1` and `arg2` from 32 bits to 64 bits.

At lines 6 - 8, if both input operands are concrete, we immediately return and concretize the output operand by returning `NULL`.

At lines 10 - 16, if only one operand is concrete, a constant symbol is generated to represent it. `_sym_build_integer` is a function from the symbolic backend API that creates a constant symbol for a given value and size in bits. `_sym_bits_helper` is a function from the symbolic backend that returns the size in bits of the value represented by a symbolic expression.

Finally at line 18, a new symbol expression is generated that represents the addition of the two operands. `_sym_build_add` is a function from the symbolic backend that generates a new symbol expression for an addition. This new symbol expression is returned and will be assigned to the symbolic version of the output operand of the instruction.

4.4 Load and store instructions

Now that we have seen how TCG instructions that perform computations are instrumented, let us discuss how stores and loads to memory are instrumented.

As discussed in 3.4.8, the TCG API offers functions for accessing the guest memory and functions for accessing host memory.

In a similar way to the *add_i32* TCG instruction, the TCG API functions for the load and store instructions are modified in SymQEMU, such that TCG ops for calling a symbolic helper function at run time ar injected.

The symbolic helper function for host memory stores performs an equivalent store to the shadow pages of the symbolic backend. This is discussed in 4.4.1.

Similarly, the helper function for host memory loads performs an equivalent load from the shadow pages of the symbolic backend. This is discussed in 4.4.2.

The helper functions for guest memory loads and stores are similar, but they perform a guest to host address translation before accessing the shadow pages. This is discussed in 4.4.3.

4.4.1 helper_sym_store_host

Let us discuss how symbolic stores to host memory are performed.

The function `helper_sym_store_host` is the helper function that is called at run time when a store to host memory happens.

Here is its definition :

Code Source : accel/tcg/tcg-runtime-sym.c

```
1 void helper_sym_store_host(void *value_expr, void *addr,
2                             uint64_t offset, uint64_t length)
3 {
4     _sym_write_memory((uint8_t*)addr + offset, length, value_expr, true);
5 }
```

`value_expr` is the symbolic version of the TCG variable stored to memory. `addr` is the host address for the store, and `offset` is an offset from this address (in TCG, all store / load instructions have an `offset` immediate operand). `length` is the size of the data stored to memory.

This function simply stores the symbolic version of the TCG variable into the shadow pages of the symbolic backend. `_sym_write_memory` is a function from the symbolic backend API for writing into the shadow pages.

4.4.2 helper_sym_load_host_i32 and helper_sym_load_host_i64

Now let us discuss how symbolic loads from host memory are performed.

The functions `helper_sym_load_host_i32` and `helper_sym_load_host_i64` are the helper functions called at run time when a load from host memory is performed.

Here is their definition :

Code Source : accel/tcg/tcg-runtime-sym.c

```
1 static void *sym_load_host_internal(void *addr, uint64_t offset,
2                                     uint64_t load_length, uint64_t result_length)
3 {
```

```
4     void *memory_expr = _sym_read_memory(
5         (uint8_t*)addr + offset, load_length, true);
6
7     if (load_length == result_length || memory_expr == NULL)
8         return memory_expr;
9     else
10        return _sym_build_zext(memory_expr, (result_length - load_length) * 8);
11 }
12
13 void *helper_sym_load_host_i32(void *addr, uint64_t offset, uint64_t length)
14 {
15     return sym_load_host_internal(addr, offset, length, 4);
16 }
17
18 void *helper_sym_load_host_i64(void *addr, uint64_t offset, uint64_t length)
19 {
20     return sym_load_host_internal(addr, offset, length, 8);
21 }
```

As we can see, they are just wrappers for the internal function `sym_load_host_internal`. Let us examine this internal function.

`addr` is the host memory address for the load, and `offset` is an offset from it.

`load_length` is the length of the data that is red in the shadow pages, while `result_length` is the length of the returned data. Those lengths can be different, because the TCG API allows for example to read one byte from memory and to store it in a 64 bits TCG variable (in this case the value red is zero extended). The symbolic helper function must imitate this behavior for the symbolic version of the execution.

A lines 4 - 5, a read from the shadow pages is performed. `_sym_read_memory` is a function from the symbolic backend API that allows to read from the shadow pages.

Then at lines 7 - 9, if the value red should be zero extended to match `result_length`, we do so. `_sym_build_zext` is a function from the symbolic backend API that generates a new symbol expression representing a zero extension operation.

4.4.3 Symbolic helper functions for guest memory accesses

We have discussed the symbolic helper functions for host memory accesses. There exist similar symbolic helper functions for guest memory accesses.

The code of those functions is similar to the helper functions for host memory access, except for the following : before accessing the shadow pages, the load / store address is translated from a guest address to a host address (see 3.4.8). This is done by calling the appropriate function from QEMU.

As a consequence, the shadow pages only store symbolic data keyed by host address. Without this design choice there would be a risc of collision between host and guest data in the shadow pages.

4.5 Instructions that trigger generation of path constraints

We have seen how instructions that perform computations and memory accesses are instrumented. Those instructions lead to the manipulation and creation of symbol expressions, but they do not trigger the generation of path constraints. Let us now discuss TCG instructions that trigger the generation of path constraint.

Those instructions are *brcond*, *setcond* and *movcond*. To understand the explanations in this section, we invite the reader to read their definition in the TCG documentation [4].

Here it is important to know that there is a difference between the theoretical functionning of concolic execution as we have described it in 1.6.3.2 and the actual implementation of SymQEMU.

In 1.6.3.2 we have explained that a path constraint is generated when a branching instruction whose condition depends on a symbolic value is executed. A conditional branching in TCG is indicated with the instruction *brcond*.

However, *setcond* and *movcond* should also trigger the generation of path constraints, as they are a high level equivalent of a conditional branching combined to a *mov* instruction. For example, when a *movcond* is executed, in the point of view of symbolic execution there are two possible paths, one where the condition is true and the *mov* is executed, and one where the condition is false and the *mov* is not executed. As a consequence, a path constraint must be generated accordingly.

At run time, when a *setcond* instruction is executed, a corresponding symbolic helper function is called. This helper function generates a new symbol expression that represents the condition of the *setcond*, and pushes this condition as a path constraint. This is discussed in 4.5.1.

There is no symbolic helper function for *brcond* and *movcond*. Instead, for those instructions, SymQEMU adds an additional *setcond* instruction, that has no effect on the concrete side of the execution but that will result in an appropriate path constraint being pushed at run time for the symbolic side of the execution. This is discussed in 4.5.2.

4.5.1 setcond

Let us present the symbolic helper functions `helper_sym_setcond_i64` and `helper_sym_setcond_i32`, which are called at run time when a *setcond* instruction is performed.

Code Source : accel/tcg/tcg-runtime-sym.c

```
1  static void *sym_setcond_internal(/* [...] */,
2                                    uint64_t arg1, void *arg1_expr,
3                                    uint64_t arg2, void *arg2_expr,
4                                    int32_t comparison_operator, uint64_t is_taken,
5                                    uint8_t result_bits)
6  {
7      /* [...] */
8
9      void *condition_symbol = build_and_push_path_constraint(/* [...] */, arg1_expr,
   ↪   arg2_expr, comparison_operator, is_taken);
10
11     assert(result_bits > 1);
12     return _sym_build_zext(_sym_build_bool_to_bit(condition_symbol),
13                            result_bits - 1);
14 }
15
16 void *helper_sym_setcond_i32(/* [...] */,
17                              uint32_t arg1, void *arg1_expr,
18                              uint32_t arg2, void *arg2_expr,
19                              int32_t comparison_operator, uint32_t is_taken)
20 {
21     return sym_setcond_internal(
22             /* [...] */, arg1, arg1_expr, arg2, arg2_expr, comparison_operator,
   ↪   is_taken, 32);
23 }
24
25 void *helper_sym_setcond_i64(/* [...] */,
26                              uint64_t arg1, void *arg1_expr,
27                              uint64_t arg2, void *arg2_expr,
28                              int32_t comparison_operator, uint64_t is_taken)
29 {
30     return sym_setcond_internal(
31             /* [...] */, arg1, arg1_expr, arg2, arg2_expr, comparison_operator,
   ↪   is_taken, 64);
32 }
```

As we can see, `helper_sym_setcond_i64` and `helper_sym_setcond_i32` are just wrappers for `sym_setcond_internal`. Let us discuss `sym_setcond_internal`.

`arg1` and `arg2` are the concrete version of the input operands. `arg1_expr` and `arg2_expr` are the symbolic version of the input operands. `comparison_operator` is an enum value that indicates the comparison operator applied between the two input operands (equal, signed / unsigned less than, signed / unsigned greater than, etc.). `is_taken` indicates if the condition was true during the concrete execution. `result_bits` is the size in bits of the output operand.

At line 9, `build_and_push_path_constraint`, which we present below, is called. Its return

value is a symbol expression that represents the boolean value for the condition.

At line 12, calls to the symbolic backend are performed to get a symbol expression that represents the boolean value converted to an integer of size `result_bits`.

This symbol expression is then returned and will be assigned to the symbolic version of the output operand of the *setcond* instruction.

Let us now present the function `build_and_push_path_constraint`.

This function is responsible for generating a symbol expression that represents the result of the comparison, and for pushing a path constraint associated to this symbol expression.

Here is its definition :

Code Source : accel/tcg/tcg-runtime-sym-common.c

```
1 void *build_and_push_path_constraint(/* [...] */, void *arg1_expr, void *arg2_expr,
  ↪  uint32_t comparison_operator, uint8_t is_taken){
2     void *(*handler)(void *, void*);
3     switch (comparison_operator) {
4         case TCG_COND_EQ:
5             handler = _sym_build_equal;
6             break;
7         case TCG_COND_NE:
8             handler = _sym_build_not_equal;
9             break;
10        case TCG_COND_LT:
11            handler = _sym_build_signed_less_than;
12            break;
13
14        /* [...] */
15
16        default:
17            g_assert_not_reached();
18    }
19
20    void *condition_symbol = handler(arg1_expr, arg2_expr);
21    _sym_push_path_constraint(condition_symbol, is_taken, /* [...] */);
22
23    return condition_symbol;
24 }
```

Lines 3 - 17 are a large switch statement that selects the appropriate symbolic backend function for generating the expression for the condition. For simplicity, we have eluded the end of this switch statement.

At line 20, the appropriate function from the symbolic backend is called to genrate a new symbol expression representing the result of the comparison.

At line 21, the symbolic backend is called to push a new path constraint, whose associated

symbol expression is the symbol expression for the comparison. To be able to generate an appropriate new input example that would have made the comparison give an opposite result, the symbolic backend needs to know if the condition was true or false during the concrete execution. This information is given with the `is_taken` argument.

Finally, the generated symbol for the condition is returned.

4.5.2 brcond and movcond

Now that we have seen how *setcond* is instrumented, let us discuss the two other instructions that trigger the generation of path constraints : *brcond* and *movcond*

Instead of using a symbolic helper function for *brcond* and *movcond*, the instrumentation for those instructions uses the following trick : At translation time, a *setcond* instruction is added, with the same condition as the condition of the instrumented *brcond* or *movcond* instruction.

The concrete result of the added *setcond* is never used, thus this *setcond* has no effect on the concrete execution. However, as the function for generating the *setcond* instruction is instrumented, at run time the instrumentation code of the *setcond* will push a path constraint as explained in 4.5.1.

This approach simplifies the implementation of SymQEMU, as it avoids having additional symbolic helper functions for *brcond* and *movcond*.

4.6 TCG variables

We have seen that the TCG instructions are instrumented such that additional TCG ops are emitted that perform equivalent computations on the symbolic versions of the TCG variables. Now, let us discuss how those symbolic TCG variables are implemented.

Recall that the implementation of TCG variables in QEMU has been discussed in 3.4.1 and 3.5.

For each concrete TCG variable created, a corresponding symbolic TCG variable is automatically created. The `TCGTemp` instance for the symbolic variable immediately follows the `TCGTemp` instance for the concrete variable in host memory. This is discussed in 4.6.1.

Thanks to this design, one can access the symbolic version of a TCG variable simply by accessing the next `TCGTemp` instance in memory. This is discussed in 4.6.2.

The symbolic versions of global TCG variables are associated to elements of an array named `env_exprs` that SymQEMU adds to QEMU. This is discussed in 4.6.3.

This design has non-obvious implications regarding the symbolic version of the guest regis-

ters. In particular, depending on the behavior of the TCG API client code, two alternative versions of a symbolic guest register can exist at the same time, which may be a problem in SymQEMU. This is discussed in 4.6.4.

4.6.1 Creation of symbolic TCG variables

In SymQEMU, for each concrete TCG variable created, a second TCG variable is created for storing the symbolic version of the concrete TCG variable value.

This is implemented by performing the following main modifications to QEMU :

- The TCG backend functions for creating TCG variables (discussed in 3.5.7, 3.5.8 and 3.5.10) are modified such that, each time a normal TCG variable is created, a second symbolic TCG variable is created. As a consequence, the symbolic TCG variable immediately follows the corresponding conrete one in the array `tcg_ctx->temps` (`tcg_ctx` is presented in 3.3.1 and `tcg_ctx->temps` is discussed in 3.5.2).

- A new member `symbolic_expression` is added to the struct `TCGTemp` (`TCGTemp` is presented in 3.5.1). This member is a boolean whose value is 1 if the TCG variable is symbolic, 0 if it is concrete.

A symbolic TCG variable always has type `ptr`, because, as discussed in 4.3.2, it contains a pointer to a C++ object representing the symbol expression.

A symbolic TCG variable always has the same kind (temporary, constant, global) as its corresponding concrete TCG variable.

4.6.2 Finding the symbolic version of a TCG variable

As the symbolic version of a TCG variable always immediately follows its concrete version in `tcg_ctx->temps`, it is easy to access the symbolic version given a pointer to the concrete version.

For a pointer `TCGTemp *p` to a `TCGTemp` instance that represents a concrete TCG variable, `p + 1` (`p` increased by the size of `TCGTemp`) is a pointer to the `TCGTemp` instance that represents the corresponding symbolic variable.

SymQEMU provides a series of util functions that allow to access the symbolic version of a TCG variable, given the concrete version of a TCG variable. Internally, those functions simply increase a pointer as we have just seen.

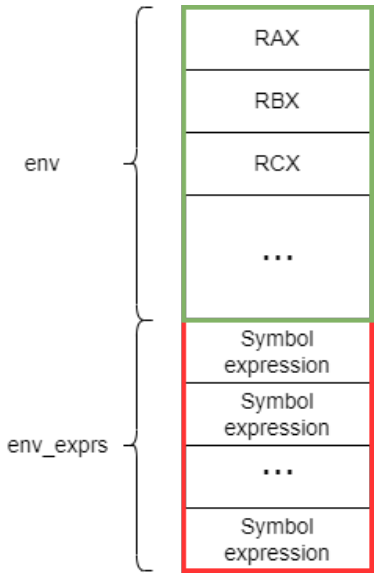

Figure 4.2: Extension of `CPUArchState`.

4.6.3 Symbolic version of a global TCG variable

Let us now discuss how symbolic global TCG variables are managed.

In 3.4.5 and in 3.5.9, we have seen that a global TCG variable is always associated to a guest register that is stored in the `CPUArchState` instance. We have also seen that, in practice, the global TCG variable is associated to an offset from the beginning of the `CPUArchState` instance, and that reads and writes to the global TCG variable will be translated to read and write to the corresponding offset of the `CPUArchState` instance.

As the symbolic version of a concrete global TCG variable is itself a global TCG variable, it has to be associated to an offset from the `CPUArchState` instance, and reads and writes performed to it will be translated to reads and writes to this offset of the `CPUArchState` instance.

As a solution, SymQEMU adds an "extension" of `CPUArchState` named `env_exprs`, that is located right after the `CPUArchState` instance in memory. `env_exprs` is an array of `void` pointers. Each global symbolic TCG variable is associated to an element of `env_exprs`. When the global symbolic TCG variable is created, the offset associated to it will be the offset from the start of the `CPUArchState` instance to the appropriate element of `env_exprs`. Note that this offset is larger than the size of `CPUArchState`, which allows to reach `env_exprs`.

The `CPUArchState` instance is a member named `env` of the `ArchCPU` struct declared in *target/<guest architecture>/cpu.h*. SymQEMU adds `env_exprs` right after it in `ArchCPU`.

Figure 4.2 illustrates how the `CPUArchState` instance is extended, assuming an x86 guest architecture. `env` is the `CPUArchState` instance and `env_exprs` is the array of void pointers added right after it in memory.

4.6.4 Location of the symbolic version of a guest register

Now that we have seen how symbolic global TCG variables are managed, let us discuss what consequences this design has on the symbolic version of the guest registers.

In 3.4.9, we have seen that the TCG API client (i.e., the QEMU code that performs the translation from guest instructions to TCG ops) can manipulate guest registers in two different methods :

1. The first method is to have a global TCG variable associated to the location of the guest register in the `CPUArchState` instance and to emit TCG ops that manipulate the global TCG variable.
2. The second method is to emit load / store TCG ops to the host memory, that will directly manipulate the guest register in the `CPUArchState` instance.

Both methods will result to reads / writes to the same location : the guest register in the `CPUArchState` instance.

Now, an interesting question is : where are stored the symbolic versions the guest registers ?

1. When the TCG API client manipulates registers with method 1, the simultaneous symbolic computation will be performed on the symbolic version of the global TCG variables, and thus the symbolic expressions for the guest register will be red / written in `env_exprs`.
2. However, when the TCG API client manipulates registers with method 2, the simultaneous symbolic execution will perform loads / stores from / to the shadow pages. The memory address associated to those shadow pages accesses will be the host address of the guest register in the `CPUArchState` instance.

As a consequence, if the TCG API client mixes both methods for manipulating the same guest register, two different versions of the symbolic version of the guest register will exist independently of each other, which will lead to an incorrect symbolic execution.

We do not know if this situation actually happens. Most probably the authors of SymQEMU were aware of this problem and checked that it never happens, but it seems important to us that contributors of SymQEMU are aware of this situation.

4.7 Miscellaneous information about the implementation of SymQEMU

We have discussed the main modifications that SymQEMU adds to QEMU. In this section, we give some additional miscellaneous information about some implementation aspects of SymQEMU.

We will discuss the fact that some TCG instructions are indirectly instrumented (4.7.1).

We will also see that some guest instructions may not be instrumented, either because the corresponding TCG instructions have not been instrumented yet in SymQEMU (4.7.2), or because those guest instructions are translated using helper functions (4.7.3).

Finally, we will briefly discuss how the original symbols are created in SymQEMU, when the target program reads its input (4.7.4).

4.7.1 Indirectly instrumented instructions

In QEMU, some TCG instructions are optionally supported, depending on the host.

The TCG API functions that generate those instructions check at translation time if the instruction is supported. If not, they emit a combination of other supported TCG ops that have a behavior equivalent as the original instruction.

In SymQEMU, instead of instrumenting those optionally supported instructions, the code of the TCG API function is modified such that the substitutes TCG ops are always used. As the substitutes TCG instructions are instrumented, this approach reduces the human work needed for implementing SymQEMU.

4.7.2 Non instrumented TCG instructions

In SymQEMU, most TCG instructions are instrumented, either directly or indirectly.

However, some instructions are still missing instrumentation. The file *accel/tcg/tcg-runtime-sym.h* contains a comment indicating a list not yet instrumented instructions. In particular, when we began this bachelor work, the vector TCG instructions were not instrumented.

As a consequence, when those instructions are encountered with symbolic operands, the symbolic execution behavior is incorrect.

4.7.3 Helper functions are not instrumented

Another thing to know about the design of SymQEMU is that the guest instructions that are translated to a call to a helper function instead of a TCG instruction are not instrumented.

The QEMU code that performs the translation from guest architecture instructions to TCG may contain helper functions for guest instructions that can not easily be translated in TCG. When one of those instructions is encountered, the translation code adds a TCG op that will trigger the execution of the helper function at run time, and the helper function simulates the behavior of the complicated instruction.

Due to the design of SymQEMU that consists of instrumenting the TCG instructions, those helper functions are not instrumented.

`tcg_gen_callN` is the internal TCG backend function that is called at translation time when a TCG op for calling a helper function is added. To avoid unwanted side effects related to non instrumented helper functions, SymQEMU adds a bit of code in `tcg_gen_callN` that checks if the helper function that will be called at run time is a symbolic helper function from SymQEMU or is another helper function from QEMU. If it is another helper function from QEMU, a TCG op is injected to concretize at run time the TCG variable that will receive the result of the helper function.

An example of such unwanted side effect that could happen if the destination of a helper function is not concretized is discussed in 5.4.

4.7.4 Interception of read system calls

In order to generate the initial symbols for the program input, SymQEMU modifies the QEMU code that handles `read` system calls. When the target program reads bytes from stdin or from a specific file, depending on the configuration of SymQEMU, the corresponding area in the shadow pages are assigned to symbols.

This is done by modifying the functions `do_openat` and `do_syscall1` in *linux-user/syscall.c*

4.8 Symbolic backend

Now that we have covered the modifications that SymQEMU adds to the code of QEMU, let us give some more information about the symbolic backend.

The symbolic backend used by SymQEMU comes from SymCC, another concolic executor created at EURECOM that we have presented in 2.1.10. SymCC itself reuses code from QSYM, a previous concolic executor presented in 2.1.9. SymCC and QSYM are written in C++.

When SymCC is compiled, one of its build outputs is a shared library that contains the symbolic backend. This shared library is a dependence of SymQEMU.

As we have seen in the explanations of the present chapter, the symbolic backend exposes an API where the name of each API function begins with `_sym`. Those API function allow, among other things, to generate symbol expressions. In the point of view of the client of the symbolic backend API, those symbol expressions are represented as `void` pointers that are given to and received from the API functions. The client of the symbolic backend API has no understanding of the meaning of those pointers.

In the symbolic backend, a symbol expression stores several data, including :

- The operation that this expression represents (addition, subtraction, zero extension, ...)
- If the expression is a constant symbol, the constant value
- The parents of the expression (we have defined what a parent of an expression is in 1.3.5)

Chapter 5

Improving SymQEMU

Our first contribution to SymQEMU was to port it to QEMU 8, the latest stable version of QEMU.

Unexpectedly, porting SymQEMU to QEMU 8 made it generate outputs of significantly lower quality, on a simple test target program. In agreement with the professors that supervised this project, we decided to focus on discovering the causes of this degradation in outputs quality, instead of carrying out the other tasks originally planned.

Understanding the causes of the differences in the results outputted by the original SymQEMU and our new version happened to be an interesting task, that required us to add a debugging feature to SymQEMU and to use a systematical approach for comparing the symbolic executions performed by the two versions of SymQEMU.

In 5.1, we will describe the approach that we used to merge SymQEMU with the version 8.1 of QEMU, and how we took care or preserving the git history while doing it. We will also explain in details some decisions and changes that we made while doing this port.

In 5.2, we will explain the methodology that we used to check if our new version of SymQEMU had a correct behavior, and the result of this validation. As we will see, we compared the outputs generated by the original SymQEMU to those generated by our new SymQEMU and we discovered that, when running it on a simple program that calls the `asprintf` function, our new SymQEMU gave clearly incorrect results.

In 5.3, we will explain the approach that we used to discover the root cause of this problem. We will see that we added a feature to the symbolic backend that makes it output a *symbolic trace*, which allowed us to compare the behaviors of the old and the new SymQEMU in a systematical way, and to identify the instructions that the two versions of SymQEMU did not handle identically. Thanks to this analysis, we identified that the problematic instructions were SSE instructions.

Finally, in 5.4, we will explain why the old and the new SymQEMU do not handle SSE instructions identically. We will see that, actually, both the new and the old SymQEMU have an incorrect behavior when they encounter SSE instructions, but that this behavior became worse in the new version of SymQEMU, which was the cause of the completely different results that we got on the `asprintf` test program.

5.1 Port of SymQEMU to QEMU 8

The first contribution that we made to SymQEMU is to port it to QEMU 8.1, which is the most recent stable version of QEMU at the time of writing.

The original SymQEMU is based on QEMU 4.1. Recently, a port to QEMU 7 was performed for a project internal to EURECOM, but this version was never actually used, and it has been only lightly tested.

In this section, we will give an overview of the work we performed for porting SymQEMU to QEMU 8.1.

In 5.1.1, we will first describe the methodology that we used to perform this port. We will also explain how we made sure that our port preserved the git history, such that the lines of code of SymQEMU get correctly attributed to their original authors.

Then, we will discuss some noteworthy aspects of the adaption that we have done. In 5.1.2, we present a problem that we encountered that is related to chains of inclusion and macro expansions, and the solution that we chose. In 5.1.3 we discuss another change, which is relative to the way macro expansions are managed for symbolic helper functions. In 5.1.4 we discuss the problematic consequences of the existence of a new type of TCG variable, and the temporary solution that we used. Finally, in 5.1.5 we discuss another problem related to includes and macros, and explain our solution.

5.1.1 Methodology for porting SymQEMU

We performed a *git diff* between QEMU 7 and the version of SymQEMU based on QEMU 7, in order to obtain an exhaustive list of the modifications that SymQEMU adds to QEMU.

For each of those modifications, we determined if the corresponding code area of QEMU had been updated between QEMU 7 and QEMU 8. For code areas that were identical between both versions of QEMU, we simply applied the same modifications to QEMU 8. For code areas that were different, we analyzed how the code of QEMU had been updated, and we adapted the SymQEMU code accordingly.

We incorporated the changes progressively in QEMU 8, and at each step we controlled that the program compiled without errors and could be run on a simple test program without

crashing. When a modification to QEMU 8 triggered a compilation error or a crash, we investigated the cause and fixed it.

In the perspective of the git history, we placed this new version of SymQEMU into a *merge commit*, whose first parent is the commit of QEMU 8.1, and whose second parent is the commit of SymQEMU based on QEMU 7.

Thanks to this, we did not "appropriate" the code of SymQEMU in the perspective of the git history. When performing a *git blame* on our new SymQEMU, lines of code that come from the original SymQEMU are correctly attributed to their author, Sebastian Poeplau, and the SymQEMU code that was added or modified for the port to QEMU 7 are correctly attributed to Aurélien Hernandez, who did this port. Only the parts of SymQEMU that we have modified or added for the port to QEMU 8 are attributed to us.

Another advantage of preserving the continuity of the git history is that it allows to easily access commit messages associated to the different parts of SymQEMU. The original author of SymQEMU wrote long and informative commit messages, which can still be found through a *git blame* on the corresponding parts of our new SymQEMU version.

5.1.2 Symbolic helper functions for guest memory load / store

Now that we have presented the global methodology used to perform the port, we are going to discuss some specific technical aspects of it.

Let us first discuss a problem that we encountered in relation with include files and macro expansions.

The functions `sym_store_guest_i32`, `sym_store_guest_i64`, `sym_load_guest_i32` and `sym_load_guest_i64` are the symbolic helpers called at run time to perform loads and stores from / to the shadow pages, for guest memory accesses.

The `i32` / `i64` in the functions name refers to the size of the loaded / stored value, not to the size of the address.

Those functions receive as argument the target guest memory address. At run time, the TCG variable that corresponds to this argument has type i32 or i64, depending on whether the guest architecture is 32 or 64 bits.

To make the code generic, the old SymQEMU declares the argument of the helper function with `dh_alias_tl`, in the symbolic helper functions header file. `dh_alias_tl` is a macro that is expanded to i32 or i64, depending on whether QEMU is compiled for a 32 or 64 bits guest architecture.

As a consequence, the C files that use the symbolic helper functions need to have the macro `dh_alias_tl` defined, as they include the header file for the symbolic helpers. Those C files are the ones that contain the TCG API functions, where SymQEMU injects TCG ops for

calling the symbolic helper functions.

The macro `dh_alias_tl` is defined in the following preprocessor directives :

Code Source : include/exec/helper-head.h

```
#ifdef NEED_CPU_H
# ifdef TARGET_LONG_BITS
#  if TARGET_LONG_BITS == 32
#   define dh_alias_tl i32
    /* [...] */
#  else
#   define dh_alias_tl i64
    /* [...] */
#  endif
# endif
# /* [...] */
#endif
```

We can see that, in order to have `dh_alias_tl` defined in it, a C file must satisfy those two conditions :

- `TARGET_LONG_BITS` must be defined. `TARGET_LONG_BITS` is defined in *target/<guest architecture>/cpu-param.h*, thus a C file that uses `dh_alias_tl` mus include *target/<guest architecture>/cpu-param.h*.
- `NEED_CPU_H` must be defined. The build system defines `NEED_CPU_H` for some C files and not for others. `NEED_CPU_H` probably means "needs CPU header" in reference to *target/<guest architecture>/cpu-param.h*, but we do not have a deep understanding of the design choices behind it.

In the old SymQEMU, `NEED_CPU_H` was defined by the build system for the C files that use the symbolic helper functions. Additionally, those files included *target/<guest architecture>/cpu-param.h* indirectly, through a chain of other header files inclusion.

In QEMU 8 however, those C files do not have `NEED_CPU_H` and they do not include *target/<guest architecture>/cpu-param.h* anymore. As a consequence, including the symbolic helpers header file in those C files triggers a compilation error.

As we can see, in QEMU there is a complex system of include chains and macros, which is, to the best of our knowledge, not documented anywhere. It seemed to us that adding modifications to this system without having a good understanding of it would be a bad idea, as it could lead to unexpected side effects and to problems difficult to debug.

Due to lack of time, we have decided to not investigate more on `dh_alias_tl`, and we replaced it with i64. For the reasons explained below, we think that this simple solution is acceptable.

This change of course does not cause any problem if the guest architecture is 64 bits, as in this case `dh_alias_tl` was already replaced with i64. Until now, SymQEMU has mostly been used to test 64 bits x86 programs, and our new version of SymQEMU does not cause problem for this use case.

The consequences of our change when compiling SymQEMU for a 32 bits guest architecture are not clear to us. In this situation, the load / store helper functions will receive the content of a 32 bits TCG variable, in an argument that has type `uint64_t`.

This may not cause any problem, and just lead to the cast of the 32 bits unsigned integer into a 64 bits unsigned integer.

If this change happens to cause problems with 32 bit guests, a simple future improvement to fix it would be to have two versions of the load / store helper functions, one that expects a 32 bits address and the other that expects a 64 bits address. Then, the code in the TCG backend that adds the instrumentation TCG ops would check if the TCG variable that holds the load / store address has type i32 or i64, and would call the appropriate load / store helper function.

5.1.3 Macro expansion for symbolic helpers

Another adaptation that we performed was related to macro expansion for helper functions.

We have presented in 3.4.10 and in 3.4.11 the design of includes and macro expansions used in QEMU 8 for automatically generating the code that makes helper functions work.

As this system was simpler in QEMU 7, we adapted the symbolic helper functions to this new system.

5.1.4 TCG variables of type i128

Let us now discuss a potential problem in SymQEMU based on QEMU 8 that is due to the new *i128* TCG variable type.

We have seen in 3.4.2 that there exists a type of TCG variable named i128, that represents 128 bits integers. In 3.5.4, we have also discussed the fact that those variables are internally split into two `TCGTemp` instances by the TCG backend.

The i128 TCG variables were introduced in QEMU 8. The fact that they are split into two successive `TCGTemp` instances in `tcg_ctx->temps` clashes with the design used by SymQEMU to store the symbolic version of TCG variables, that we have described in 4.6, as the symbolic equivalent of a `TCGTemp` instance stored in `tcg_ctx->temps[i]` is expected to be found at `tcg_ctx->temps[i+1]`.

Luckily, i128 variables do not seem to be widely used by the TCG client. In the programs

that we have tested SymQEMU with, i128 variables were never generated.

For now, we have added an `assert` statement in the function `tcg_temp_new_internal` that triggers an error if an i128 variable is created, and an accompanying comment that explains the problem.

In the future, an important redesign of the way symbolic TCG variables are managed would probably be necessary if support for i128 TCG variables must be added to SymQEMU, as all the code that handles the symbolic versions of TCG variables is organized around the fact that the concrete and the symbolic version are next to each other in `tcg_ctx->temps`.

5.1.5 Util functions for accessing the symbolic version of a TCG variable

Finally, let us discuss an adaptation that we performed which is once again related to macro expansions.

In the old SymQEMU, the function `tcgv_i32_expr_num` takes as argument a TCG variable as a `TCGv_i32`, and returns the symbolic version of the TCG variable, as a `TCGv`. Similarly, the function `tcgv_i64_expr_num` takes as argument a TCG variable as a `TCGv_i64`, and returns the symbolic version of the TCG variable, as a `TCGv`.

`TCGv` is an alias for `TCGv_i32` or `TCGv_i64`, depending on whether the host architecture is 32 or 64 bits.

However, the actual code of those functions returns a `TCGv_i64`. Furthermore, in practice the return value of those functions are always use as operands of TCG instructions that expect operands of type `TCGv_i64`, not `TCGv_i32`. We do not know the motivation that made the original author of SymQEMU use `TCGv` here.

In practice, as SymQEMU is always executed on a 64 bit host, `TCGv` will always be replaced with `TCGv_i64`

In QEMU 8 `TCGv` is not defined anymore in the file that contains `tcgv_i32_expr_num` and `tcgv_i64_expr_num`. As a solution we simply replaced it with `TCGv_i64`, as this is what `TCGv` was substituted with anyway in the old SymQEMU.

Note that this is a different situation than 5.1.2. In 5.1.2, we replaced a macro whose expansion depended on the guest architecture, which can be 32 or 64 bits depending on the target program. Here, we replaced a macro whose expansion depends on the host architecture, which is always 64 bits.

5.2 Testing of the port to QEMU 8

After realizing the port of SymQEMU to QEMU 8.1, we wanted to validate that its behavior was correct.

Verifying the correctness of the outputs produced by SymQEMU is a difficult problem. Even minimalistic programs like a C hello world already lead to the execution of thousands of instructions, which makes impossible to manually determine the expected manipulation of symbol expressions during the execution.

The approach we chose was to consider the outputs generated by the original SymQEMU as being the reference. The criteria of correctness for the new SymQEMU would be that it gives the same output as the old SymQEMU, when run on the same target program.

We wrote a testing tool to automate this process, that we describe in 5.2.1. During this work, we discovered that SymQEMU can give different outputs depending on its execution context, we discuss this in 5.2.2.

We did not use the intermediate SymQEMU version based on QEMU 7 as reference, as this version had never been used or extensively tested. Note that, as our port is basd on the port to QEMU 7, a difference in behavior discovered between our new SymQEMU and the original one could mean that a bug was introduced in our port to QEMU 8 or that a bug already existed in the port to QEMU 7.

We planned to progressively test our new SymQEMU on target programs of increasing complexity. However, already when testing SymQEMU on our second test target program, a significant difference in behavior was discovered. We present the results of the tests on the first and second target program in 5.2.3.

5.2.1 End-to-end testing tool

In order to validate that our port of SymQEMU to QEMU 8 was correct, we compared the outputs that it generates to the outputs generated by the old SymQEMU.

For this purpose, we wrote a simple automated “end-to-end” testing tool in Python.

This testing tool is intended to be used with a series of test target programs. Each test program reads data from a file on disk and performs some computation on it. For each test target program, we store in a dedicated directory structure the executable of the test target program, and some standardized information about which arguments the target program should be run with and the concrete content of the file that the target program reads.

When a new test target program is added, the end-to-end testing tool runs the old SymQEMU on this new program, and stores, in the appropriate directory, the outputs generated by the old SymQEMU. Those outputs are considered to be the correct results.

Then, the end-to-end testing tool allows to automatically test a new version of SymQEMU by automatically running it on each test target program, and verifying that it gives the expected outputs.

Our goal was to progressively add test target programs of increasing complexity, and to validate each time that the outputs generated by the new SymQEMU were correct, when considering the outputs of the old SymQEMU to be the reference.

5.2.2 Reproducibility of the tests

Although the primary motivation for creating this end-to-end testing tool was to validate our port to QEMU 8, the tool could also be useful more globally in the SymQEMU project, to automatically test future modifications added to SymQEMU. The end-to-end testing tool could typically be used in a CI/CD pipeline for SymQEMU.

For this reason, it is necessary that the target test programs do not make SymQEMU have a different behavior on them depending on the execution context. This point is important, because in practice the behavior of SymQEMU is easily influenced by causes that come from outside of the target program, which leads to the generation of non-deterministic SymQEMU outputs.

For example, in an initial attempt to create a small test target program, we wrote a program that would read content from a file and print it to stdout. When running SymQEMU on this program with the file content being symbolic, the outputs of SymQEMU were not the same if we redirected the stdout of the program to a file (scenario 1) as if we executed it normally with stdout connected to a terminal (scenario 2).

After investigating, we discovered that, inside some low level glibc code that is responsible for writing to stdout, the execution path is different in scenario 1 than in scenario 2. As this code area manipulates the data to be printed, which is symbolic, the symbolic state diverges and different path constraints are generated, which leads to the generation of different outputs by SymQEMU.

For this reason, it is important that the test target program do not print symbolic data to stdout.

Another source of differences in the generated outputs are the dynamically linked shared libraries. If a test target program makes a call to a function from a shared library like glibc with a symbolic argument, then the symbolic execution will be different depending on which version of the shared library is installed on the system.

As a solution, we compiled the test target programs with a static linkage.

A future improvement to avoid other sources of non-deterministic behavior could be to run the end-to-end tests inside a docker container.

5.2.3 Results of the test of the new SymQEMU

The first target test program that we used was the following :

Code Source : tests/symqemu/binaries/simple/binary.c

```c
#include <stdio.h>

int main(int argc, char *argv[]) {

    if (argc != 2) {
        puts("ERROR: You need one argument.");
        return 1;
    }

    FILE* file_stream = fopen(argv[1], "r");

    if (!file_stream) {
        puts("ERROR: Could not open file.");
        return 1;
    }

    char input1 = getc(file_stream);
    char input2 = getc(file_stream);
    char input3 = getc(file_stream);

    if (input1 == 'a') {
        puts("foo1");
    }

    if (input2 == 'b') {
        puts("foo2");
    }

    if (input3 == 'c') {
        puts("foo3");
    }

}
```

This program reads three bytes from a file and compares them to constant values. The content of the file is symbolized when running the test.

With this program, our port of SymQEMU generates the same correct outputs as the original SymQEMU.

The second test target program was the following :

Code Source : tests/symqemu/binaries/simple/printf.c

```c
#define _GNU_SOURCE
#include <stdio.h>

int main(int argc, char *argv[]) {

    if (argc != 2) {
        puts("ERROR: You need one argument.");
        return 1;
    }

    FILE* file_stream = fopen(argv[1], "r");

    if (!file_stream) {
        puts("ERROR: Could not open file.");
        return 1;
    }

    char format_string[100];

    fgets(format_string, 100, file_stream);

    char *formatted_string;
    asprintf(&formatted_string, format_string, 1, 2, 3, 4);
}
```

This program reads the content of a file and uses it as the format string argument in a call to `asprintf`. We used `asprintf` instead of `printf` for the reproducibility reasons explained in 5.2.2.

When running the test, the input file contains the text `aaaaa %d %d %d %d aaaaa\n`, and is symbolized.

Table 5.1 contains the test cases generated by the old SymQEMU. The test cases are displayed as printed Python bytes objects, the non-printable characters are denoted by `\x` followed by a hexadecimal value.

Test case
b'\naaaa %d %d %d %d aaaaa\n'
b'aaaaa \x00d %d %d %d aaaaa\n'
b'aaaaa %\x9b %d %d %d aaaaa\n'
b'aaaaa % %d %d %d aaaaa\n'
b'aaaaa %X %d %d %d aaaaa\n'
b'aaaaa %X %d %d %d aaaaa\n'
b'aaaaa %d \x00d %d %d aaaaa\n'
b'aaaaa %d %{ %d %d aaaaa\n'

b'aaaaa %d %[%d %d aaaaa\n'
b'aaaaa %d %X %d %d aaaaa\n'
b'aaaaa %d %X %d %d aaaaa\n'
b'aaaaa %d %d \x00d %d aaaaa\n'
b'aaaaa %d %d %{ %d aaaaa\n'
b'aaaaa %d %d % %d aaaaa\n'
b'aaaaa %d %d %X %d aaaaa\n'
b'aaaaa %d %d %X %d aaaaa\n'
b'aaaaa %d %d %d \x00d aaaaa\n'
b'aaaaa %d %d %d %[aaaaa\n'
b'aaaaa %d %d %d %X aaaaa\n'

Table 5.1: Test cases generated by the old SymQEMU.

Table 5.2 contains the test cases generated by our port to QEMU 8.

Test case
b'\naaaa %d %d %d %d aaaaa\n'
b'aaaaa %d\xff%d %d %\x00 aaaaa\n'
b'aaaaa %d %d %d %d aaaaa\x00'
b'aaaaa %d %d %d %d aaaaa\x08'
b'aaaaa %d %d %d %d aaaaap'
b'aaaaa %d %d %d %d aaaaa\x04'
b'aaaaa %d %d %d %d aaaaa\xf8'
b'aaaaa %d %d %d %d aaaaa\xf8'
b'aaaaa %d %d %d %d aaaaa\xf8'
b'aaaaa %d %d %d %d aaaaa\xf8'

Table 5.2: Test cases generated by our port to QEMU 8.

As we can see, the outputs generated by the original SymQEMU and our port to QEMU 8 were vastly different.

The old SymQEMU generated test cases that seem relevant, as it was able to detect alternative format specifier, that would have made `asprintf` follow a different execution path.

Our port to SymQEMU generated fewer outputs, and those outputs do not seem to be relevant. No alternative format specifier were discovered, the outputs simply consist of adding non-printable characters to the original concrete input.

As an additional test, we executed the version of SymQEMU based on the port to QEMU 7 on the `asprintf` test program. The outputs that we got were the same as those generated

by SymQEMU based on QEMU 8. As a consequence, the problem was already present in the port to QEMU 7.

5.3 Discovering the root causes of the incorrect outputs

After discovering that our new version of SymQEMU gave incorrect results, we investigated in order to find what was the root cause for it.

As already discussed in the previous section, analyzing the causes that make SymQEMU output a given result is difficult. An output generated by SymQEMU is influenced by all path constraints that were previously generated in the execution. Each path constraint is itself influenced by all the symbol expressions generated previously, and all the computations performed on them.

This situation if prone to "butterfly effects" , where a difference in the behavior of two versions of SymQEMU at one point of the execution can lead to differences in outputs generated way later in the execution.

In this section, we will explain how we determined which instructions were the root cause of the differences observed in the output of the two SymQEMU versions.

In 5.3.1 we will first describe a new feature that we added to the symbolic backend, in order to make it output a full history of the symbolic state during the execution.

In 5.3.2, we will explain first findings that we made using it. Without surprise, we determined that the old and the new SymQEMU did not push the same path constraints. We found that the new SymQEMU pushing path constraints that the old one did not push was expected, for reasons that we will explain. However, we also found that the old SymQEMU pushed path constraints that the new one did not push, and that this was at least one of the causes of the bad quality outputs generated by the new SymQEMU.

Then in 5.3.3 and 5.3.4, we will present the methodology that we used for identifying which instructions were the cause of the new SymQEMU not pushing path constraints that the old SymQEMU did push.

Finally, in 5.3.5 we will explain how we applied this methodology in practice by automating parts of the analysis. As we will see, we found that the instructions responsible for the differences in behavior were SSE instructions.

5.3.1 Symbolic tracing

In order to be able to analyse and compare the symbolic state of SymQEMU during an execution, we added a feature to the symbolic backend, that makes it output a symbolic trace at the end of each execution.

The symbolic trace contains :

- A list of symbol expressions.
- A list of memory snapshots.
- A list of pushed path constraints.

Let us detail each element.

The list of symbol expressions is the list of all symbol expressions that have been generated during the execution. For each symbol, the following information is stored :

- A unique ID that identifies the expression.
- The expression operation (add, sub, ...).
- If the expression is a constant symbol, its concrete value.
- The size in bits of the value that this expression represents.
- A list of the IDs of the parents of the symbol expression.

The list of memory snapshots contains snapshots of the symbolic state of the memory, that are collected repetitively during the execution, at the beginning of each instruction block.

A snapshot contains the list of all memory bytes that are symbolic at a given time of the execution, along with the IDs of the symbol expressions that are associated to them. The snapshot covers both the shadow pages and the symbolic version of the guest registers stored in the memory of QEMU. Each snapshot also contains the PC value at the time of the snapshot creation.

The list of pushed path constraints contains an entry for each path constraint that is pushed during the execution. Each entry includes :

- The ID of the symbol expression associated to the path constraint.
- A boolean value that indicates if the associated path is taken or not.
- The index of the most recent memory snapshot at the time of pushing the path constraint.
- The new input value generated by the backend, if the backend was able to generate a new input value by negating this path constraint.

Technically, this was done by creating a C++ class that stores the trace data accumulated during the execution. Calls to this class are performed regularly during execution, when events like the execution of a new instruction block or the generation of a new path constraint happen.

At the end of the execution, the trace is outputted as a json file.

5.3.2 Different path constraints

By analyzing the symbolic traces generated by the old and the new SymQEMU, we determined that the two versions do not push the same path constraints.

After some investigation, we discovered a cause that explained some of those differences : some guest instructions that are translated using helper functions in the old SyQEMU are now translated to TCG ops in the new SymQEMU. As discussed in 4.7.3, SymQEMU does not instrument helper functions. The fact that fewer instructions use helper functions in the new SymQEMU makes it expected that this version generates more symbolic expressions, and thus more path constraints.

However, we also observed that some path constraints pushed by the old SymQEMU were not pushed by the new SymQEMU. In particular, we saw that the good quality test cases outputted by the old SymQEMU were generated by negating path constraints that the new SymQEMU did not push, which confirmed that those missing path constraints were at least part of the cause of the missing good quality test cases.

As a consequence, we needed to understand what was the root cause that made the new SymQEMU not push path constraints that the old SymQEMU did push.

5.3.3 Description of the problem to solve

We are now going to present a more formal description of the problem that we had to solve, in order to understand the root causes that made the old SymQEMU push path constraints that the new one did not push.

Let us introduce some definitions :

For any time T in the execution, we call a *missing path constraint* any path constraint that the old SymQEMU has pushed at time T, but that the new SymQEMU has not pushed.

Furthermore, for any snapshot S of the symbolic trace of the old SymQEMU, and the corresponding snapshot S' of the symbolic trace of the new SymQEMU, we call a *missing symbolic byte* any byte that is symbolic in S but that is not symbolic in S', or any byte that is symbolic in both S and S' but with different associated symbol expressions.

Our goal is to understand why the missing path constraints are not pushed by the new

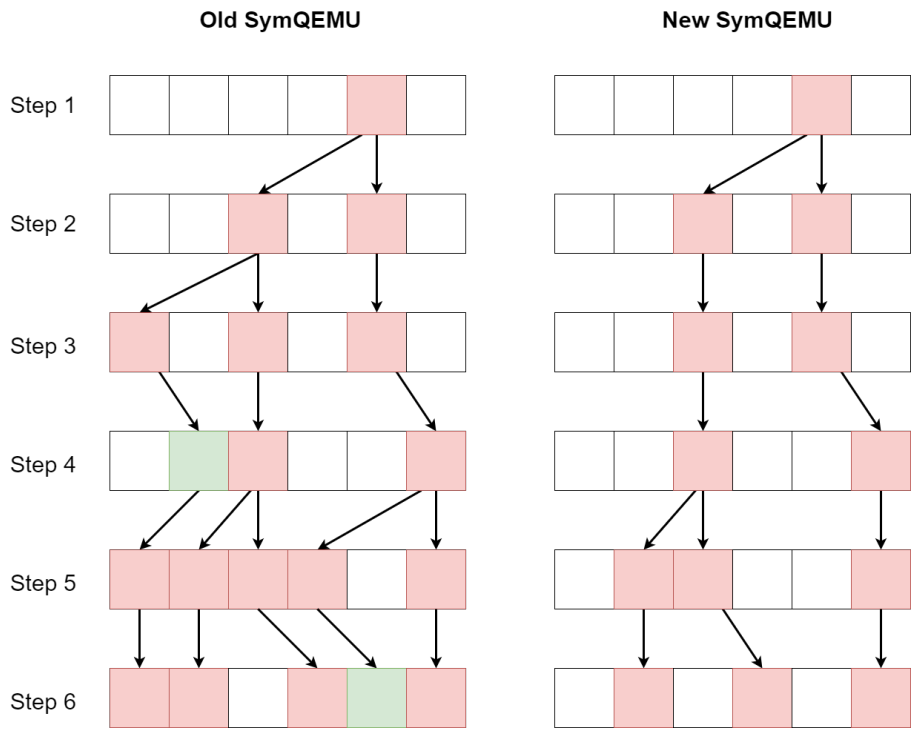

Figure 5.1: Comparison of symbolic traces of the old and new SymQEMU.

SymQEMU.

We can easily understand that this problem is caused by the fact that some guest instructions make the old and the new SymQEMU apply a different modification on the symbolic state. The propagation of those differences lead to the symbols associated to the missing path constraints being lost in the new SymQEMU.

Let us call those instructions the *problematic instructions*.

As an illustration, Figure 5.1 represents a comparison of the execution traces generated by the old and the new SymQEMU.

Each row corresponds to an execution step for which a symbolic memory snapshot was taken. Each square is a memory byte (or a register). A red square means that the memory byte is symbolic. An arrow between two symbolic bytes A and B means that either A and B are associated to the same symbol expression, or that the expression associated to byte A is a parent of the expression associated to byte B. If a row contains a green byte, a path constraint was pushed at the corresponding execution step, and the symbol expression of the path constraint is the symbol expression associated to the green byte.

In this example, we can see that the path constraints pushed by the old SymQEMU at steps

4 and 6 are missing path constraints.

The problem that we must solve is the following : given two symbolic traces like the ones represented in 5.1, we need to find at which execution steps problematic instructions were executed.

5.3.4 Methodology for discovering problematic instructions

The first problematic instruction is easy to find, because it is the last instruction that was executed before the apparition of the first missing symbolic byte.

In the example represented in 5.1, the first missing symbolic byte is the first byte at step 3. As a consequence, we know that the first problematic instruction was executed between steps 2 and 3.

Finding the subsequent problematic instructions is more difficult, because the difference in the symbolic state caused by the first problematic instruction will propagate, and thus in the subsequent steps it is not obvious if a missing symbolic byte is a consequence of the first problematic instruction, or if it is caused by a new problematic instruction.

The approach that we used to solve this problem is the following :

We start by taking the first missing path constraint. Let us call A the symbol expression that is associated to it. Let b be the first missing symbolic byte that is an ancestor of A. Then, the last executed instruction before b appeared is a problematic instruction.

We repeat this process with the second path constraint, the third, etc. This gives us a list of problematic instructions.

To illustrate this, Figure 5.2 represents the same comparison as Figure 5.1, but, for the old SymQEMU, only the symbol expressions that are ancestors of the second path constraint are showed. We see that the first missing symbolic byte appears at step 5, and thus a problematic instruction was executed between steps 4 and 5.

For the sake of simplicity, in this example all missing symbolic bytes were simply not symbolic in the new SymQEMU version. However, note that in practice it is important to not only verify that the corresponding byte in the new version of SymQEMU is symbolic, but also to control if both symbol expressions are equal. This is due to the fact that a problematic instruction may actually generate a symbol expression, but an incorrect one.

To check that two symbols are equal, we verify that all their characteristics are equal (same type, length, etc.) and recursively perform the same check for each parent expression.

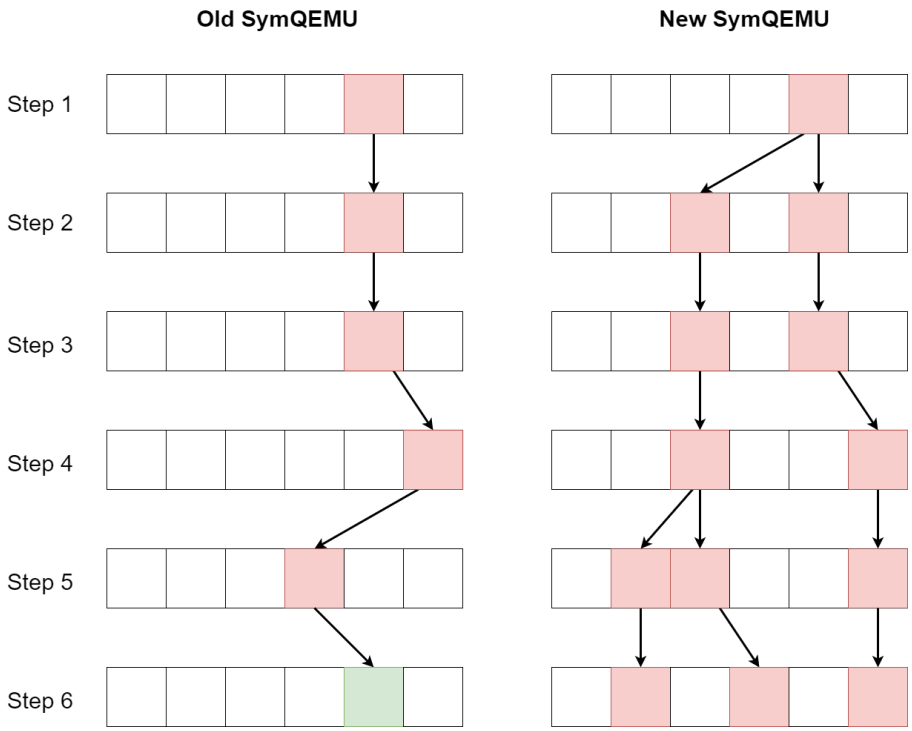

Figure 5.2: Comparison of symbolic traces of the old and new SymQEMU, with filtered symbols.

5.3.5 Problematic instructions discovered

Using this methodology, we compared the symbolic traces generated by the new and the old SymQEMU when executed on the `asprintf` test target program.

We did this in a semi-automated way, with the help of Python code for automatically parsing the json symbolic trace, detecting the missing symbolic bytes, determining if a symbolic expression in an ancestor of another, etc.

One problem that we encountered for this analysis is that the old and the new SymQEMU use different memory addresses for storing corresponding information. For example, the stack is not located at the same location in memory in the old and in the new SymQEMU. Additionally, the elements of `CPUArchState` that represent the guest registers are not located at the same memory address in the two versions of SymQEMU.

As the symbolic traces stores the memory addresses of the symbolic bytes, those traces can not be directly compared. As a solution, we added some code in both versions of SymQEMU that makes them output, in another json file, the addresses of the different memory regions, including the addresses of the guest memory segments (stack, data, etc.) and the addresses of the registers in `CPUArchState`.

Our Python code uses this information to interpret addresses not as absolute values but as offsets relative to the memory regions.

The result of our analysis was the following : all problematic instructions that we discovered were SSE instructions.

5.4 SSE instructions in SymQEMU

In the previous sections, we saw that, when executed on the `asprintf` test program, the old SymQEMU pushed path constraints that the new version did not push, which was the probable cause of the low quality outputs generated by the new SymQEMU. We then discovered that the cause for those missing path constraints was the fact that SSE instructions had a different effect on the symbolic state in the old SymQEMU than in the new one.

After this discovery, the next step was to understand how SymQEMU handles SSE instructions.

We first examined how SSE instructions are translated in QEMU 4, and which consequences this translation has for the old SymQEMU. We discovered that the old SymQEMU handles SSE instructions in a partially incorrect way. Our findings are explained in 5.4.1.

Then, we did the same for QEMU 8 and the new SymQEMU. We discovered that the way SSE instructions are translated to TCG ops has changed between QEMU 4 and QEMU 8, and that this change make the new SymQEMU completely unable to handle SSE instruc-

tions. This is explained in 5.4.2.

Finally, in 5.4.3, we will analyze how this difference in handling of SSE instructions leads to the new SymQEMU and the old SymQEMU generating different path constraints.

5.4.1 SSE instructions with the old SymQEMU

In QEMU 4, the translation from x86 to TCG of an SSE instruction is performed in either one of the following ways :

- Simple move SSE instructions are translated to standard TCG ops. Typically, this is done using *ld* and *st* TCG ops, that target the simulated XMM registers in the `CPUArchState` instance. Let us call `type 1` instructions the SSE instructions that QEMU 4 translates to TCG ops.
- SSE instructions that perform more complex computations are translated to calls to dedicated helper functions. The helper functions directly perform the computations on the simulated XMM registers in the `CPUArchState` instance. Let us call `type 2` instructions the SSE instructions that QEMU 4 translates to calls to helper functions.

Let us examine the behavior that this situation causes for the old SymQEMU.

`Type 1` instructions lead to a correct propagation of symbol expressions by SymQEMU, as the TCG ops are instrumented.

With `type 2` instructions, no symbol expressions are generated for the SSE computations, as SymQEMU does not instrument helper functions. Even worse, instead of just losing the symbolic state, `type 2` instructions tend to lead to the propagation of incorrect symbol expressions. This is due to the fact that `type 2` instructions have no effect at all on the symbolic state, and thus if the destination operand was symbolic before the instruction, it stays associated to the same symbol expression and subsequent computations on it will cause the generation of incorrect expressions.

Note that, as we have seen in 4.7.3, SymQEMU concretizes the return value of helper functions. However, as the helper functions for SSE instructions directly manipulate the host memory and do not return anything, this concretization does not happen.

Let us present a real situation that involves `type 1` and `type 2` instructions.

At some point in our test target program execution, the following x86 instructions are executed while the memory area to which RDI points is symbolic :

```
movdqu xmm0,XMMWORD PTR [rdi]
pcmpeqb xmm0,xmm1
pmovmskb eax,xmm0
```

Let us analyze what happens for the old version of SymQEMU.

First, `movdqu` is translated to standard TCG ops, because it is a `type 1` instruction. Thus, the symbolic side of the execution is done correctly and XMM0 gets a symbolic value.

Then, `pcmpeqb` is executed with a helper function, because it is a `type 2` instruction. `pcmpeqb` performs a parallel comparison of the elements of XMM0 and XMM1 and sets each element of XMM0 to all 0 or all 1, depending on the result. After a correct symbolic execution of this instruction, XMM0 should become concrete (and of course some path constraint should be pushed). But because the helper function is not instrumented, XMM0 is instead still associated to the same symbol expression as before the instruction.

Finally, `pmovmskb` is translated to standard TCG ops, because it is a `type 1` instruction. As a consequence, the incorrect symbol expression hold by XMM0 is moved to EAX, and this incorrect symbolic state will be propagated to the subsequent steps of the execution.

In summary, with the old SymQEMU, some SSE instructions are correctly handled, while others are not and may lead to the propagation of incorrect symbol expressions.

5.4.2 SSE instructions with the new SymQEMU

In QEMU 8, most SSE instructions are translated to vector TCG ops. Actually, this is the case for all SSE instructions encountered in the `asprintf` test target program. In particular, `type 1` instructions, that QEMU 4 translated to standard (non vector) TCG ops, are now translated to vector TCG ops.

Vector TCG ops are not instrumented in SymQEMU. As a consequence, in the new SymQEMU, SSE instructions have no effect at all on the symbolic state. This does not lead to the propagation of incorrect symbol expressions like for `type 2` instructions in the old SymQEMU, because, in the new SymQEMU, there is no situation where an XMM register can get a symbolic value. Therefore, executing an SSE instruction simply leads to the loss of symbol expressions.

5.4.3 Consequences of the different handling of SSE instructions

Now that we have a good understanding of how the old and the new SymQEMU handle SSE instructions, we can see that missing path constraints are due to two different root causes.

The first cause is that, when a `type 1` SSE instruction is executed on a symbolic operand, the old SymQEMU correctly propagates symbol expressions while the new one drops them, which leads to correct path constraints being pushed by the old SymQEMU while the new SymQEMU does not push them.

The second cause is that, when a `type 2` SSE instruction is executed on a symbolic operand,

the old SymQEMU may propagate incorrect symbol expressions, while the new SymQEMU just drops the symbol expressions. This also results to path constraints being pushed by the old SymQEMU and not the new one. Note that, here, the path constraints pushed by the old SymQEMU are incorrect.

Thanks to this result, we instrumented the vector TCG ops, in order to obtain a version of the new SymQEMU that was able to give as good quality outputs as the old SymQEMU for the `asprintf` target program. This work is the subject of Chapter 6.

Chapter 6

Instrumentation of the vector TCG instructions

In this chapter, we will present the main contribution that we have made to SymQEMU: the instrumentation of the vector TCG instructions.

As we have already discussed, the main modification that SymQEMU does to QEMU is to inject additional *instrumentation TCG ops* during the translation of instructions, that perform a symbolic execution of the target program, in parallel to the normal concrete execution.

Adding this new ability to QEMU requires to manually edit the code that generates each TCG instruction, in order to make it emit instrumentation TCG ops that reflects, on the symbolic state, the effect that this instruction has on the concrete state. Although the authors of SymQEMU have already done this instrumentation work for most TCG instructions, when we started this project SymQEMU missed the instrumentation of vector TCG instructions.

Implementing those instructions was motivated by the results of the analysis that we performed in Chapter 5. It allowed us to obtain a new version of SymQEMU that generates outputs of better quality.

To present our contribution, in Section 6.1 we will first enumerate all vector TCG instructions and give an overview of the similarities and differences in the instrumentation work that was performed for each of them. In Section 6.2, we will discuss an interesting problem that we encountered for passing the vectors to the symbolic helper functions at run time, and the solution that we have chosen for it.

We will then present in details the implementation of the instrumentation of two vector instructions. In 6.3, we will show how the instrumentation of the *add_vec* instruction is implemented. Then in 6.4 we will show how we instrumented the *min* and *max* vector

Vector-vector	**Vector-integer**	**Load / store**	**Duplication**	**Mov**	**Path constraint**	**Indirect instrumentation**
See 6.1.1	See 6.1.2	See 6.1.3	See 6.1.4	See 6.1.5	See 6.1.6	See 6.1.7
and_vec	*shls_vec*	*ld_vec*	*dup_i64_vec*	*mov_vec*	*smin_vec*	*dupi_vec*
or_vec	*shrs_vec*	*st_vec*	*dup_i32_vec*		*umin_vec*	*andc_vec*
xor_vec	*sars_vec*		*dup_mem_vec*		*smax_vec*	*orc_vec*
add_vec	*rotls_vec*				*umax_vec*	*nand_vec*
sub_vec	*shli_vec*				*cmp_vec*	*nor_vec*
mul_vec	*shri_vec*					*eqv_vec*
ssadd_vec	*sari_vec*					*not_vec*
usadd_vec	*rotli_vec*					*neg_vec*
ssub_vec	*rotri_vec*					*abs_vec*
ussub_vec						*bitsel_vec*
shlv_vec						*cmpsel_vec*
shrv_vec						
sarv_vec						
rotlv_vec						
rotrv_vec						

Table 6.1: Vector TCG instructions by categories.

TCG instructions, which are of particular interest as they trigger the generation of path constraints.

6.1 Instrumentation strategies

Table 6.1 contains all TCG vector instructions, grouped by similarities.

In this section, we will discuss how we have instrumented each group of instructions. Each subsection below refers to one of the columns of Table 6.1.

6.1.1 Vector-vector TCG instructions

Those TCG instructions have two input vector operands `arg1` and `arg2` and one output vector operand `result`. They perform and SIMD computation on the elements of the vectors, such that `result[i] = arg1[i] <operation> arg2[i]`, where `vector[i]` denotes the ith element of `vector`.

For each of them, we have written a corresponding symbolic helper function in *accel/tcg/tcg-runtime-sym-vec.c*. We have also added code to the TCG API functions for generating

those TCG instructions, that injects instrumentation TCG ops for calling the corresponding symbolic helper at run time.

As the instrumentation work is near identical for all of those TCG ops, we used util functions to factorize it.

Section 6.3 shows in details how the instrumentation works for the TCG instruction *add_vec*.

6.1.2 Vector-integer TCG ops

Those TCG instructions have one input vector operand `arg1`, one input integer operand `arg2` and one output vector operand `result`. They perform and SIMD computation on the elements of the vectors, such that `result[i] = arg1[i] <operation> arg2`, where `vector[i]` denotes the ith element of `vector`.

We have instrumented those instructions in a way similar to the vector-vector TCG instructions.

For each of them, we have written a corresponding symbolic helper function in *accel/tcg/tcg-runtime-sym-vec.c*. We have also added code to the TCG API functions for generating those TCG instructions, that injects instrumentation TCG ops for calling the corresponding symbolic helper at run time.

Like for vector-vector TCG instructions, we used util functions to factorize similar code.

6.1.3 Load / store vector TCG instructions

The TCG instructions *ld_vec* and *st_vec* perform respectively a load from memory to a vector TCG variable, and a store from a vector TCG variable to memory.

We have instrumented them mostly be reusing the functions that had already been written for the symbolic loading / storing of normal TCG variables. The instrumentation performs loads and stores from / to the shadow pages of the symbolic backend.

6.1.4 Duplication vector TCG instructions

The TCG instructions *dup_i64_vec*, *dup_i32_vec* and *dup_mem_vec* allow to create vector TCG variables by filling them with a repeated integer value.

We have instrumented them by writing appropriate symbolic helper functions and adding code that make those functions be called at run time.

6.1.5 Vector TCG instructions that trigger the creation of path constraint

Those TCG instructions all perform an element-wise comparison of vector elements. We instrumented them such that appropriate path constraints are pushed when they are executed.

In Section 6.4, we present a detailed description of this instrumentation.

6.1.6 mov_vec TCG instruction

The TCG instruction *mov_vec* copies the value of a source vector operand to a target vector operand.

To instrument it, we simply added a line of code that injects an equivalent `mov` instruction for the symbolic version of the operands.

6.1.7 Vector TCG instructions that delegate to other TCG instructions

When the TCG API client requests to add a TCG op for one of those instructions, the TCG backend may instead add different vec TCG ops that have an equivalent effect. For example, a `nand_vec` op can be replaced with an `and_vec` op followed by a `not_vec` op applied on the result.

This substitution is normally performed only if the host does not support certain TCG instructions.

We have edited the code to make sure that those substitutions are systematically performed. As the substitute TCG instructions are instrumented, there is no need to instrument the substituted TCG instructions.

This approach simplifies the instrumentation work, at the cost of slightly lowering the performance of SymQEMU as the use of multiple TCG ops instead of a single one could lead to the generation of more complex symbol expressions.

6.2 Passing the concrete version of the vectors to the symbolic helpers

Like for the other instrumented instructions in SymQEMU, the instrumentation TCG ops need to pass the concrete version of the operands to the symbolic helper. This is necessary, because if one operand is symbolic and the other is concrete, the symbolic helper needs to create a new constant symbol for the concrete operand.

We have seen that, for standard non vector instructions, those values are simply passed as argument to the symbolic helper. However, for vector instructions, passing the concrete version of a vector operand in a helper argument is not possible, because of the size of the vector.

TCG vectors can have a size of 64, 128 or 256 bits. Helper functions can receive arguments of size 64 bits, and there is also some support for 128 bit arguments, but passing a 256 bit argument is not supported.

The solution that we have implemented is to make the instrumentation TCG ops store the vector into a heap buffer at run time, and pass its address to the symbolic helper.

More precisely, each time a concrete vector should be passed to a symbolic helper, we add instrumentation TCG ops that do the following, at run time :

- Call a helper function that is a wrapper around the C malloc function.
- Store the concrete vector into the malloced buffer.
- Call the symbolic helper with the malloced buffer address as argument.
- Call a helper function that is a wrapper around the C free function, in order to free the buffer.

The implementation of this solution is showed in 6.3, where we present the code for instrumenting the *add_vec* instruction.

6.3 Instrumentation of the add_vec instruction

In this section, we will explain in detail how the instrumentation for the TCG instruction *add_vec* is performed.

Our goal is to show a representative example of instrumentation of a vector instruction. In particular, all TCG instructions listed in 6.1.1 are instrumented in a near identical way to *add_vec*.

We will first discuss the code that is executed at translation time, and that injects instrumentation TCG ops. The starting point is the function `tcg_gen_add_vec`, discussed in 6.3.1, which is the function called by the TCG API client to add an *add_vec* op. The code that we have added to this function performs a call to the util function `vec_vec_op_instrumentation`, discussed in 6.3.2. This function itself uses `store_vector_in_memory`, which we will discuss in 6.3.3

Then we will present the code called at run time by the instrumentation TCG ops. We will start with the helper function `helper_sym_add_vec` in 6.3.4 and we will see that it

is actually a wrapper for the function `build_expression_for_vector_vector_op`, presented in 6.3.5. In 6.3.6 and 6.3.7, we will also show the functions `split_expression` and `apply_op_and_merge`, which are used by `build_expression_for_vector_vector_op`.

6.3.1 Function tcg_gen_add_vec

This function is part of the TCG API, it is called by the client to add an *add_vec* TCG op.

We have modified it such that it injects instrumentation TCG ops right before the *add_vec* TCG op.

Here is the modified definition of `tcg_gen_add_vec` :

Code Source : tcg/tcg-op-vec.c

```
1 void tcg_gen_add_vec(unsigned vece, TCGv_vec r, TCGv_vec a, TCGv_vec b)
2 {
3     vec_vec_op_instrumentation(vece, r, a, b, gen_helper_sym_add_vec);
4     do_op3_nofail(vece, r, a, b, INDEX_op_add_vec);
5 }
```

The argument `vece` controls the element size, as explained in 3.4.7. `r` is the output vector operand, `a` and `b` are the input vector operands.

We have added the line 3, which injects the instrumentation TCG ops. `gen_helper_sym_add_vec` is the function for calling the symbolic helper `helper_sym_add_vec` that is discussed in 6.3.4.

Recall that helper functions have been discussed in 3.4.10.

6.3.2 Function vec_vec_op_instrumentation

`vec_vec_op_instrumentation` is a util function that is responsible for injecting TCG ops for calling a symbolic helper.

`vec_vec_op_instrumentation` is used for the instrumentation of vector TCG instructions that have two vector input operands. There is also a similar function `vec_int32_op_instrumentation` for the instrumentation of TCG instructions that have one vector and one integer input operand.

Here is the definition of the function `vec_vec_op_instrumentation` :

Code Source : tcg/tcg-op-vec.c

```
1 /* Adds instrumentation TCG ops for an instruction of the form vec = vec <op> vec.
2  *
```

```
3  * Args
4  *      vece: element size = 8 * 2^vece
5  *      r: output operand
6  *      a: first input operand
7  *      b: second input operand
8  *      sym_helper: function for calling the symbolic helper
9  */
10 static void vec_vec_op_instrumentation(
11         unsigned vece,
12         TCGv_vec r, TCGv_vec a, TCGv_vec b,
13         void (*sym_helper)(TCGv_ptr, TCGv_ptr, TCGv_ptr, TCGv_ptr, TCGv_ptr,
   ↪  TCGv_i64, TCGv_i64)
14 ) {
15     TCGv_ptr buffer_address_a = store_vector_in_memory(a);
16     TCGv_ptr buffer_address_b = store_vector_in_memory(b);
17     int size_a = vec_size(a);
18     int size_b = vec_size(b);
19     int size_r = vec_size(r);
20
21     g_assert(size_a == size_b && size_b == size_r);
22
23     sym_helper(
24             tcgv_vec_expr(r),
25             buffer_address_a,
26             tcgv_vec_expr(a),
27             buffer_address_b,
28             tcgv_vec_expr(b),
29             tcg_constant_i64(size_a),
30             tcg_constant_i64(vece)
31     );
32
33     gen_helper_free(buffer_address_a);
34     gen_helper_free(buffer_address_b);
35 }
```

At lines 15 - 16, we add TCG ops for storing the concrete versions of the input vector operands into a malloced buffer. This is done by calling `store_vector_in_memory`, discussed in 6.3.3.

At line 23, we add a TCG op that will call the symbolic helper function at run time. This is done by calling the function that was given as argument to `vec_vec_op_instrumentation`. In our case, as we are following the instrumentation of the *add_vec* instruction, this function is `gen_helper_sym_add_vec`, which adds a TCG op that will call the helper function `helper_sym_add_vec` at run time.

At run time, `helper_sym_add_vec` will receive as argument :

- The addresses of the two buffers where the concrete versions of the input operands

have been stored

- The size of those buffers
- The symbolic version of the two input operands
- The `vece` value, that determines the element size for the SIMD operation, as described in the QEMU documentation [4].

The value returned by the helper function will be stored in the symbolic version of the output operand.

Finally, at lines 33 - 34 we add TCG ops that will, at run time, call helper functions for freeing the two previously allocated buffers.

6.3.3 Function store_vector_in_memory

The function `store_vector_in_memory` is responsible for adding TCG ops that allocate a heap buffer and store into it the content of a vector TCG variable.

Here is its definition :

Code Source : tcg/tcg-op-vec.c

```
1  /* Adds TCG ops for storing a vector TCG variable in a heap buffer.
2   *
3   * The caller must add TCG ops for freeing the buffer.
4   *
5   * Returns
6   *      A TCG variable that will contain the buffer address at run time.
7   */
8  static TCGv_ptr store_vector_in_memory(TCGv_vec vector){
9      TCGv_ptr buffer_address = tcg_temp_new_ptr();
10     gen_helper_malloc(buffer_address, tcg_constant_i64(vec_size(vector) / 8));
11
12     /* store vector at buffer_address */
13     vec_gen_3(
14             INDEX_op_st_vec,
15             tcgv_vec_temp(vector)->base_type,
16             0,
17             tcgv_vec_arg(vector),
18             tcgv_ptr_arg(buffer_address),
19             0
20     );
21
22     return buffer_address;
23 }
```

The argument `vector` represents the vector TCG variable that we want to store in memory at run time.

At line 9, we generate a new TCG variable for the buffer address. At line 10, we add a TCG op for calling the malloc helper function and storing the resulting address in the TCG variable.

At lines 13 - 20, we manually generate a *st_vec* TCG op that stores the content of the vector TCG variable in the buffer.

Note that the normal way of generating a *st_vec* TCG op is to call the TCG API function `tcg_gen_st_vec`. However, here we should not use it, because `tcg_gen_st_vec` is instrumented. Calling `tcg_gen_st_vec` would add the instrumentation TCG ops that store the symbolic version of `vector` into the shadow pages, which is not what we want.

As a consequence, we have to use lower level code for generating the *st*. The function call that we make at lines 13 - 20 is equivalent to what `tcg_gen_st_vec` does internally.

6.3.4 Function helper_sym_add_vec

Now that we have discussed the instrumentation code executed at translation time, let us see the code called by this instrumentation at run time.

`helper_sym_add_vec` is the symbolic helper function that is called at run time by the instrumentation TCG ops, just before an *add_vec* op is executed.

Here is its definition :

Code Source : accel/tcg/tcg-runtime-sym-vec.c

```
1 void *HELPER(sym_add_vec)(void *arg1_concrete, void *arg1_symbolic, void
  ↪  *arg2_concrete, void *arg2_symbolic, uint64_t vector_size, uint64_t vece){
2
3     return build_expression_for_vector_vector_op(arg1_concrete, arg1_symbolic,
  ↪  arg2_concrete, arg2_symbolic, size, vece, _sym_build_add);
4 }
```

Recall that, as explained in 3.4.10, `*HELPER(sym_add_vec)` is expanded to `helper_sym_add_vec`.

`helper_sym_add_vec` actually is just a wrapper that calls `build_expression_for_vector_vector_op`, and passes to it the function `_sym_build_add` as argument. `_sym_build_add` is the symbolic backend API function for generating a symbol that corresponds to an addition operation.

6.3.5 Function build_expression_for_vector_vector_op

`build_expression_for_vector_vector_op` is responsible for generating a symbol expression that represents the result of an SIMD operation.

Here is its definition :

Code Source : accel/tcg/tcg-runtime-sym-vec.c

```
/*
 * Builds a symbol expression for an SIMD operation on two vector operands.
 * Args
 *      arg1_concrete, arg2_concrete : pointers to buffers that store the concrete values of the input vector operands
 *      arg1_symbolic, arg2_symbolic : symbolic expressions of the input vector operands
 *      vector_size : size of the vectors in bits
 *      vece : element size in bits = 8 * 2^vece
 *      symbolic_operation : function for the symbolic operation applied on each element of the vectors
 * Returns
 *      A symbol expression that represents the output of the SIMD operation
 */
static void *build_expression_for_vector_vector_op(
        void *arg1_concrete, void *arg1_symbolic,
        void *arg2_concrete, void *arg2_symbolic,
        uint64_t vector_size, uint64_t vece,
        void *(*symbolic_operation)(void *, void *)
) {
    g_assert(vector_size == 64 || vector_size == 128 || vector_size == 256);
    uint64_t element_size = vece_element_size(vece);
    g_assert(element_size <= vector_size);
    g_assert(vector_size % element_size == 0);

    if (arg1_symbolic == NULL && arg2_symbolic == NULL) {
        return NULL;
    }

    if (arg1_symbolic == NULL) {
        arg1_symbolic = _sym_build_integer_from_buffer(arg1_concrete, vector_size);
    }

    if (arg2_symbolic == NULL) {
        arg2_symbolic = _sym_build_integer_from_buffer(arg2_concrete, vector_size);
    }

    g_assert(_sym_bits_helper(arg1_symbolic) == _sym_bits_helper(arg2_symbolic) &&
             _sym_bits_helper(arg1_symbolic) == vector_size);

    uint64_t element_count = vector_size / element_size;
```

```
39      void *arg1_elements[element_count];
40      void *arg2_elements[element_count];
41      split_expression(arg1_symbolic, element_count, element_size, arg1_elements);
42      split_expression(arg2_symbolic, element_count, element_size, arg2_elements);
43
44      void *result = apply_op_and_merge(symbolic_operation, arg1_elements,
    ↪   arg2_elements, element_count);
45      g_assert(_sym_bits_helper(result) == vector_size);
46      return result;
47  }
```

The argument `symbolic_operation` is the symbolic backend API funtion that corresponds to the operation that must be applied to each pair of vector elements.

At line 23, if both input operands are concrete we return `NULL`, which makes the output operand concrete.

At lines 27 - 33, if one input operand is concrete and the other is symbolic, we create a new constant symbol for the concrete operand. `_sym_build_integer_from_buffer` is a function that we have added to the symbolic backend API, which allows to create large constant symbols.

At lines 38 - 42, for both input operands we generate a separate symbol expression for each vector element. This is done with the function `split_expression` discussed in 6.3.6.

Finally, at line 44, we apply the operation to each pair of symbol expressions, and we generate a final symbol expression that represents the concatenation of all operation results. This is done with the `apply_op_and_merge` function, which is discussed in 6.3.7.

6.3.6 Function split_expression

The function `split_expression` is used to generate symbol expressions that correspond to each individual elements of a vector.

Here is its definition :

Code Source : accel/tcg/tcg-runtime-sym-vec.c

```
1  /*
2   * Splits an expression into multiple symbol expressions of equal size.
3   *
4   * The expressions are stored in little endian order : result[0] is the least
    ↪   significant element of the original expression,
5   * result[element_count - 1] is the most significant element.
6   *
7   * Args
```

```
 *      expression : the expression to split
 *      element_count : number of expressions to split into
 *      element_size : size of each result expression in bits
 *      result : array of size element_count to store the result
 */
static void split_expression(void* expression, uint64_t element_count, uint64_t
    element_size, void* result[]){
    g_assert(_sym_bits_helper(expression) % element_count == 0);
    for(uint64_t i = 0; i < element_count; i++){
        result[i] = _sym_extract_helper(expression, (i + 1) * element_size - 1, i *
    element_size);
    }
}
```

`_sym_extract_helper` is a function of the symbolic backend. It creates a symbol expression corresponding to the extraction of a range of bits from another symbol expression.

6.3.7 Function apply_op_and_merge

The function `apply_op_and_merge` is used to apply a symbolic operation on pairs of symbol expressions, and to concatenate the results into a single symbol expression.

Here is its definition :

Code Source : accel/tcg/tcg-runtime-sym-vec.c

```
/* Pairwise applies a binary operation on two arrays of symbol expressions and
    concatenates the result into a single symbol expression.
 *
 * The concatenation is done in little endian order : symbolic_operation(args1[0],
    args2[0]) is the least significant
 * element of the result, symbolic_operation(args1[element_count - 1],
    args2[element_count - 1]) is the most significant.
 *
 * Args
 *      symbolic_operation : function for the symbolic operation applied on each pair
    of elements
 *      args1 : first array of symbol expressions
 *      args2 : second array of symbol expressions
 *      element_count : number of elements in each array
 * Returns
 *      A symbol expression that represents the concatenated results
 */
static void *apply_op_and_merge(
        void *(*symbolic_operation)(void *, void *),
        void *args1[], void *args2[],
        uint64_t element_count
```

```
18 ) {
19     void *merged_result = symbolic_operation(args1[0], args2[0]);
20     for (uint64_t i = 1; i < element_count; i++) {
21         merged_result = _sym_concat_helper(symbolic_operation(args1[i], args2[i]),
   ↪ merged_result);
22     }
23     return merged_result;
24 }
```

6.4 Instrumentation of min / max vector TCG instructions

In this section, we are going to show how the `min` and `max` vector TCG instructions are instrumented.

Our goal is to demonstrate how the instrumentation of TCG instructions that trigger the creation of path constraints works. The other vector TCG instruction that leads to new path constraints, *cmp_vec*, is instrumented in a way similar to the `min` and `max` ops

First, in 6.4.1 we are going to see that QEMU substitutes the `min` and `max` vector TCG ops with a more generic TCG op generated by the function `do_minmax`. In 6.4.2 we will analyze the semantic of this TCG op during a concrete execution. Then, in 6.4.3 we will explain what effect this TCG op should have on the symbolic state. Finally in 6.4.4 we will present our implementation of the instrumentation for this instruction.

6.4.1 Delegation to do_minmax

The instructions *smin_vec*, *umin_vec*, *smax_vec* and *umax_vec* are the TCG instructions for performing an element-wise min or max operation on vector TCG variables (the *s* and *u* stand for signed / unsigned).

Those TCG instructions do not actually exist in the point of view of the TCG backend. When the TCG API client adds one of those instructions, what really happens is that the TCG backend internally calls the function `do_minmax`.

6.4.2 Semantic of the TCG op added by do_minmax

Given a vector element size and a comparison operator like "`<`" or "`>`", `do_minmax` adds a TCG op whose semantic is the following :

Let *result* , *arg1* , *arg2* be respectively the output, first input and second input of the operation. *result* , *arg1* and *arg2* are vector TCG variables. Furthermore, let n be the

number of elements in which the above vector operands are divided. Let `vec[i]` denote the ith element of the vector operand *vec*.

The op added by `do_minmax` computes `result` such that

`result[i] = arg1[i] <comparison operator> arg2[i] ?  arg1[i] :  arg2[i]`

for $i = 1 \dots n$.

We can easily see that, for example, calling `do_minmax` with the "`<`" comparison operator is equivalent to adding an element-wise min TCG op.

6.4.3 Symbolic semantic of the TCG op added by do_minmax

In the symbolic world, the TCG op described in 6.4.2 has two effects :

- For each element-wise comparison, a path constraint is pushed with a symbol expression that corresponds to the result of the comparison. For example if the comparison `arg1[0] < arg2[0]` is performed, and this comparison is false in the concrete execution, then the path constraint `!(arg1[0] < arg2[0])` should be pushed.
- An output symbol expression is built, that represents a vector where each element is either the corresponding element in the symbol expression for arg1 or the corresponding element in the symbol expression for arg2. For each element, the choice between arg1 and arg2 depends on the result of the comparison in the concrete execution.

6.4.4 Implementation of the instrumentation for do_minmax

To instrument the function `do_minmax`, we have written the symbolic helper function `helper_sym_ternary_vec` and we have added code to `do_minmax` that injects TCG ops for calling this helper function at run time.

`helper_sym_ternary_vec` must both push appropriate path constraints and build a symbol expression for the result of the operation. As we have just seen, both of those task require `helper_sym_ternary_vec` to know the concrete result of each element-wise comparison. To achieve this, `helper_sym_ternary_vec` receives as argument the concrete value of the result of the concrete operation.

Here is the implementation of `helper_sym_ternary_vec`:

Code Source : accel/tcg/tcg-runtime-sym-vec.c

```
1 /*
2  * Symbolic equivalent of an SIMD ternary operation.
3  *
```

```
4  * This function performs the symbolic equivalent of computing an output vector of
   ↪  the form :
5  * result[i] = arg1[i] <comparison_operator> arg2[i] ? arg1[i] : arg2[i]
6  * where <vector>[i] denotes the ith element of the vector <vector>.
7  *
8  * For each element, the concrete result is used to determine if the ternary
   ↪  condition was true or false,
9  * and path constraints are pushed accordingly.
10 *
11 * Args
12 *      arg1_concrete, arg2_concrete : pointers to buffers that store the concrete
   ↪  values of the input vector operands
13 *      arg1_symbolic, arg2_symbolic : symbolic expressions of the input vector
   ↪  operands
14 *      comparison_operator : enum value that represents a comparison operator like
   ↪  <, >, ==, etc.
15 *      concrete_result : pointer to a buffer that stores the concrete value of the
   ↪  result vector, as computed by the
16 *          concrete execution of the SIMD operation
17 *      vector_size : size of the vectors in bits
18 *      vece : element size in bits = 8 * 2^vece
19 * Returns
20 *      A symbol expression that corresponds to the symbolic result of the SIMD
   ↪  operation
21 */
22 void *HELPER(sym_ternary_vec)(
23         CPUArchState *env,
24         void *arg1_concrete, void *arg1_symbolic,
25         void *arg2_concrete, void *arg2_symbolic,
26         TCGCond comparison_operator, void *concrete_result,
27         uint64_t vector_size, uint64_t vece
28 ) {
29
30     /* [...] */
31
32     void *arg1_elements[element_count];
33     void *arg2_elements[element_count];
34
35     split_expression(arg1_symbolic, element_count, element_size, arg1_elements);
36     split_expression(arg2_symbolic, element_count, element_size, arg2_elements);
37
38     /* For each element, the condition of the ternary was true iff the element of the
   ↪  result is equal to the element
39      * of arg1. */
40     int concrete_condition_was_true[element_count];
41
42     for (int i = 0; i < element_count; i++) {
43         void *result_element_ptr = element_address(concrete_result, i, element_size,
   ↪  vector_size);
```

```
44          void *arg1_element_ptr = element_address(arg1_concrete, i, element_size,
    ↪ vector_size);
45          concrete_condition_was_true[i] = memcmp(result_element_ptr, arg1_element_ptr,
    ↪ element_size / 8) == 0;
46      }
47
48      for (int i = 0; i < element_count; i++) {
49          build_and_push_path_constraint(
50                  env,
51                  arg1_elements[i],
52                  arg2_elements[i],
53                  comparison_operator,
54                  concrete_condition_was_true[i]
55          );
56      }
57
58      void *result_expression = concrete_condition_was_true[0] ? arg1_elements[0] :
    ↪ arg2_elements[0];
59      for (int i = 1; i < element_count; i++) {
60          result_expression = _sym_concat_helper(concrete_condition_was_true[i] ?
    ↪ arg1_elements[i] : arg2_elements[i], result_expression);
61      }
62
63      g_assert(_sym_bits_helper(result_expression) == vector_size);
64      return result_expression;
65  }
```

The eluded part a line 30 is similar to the beginning of function `build_expression_for_vector_vector_op`. It checks if operands are concrete, and creates a symbol expression for a concrete operand if necessary.

At lines 32 - 36, we build symbol expressions for each element of the input vectors.

At lines 38 - 46, for each element of the concrete result vector we check if the condition of the ternary operation was true. This is done by checking if the element of the result is equal to the corresponding element of arg1. `element_address` is a simple util function that computes a pointer to an element inside a vector.

At lines 48 - 56, we push a path constraint for each ternary condition by calling the function `build_and_push_path_constraint`. This function was discussed in 4.5.

Finally, at lines 58 - 61 we build the returned symbol expression that represents the output vector. The returned symbol expression is the result of the concatenation of the elements of arg1 or arg2. For each element, we take arg1 if the ternary condition was true or arg2 otherwise, in order to build a symbol expression that corresponds to the concrete version of the execution.

Chapter 7

Results

In this chapter, we are going to discuss the correctness and the usefulness of our new SymQEMU version.

We will first evaluate if the new SymQEMU gives as good results as the old SymQEMU, when executed on the `asprintf` test program. For this, we will compare and analyze the symbolic traces generated by the two versions of SymQEMU, in order to determine if the new SymQEMU is able to generate the same symbol expressions as the old SymQEMU. This will be done in Section 7.1. In 7.2, we will also show the test cases generated by the two versions of SymQEMU on the `asprintf` test program, and we will discuss the differences.

Then, in 7.3 we will see program examples where the new version of SymQEMU outperforms the old one. We will show that, even on simple programs, situations can easily happen where the old SymQEMU is not able to generate correct outputs, while the new one does.

7.1 Comparison of the symbolic traces for the asprintf test program

In this section, we will compare the execution traces generated by the old and the new SymQEMU, when run on the `asprintf` test program.

In 7.1.1, we will describe the comparison methodology used. In particular, we will explain which criteria we will use to determine if the implementation of the new SymQEMU is incorrect. In 7.1.2, we will present the result of the comparison. Finally in 7.1.3 we will give an interpretation of the results obtained.

7.1.1 Comparison methodology

After instrumenting the vector TCG instructions, it is expected that we get a version of SymQEMU that has the following behavior, when compared to the old SymQEMU :

- Non SSE instructions should have an identical effect on the symbolic state for the old and the new SymQEMU.
- `Type 1` SSE instructions should have an identical effect on the symbolic state for the old and the new SymQEMU.
- `Type 2` SSE instructions should have a different effect on the symbolic state between the two versions of SymQEMU. When executing those instructions, the old SymQEMU looses symbol expressions or propagates wrong expressions, while the new SymQEMU should handle those instructions correctly.

Recall that we have defined `type 1` and `type 2` instructions in 5.4.1.

The fact that `type 2` instructions can lead to the generation of different symbol expressions in the old and in the new SymQEMU makes difficult to verify that `type 1` instructions and non vector instructions have the same behavior in both versions of SymQEMU. When comparing the symbolic traces, there is no easy way to see if the differences in the symbolic state are only caused by `type 2` instructions, as a difference induced by a `type 2` instruction can propagate and cause differences in the subsequent steps of the execution.

In order to make this validation possible, we did the following : we created a modified version of the old SymQEMU, that concretizes the destination XMM operand when an SSE helper function is executed. As our evaluation is limited to the `asprintf` test program, we did this only for SSE instructions that are called with a symbolic operand during the execution of this program. Starting from now and for the rest of this section, when referring to the old SymQEMU, we specifically refer to the modified old SymQEMU with this added concretization.

Now, if our implementation of the new SymQEMU is correct, the only differences between the symbolic traces of the two versions should be some symbol expressions created by `type 2` instructions in the new SymQEMU, that are not created in the old SymQEMU (and the descendents of those symbol expressions).

As a consequence, we can check the correctness of our implementation of the new SymQEMU by verifying if the following condition is satisfied :

The only differences between the old and the new SymQEMU should be “one way” differences. The new SymQEMU can have symbol expressions that the old does not have, but the old SymQEMU never has symbol expressions that the new does not have.

Note that this condition is necessary but not sufficient to prove that the new SymQEMU implementation is correct. If a `type 1` instruction or a non vector instruction of the new SymQEMU incorrectly generates additional symbol expressions, we will still only see "one way" differences.

7.1.2 Comparison results

We ran both the old and the new SymQEMU on the `asprintf` test program, and we collected the two symbolic traces. All the differences that we observed were the following :

Many times during the execution, a symbol expression appears in the symbolic state of the new SymQEMU, with no equivalent symbol expression in the old SymQEMU. Each time this arrives, the following happens : the symbol expression is immediately involved in a path constraint pushed by the new SymQEMU. Then, the symbol expression does not persist in memory during the execution steps that follow the one where the path constraint was pushed.

Additionally, we observed that no symbol expression exists in the old SymQEMU and does not exist in the new SymQEMU.

We also compared the path constraints pushed by the two versions during the execution.

The old SymQEMU pushes 24 path constraints, while the new SymQEMU pushes 216 path constraints.

The path constraints of the old SymQEMU is a strict subset of the path constraints of the new SymQEMU : all 24 path constraints pushed by the old SymQEMU are also pushed by the new SymQEMU.

The 192 additional path constraints of the new SymQEMU are pushed during the execution of the following blocks of instructions :

- 16 path constraints are pushed by the block of instructions starting at PC 0x4243c8.
- 16 path constraints are pushed by the block of instructions starting at PC 0x424610.
- The other 160 path constraints are pushed by the block of instructions starting at PC 0x438977. This block is exececuted several times, 32 path constraints are pushed each time.

7.1.3 Interpretation of the results

The fact that the only differences in the symbolic traces appear right before a path constraint is pushed by the new SymQEMU, and do not persist afterwards, implies the following : the symbol expressions involved in the path constraints that only the new SymQEMU pushes

are generated just before the generation of the path constraint, and they do not have any impact on the subsequent path constraints.

As a consequence, all the instructions that trigger a different behavior between the old and the new SymQEMU are executed just before the creation of the path constraints that only the new SymQEMU pushes.

As we know at which moment of the execution each of those path constraints is generated, we can easily look at the corresponding target program assembly code, to find the responsible instructions.

The block of instructions at PC 0x4243c8 and 0x424610 are part of the `memchr` function from the C standard library. It contains several `type 2` SSE instructions, which explains the presence of additional symbol expressions and path constraints in the new SymQEMU.

The block of instructions at PC 0x438977 is part of the `strchrnul` function from the C standard library. Like for `memchr`, `strchrnul` uses several `type 2` SSE instructions.

We can conclude that, according to the criterium discussed in 7.1.1, there is no evidence that our new version of SymQEMU is incorrectly implemented.

7.2 Comparison of generated test cases for the asprintf test program

In this section, we will present the test cases generated by the old and the new SymQEMU on the `asprintf` test program. We will also give a short discussion of the differences observed.

7.2.1 Old SymQEMU

Table 7.1 contains the list of test cases generated by the old SymQEMU,

For each line, the execution step is the number of instruction blocks that were executed at the time of generating the test case. PC is the address of the instruction block that contains the instruction that triggered the generation of the test case. The test cases are displayed as printed Python bytes objects, the non printable characters are denoted by `\x` followed by a hexadecimal value.

Execution step	PC	Test case
2063	0x40c495	`b'\naaaa %d %d %d %d aaaaa\n'`
2177	0x461bb5	`b'aaaaa \x00d %d %d %d aaaaa\n'`
2181	0x461bf9	`b'aaaaa %\x9b %d %d %d aaaaa\n'`
2182	0x461d30	`b'aaaaa %  %d %d %d aaaaa\n'`
2191	0x462490	`b'aaaaa %X %d %d %d aaaaa\n'`

2192	0x47f690	b'aaaaa %X %d %d %d aaaaa\n'
2255	0x462179	b'aaaaa %d \x00d %d %d aaaaa\n'
2256	0x461c40	b'aaaaa %d %{ %d %d aaaaa\n'
2257	0x461d30	b'aaaaa %d %[%d %d aaaaa\n'
2266	0x462490	b'aaaaa %d %X %d %d aaaaa\n'
2267	0x47f690	b'aaaaa %d %X %d %d aaaaa\n'
2330	0x462179	b'aaaaa %d %d \x00d %d aaaaa\n'
2331	0x461c40	b'aaaaa %d %d %{ %d aaaaa\n'
2332	0x461d30	b'aaaaa %d %d % %d aaaaa\n'
2341	0x462490	b'aaaaa %d %d %X %d aaaaa\n'
2342	0x47f690	b'aaaaa %d %d %X %d aaaaa\n'
2405	0x462179	b'aaaaa %d %d %d \x00d aaaaa\n'
2407	0x461d30	b'aaaaa %d %d %d %[aaaaa\n'
2416	0x462490	b'aaaaa %d %d %d %X aaaaa\n'

Table 7.1: Test cases generated by the old SymQEMU.

7.2.2 New SymQEMU

Table 7.2 contains the list of test cases generated by the new SymQEMU.

Execution step	**PC**	**Test case**
2063	0x40c495	b'\naaaa %d %d %d %d aaaaa\n'
2070	0x4243c8	b'a\naaa %d %d %d %d aaaaa\n'
2070	0x4243c8	b'aa\naa %d %d %d %d aaaaa\n'
2070	0x4243c8	b'aaa\na %d %d %d %d aaaaa\n'
2070	0x4243c8	b'aaaaa\n%d %d %d %d aaaaa\n'
2070	0x4243c8	b'aaaaa %d \nd %d %d aaaaa\n'
2074	0x424610	b'aaaaa %d %d %d %\n aaaaa\n'
2074	0x424610	b'aaaaa %d %d %d %d\naaaaa\n'
2074	0x424610	b'aaaaa %d %d %d %d \naaaa\n'
2074	0x424610	b'aaaaa %d %d %d %d aa\naa\n'
2074	0x424610	b'aaaaa %d %d %d %d aaaaa\xf5'
2152	0x438977	b'%aaaa %d %d %d %d aaaaa\n'
2152	0x438977	b'a%aaa %d %d %d %d aaaaa\n'
2152	0x438977	b'aa%aa %d %d %d %d aaaaa\n'
2152	0x438977	b'aaaa% %d %d %d %d aaaaa\n'
2152	0x438977	b"aaaaa 'd %d %d %d aaaaa\n"
2152	0x438977	b'aaaaa %% %d %d %d aaaaa\n'
2152	0x438977	b'aaaaa %d \x00d %d %d aaaaa\n'

2152	0x438977	b'aaaaa %d %% %d %d aaaaa\n'
2152	0x438977	b'aaaaa %d %d %d%%d aaaaa\n'
2152	0x438977	b'\x00aaaa %d %d %d %d aaaaa\n'
2152	0x438977	b'aaaaa %d\x00%d %d %d aaaaa\n'
2177	0x461bb5	b'aaaaa \x00d %d %d %d aaaaa\n'
2181	0x461bf9	b'aaaaa %\xf0 %d %d %d aaaaa\n'
2182	0x461d30	b'aaaaa %o %d %d %d aaaaa\n'
2191	0x462490	b'aaaaa %X %d %d %d aaaaa\n'
2192	0x47f690	b'aaaaa %X %d %d %d aaaaa\n'
2234	0x438977	b'aaaaa %d%%d %d %d aaaaa\n'
2234	0x438977	b'aaaaa %d \xdad %d %d aaaaa\n'
2234	0x438977	b'aaaaa %d %% %d %d aaaaa\n'
2234	0x438977	b'aaaaa %d %d \xdad %d aaaaa\n'
2234	0x438977	b'aaaaa %d %d %% %d aaaaa\n'
2234	0x438977	b'aaaaa %d %d %d %d %aaaa\n'
2234	0x438977	b'aaaaa %d %d %d %d aaaa%\n'
2234	0x438977	b'aaaaa %d %d %d\x00%d aaaaa\n'
2255	0x462179	b'aaaaa %d \x00d %d %d aaaaa\n'
2256	0x461c40	b'aaaaa %d %\xe7 %d %d aaaaa\n'
2257	0x461d30	b'aaaaa %d %? %d %d aaaaa\n'
2266	0x462490	b'aaaaa %d %X %d %d aaaaa\n'
2267	0x47f690	b'aaaaa %d %X %d %d aaaaa\n'
2309	0x438977	b'aaaaa %d %d%%d %d aaaaa\n'
2309	0x438977	b'aaaaa %d %d %% %d aaaaa\n'
2309	0x438977	b'aaaaa %d %d %d \xdad aaaaa\n'
2309	0x438977	b'aaaaa %d %d %d %d aaaa%\n'
2330	0x462179	b'aaaaa %d %d \x00d %d aaaaa\n'
2331	0x461c40	b'aaaaa %d %d %\xe7 %d aaaaa\n'
2332	0x461d30	b'aaaaa %d %d %A %d aaaaa\n'
2341	0x462490	b'aaaaa %d %d %X %d aaaaa\n'
2342	0x47f690	b'aaaaa %d %d %X %d aaaaa\n'
2384	0x438977	b'aaaaa %d %d %d%%d aaaaa\n'
2405	0x462179	b'aaaaa %d %d %d \x00d aaaaa\n'
2406	0x461c40	b'aaaaa %d %d %d %\x7f aaaaa\n'
2407	0x461d30	b'aaaaa %d %d %d %o aaaaa\n'
2416	0x462490	b'aaaaa %d %d %d %X aaaaa\n'
2459	0x438977	b'aaaaa %d %d %d %d%aaaaa\n'
2459	0x438977	b'aaaaa %d %d %d %d %aaaa\n'

Table 7.2: Test cases generated by the new SymQEMU.

7.2.3 Discussion

The new SymQEMU generates a larger number of test cases. This is expected, as whe have seen that many more path constraints are pushed with the new version, due to a better handling of SSE instructions.

The new SymQEMU also seems to generate more relevant test cases.

For instance, at execution step 2332, the old SymQEMU generated the test case `aaaaa %d %d % %d aaaaa` while the new one generated `aaaaa %d %d %A %d aaaaa`. `%A` is a valid format specifier, and only the new SymQEMU was able do discover it.

An other example is the execution step 2407, where the old SymQEMU generated the test case `b'aaaaa %d %d %d %[ aaaaa'` and the new SymQEMU generated `b'aaaaa %d %d %d %o aaaaa'`. `%[` is not a correct format specifier, but `%o` is a possible format specifier.

Note that a test case generated by a concolic executor is influenced by all the path constraints that have been pushed before its generation (see 1.6.3.3). As the new SymQEMU pushes many more path constraints than the old one, we can not easily know the exact causes that explain the differences in the generated test cases.

7.3 Minimal programs that make the new and the old SymQEMU behave differently

As we have explained in 5.3.5, the initial motivation for instrumenting the vector instructions of SymQEMU was to make our port to QEMU 8 able to give as good results as the original SymQEMU, on the `asprintf` test program. However, thanks to this instrumentation the new SymQEMU now outperforms the old one on programs that use SSE instructions.

SSE instructions are common in real world software. For example, passing the flag `-O2` to GCC is enough to make it use loop vectorization [2] when appropriate.

We believe that the incorrect handling of SSE instructions was a non negligible limitation of the old SymQEMU. To support this affirmation, we are going to present very minimalist programs and test the use of the two versions of SymQEMU on them. We will see that, even though those programs are very simple, they can already not be correctly tested by the old SymQEMU, while the new SymQEMU generates useful results for them.

Each of the programs presented below were compiled with GCC 13.2. The only compilation flag used was `-O2`.

7.3.1 Input comparison test program

We have tested the old and the new SymQEMU on the following C program :

Code Source :

```
1  #include <stdio.h>
2
3  #define SIZE 8
4  int main(int argc, char** argv) {
5      char SECRET[SIZE] = {'m', 'y', 's', 'e', 'c', 'r', 'e', 't'};
6
7      char input[SIZE] = {0};
8      char xors[SIZE] = {0};
9
10     for(int i = 0; i < SIZE; i++){
11         input[i] = getchar();
12     }
13
14     for(int i = 0; i < SIZE; i++){
15         xors[i] = input[i] ^ SECRET[i];
16     }
17
18     char result = 0;
19     for(int i = 0; i < SIZE; i++){
20         result |= xors[i];
21     }
22
23     puts(result == 0 ? "You win :)" : "You lose :(");
24
25     return 0;
26 }
```

This program reads 8 bytes from STDIN and checks if they are equal to the buffer `SECRET`. The `for` loop at lines 14 - 16 is compiled to SSE instructions.

Of course, this code is probably not the most natural way to compare an input to a fixed value in C. We have written the program in this way to make sure that the compiler uses SSE instructions. However, we believe that similar code could be found in real world software.

When running this program with the old SymQEMU, and giving in STDIN the input `"aaaaaaaa"`, no test cases are generated.

When doing the same with the new SymQEMU, a testcase `"mysecret"` is correctly generated.

7.3.2 Input computation test program

Here is another example of a simple test program :

Code Source :

```
1  #include <stdio.h>
2
3  #define SIZE 4
4  int main(int argc, char** argv) {
5
6      char input[SIZE];
7      char offsets[SIZE] = {4, 6, 3, 2};
8      char result[SIZE];
9
10     for(int i = 0; i < SIZE; i++){
11         input[i] = getchar();
12     }
13
14     for(int i = 0; i < SIZE; i++){
15         result[i] = input[i] + offsets[i];
16     }
17
18     int sum = 0;
19     for(int i = 0; i < SIZE; i++){
20         sum += result[i];
21     }
22
23     puts(sum == 439 ? "You win :)" : "You lose :(");
24
25     return 0;
26 }
```

This program reads 4 bytes from STDIN, computes a value from them, and compares this value to a magic number. Lines 14 - 16 are compiled to SSE instructions.

When running this program with the old SymQEMU, and giving in STDIN the input `"aaaa"`, no test cases are generated.

When doing the same with the new SymQEMU, a testcase `"Lkyx"` is generated. This result is correct : running the program with the input `"Lkyx"` leads to the condition at line 23 to be satisfied.

7.3.3 memchr test program

In 7.1.3, we have seen that the execution of the function `memchr` makes the new SymQEMU push path constraint the the old SymQEMU does not push.

We have written the following test program, in order to specifically test the behavior of SymQEMU on this function :

Code Source :

```
1 #include <stdio.h>
2 #include <string.h>
3
4 #define SIZE 4
5 int main(int argc, char** argv) {
6
7     char input[SIZE] = {0};
8
9     for(int i = 0; i < SIZE; i++){
10         input[i] = getchar();
11     }
12
13     puts(memchr(input, 'z', SIZE) != 0 ? "You win :)" : "You lose :(");
14
15     return 0;
16 }
```

This program reads 4 bytes from STDIN and checks if any of them is the char `'z'`.

When running this program with the old SymQEMU, and giving in STDIN the input `"aaaa"`, no test cases are generated.

When doing the same with the new SymQEMU, the testcases `"zaaa"`, `"azaa"` and `"aaza"` are generated. This result is correct, as those inputs would satisfy the condition at line 13.

Chapter 8

Conclusion

During this project, we made several contributions to SymQEMU.

We first presented a technical analysis of the design and implementation of this tool, that could be useful for future contributors that want to familiarize themselves with the internal of QEMU and SymQEMU.

We then ported SymQEMU to the most recent stable version of QEMU. We also created a simple end-to-end testing tool, that allows to automatically check if different versions of SymQEMU generate the same outputs. Additionally, we added a symbolic tracing feature to SymQEMU, and we showed how to use it for debugging purposes and for discovering the root cause of a difference in the behaviors of two versions of SymQEMU.

Finally, we instrumented the vector TCG instructions in SymQEMU, which significantly improved the quality of the outputs of SymQEMU when testing software that contains x86 SSE instructions.

8.1 Future work

8.1.1 Testing and improvement of the port to QEMU 8

An important limitation to the work we realized for porting SymQEMU to QEMU 8 is that we only tested it on a small number of simple programs. Further testing of this new version is needed, to validate that no bugs were introduced when performing this port.

In particular, it must be checked that the choice we made in 5.1.2 regarding the load and store helper functions does not pose problem with 32 bits guests.

Additionally, as discussed in 5.1.4, a redesign of the management of symbolic TCG variables may be required if the support for TCG variables of type *i128* is needed in SymQEMU, as

i128 TCG variables conflict with the current design that consists of the symbolic version of a TCG variable immediately following the corresponding concrete version in the array that stores them.

A possible new design could be to have a separate array for symbolic TCG variables that mirrors the array of concrete TCG variables. A symbolic TCG variable at index i of this new array would correspond to the concrete TCG variable at index i of the normal TCG variables array.

8.1.2 End-to-end testing of SymQEMU

The end-to-end testing tool that we presented in 5.2.1 could be improved by adding more test programs to it, as it currently only contains two test binaries.

Another improvement could be to make it run inside a docker container, which would make its execution more deterministic and avoid changes in the behavior of SymQEMU that are due to changes in the execution context.

This tool could then be integrated in the CI/CD pipeline of SymQEMU to protect it from regressions.

8.1.3 Symbolic tracing of the backend

The symbolic tracing feature that we haved added to the backend could also be used for automated testing of SymQEMU. In a way similar to the end-to-end testing tool, it could be useful to record the symbolic trace generated by a reference version of SymQEMU on some known target programs, and compare this trace to the one generated by new versions of SymQEMU to verifiy that they have a correct behavior.

An advantage of this approach compared to the end-to-end tests is that it would allow to more easily find the root cause of a change in behavior of a future version of SymQEMU.

8.1.4 Testing of the instrumentation of vector TCG variables

Like for the port to QEMU 8, an important limit of our work is that we tested the instrumentation of TCG variables only with a small number of programs, mostly the `asprintf` program.

Further testing is required to validate that our implementation of this instrumentation is correct.

8.2 Research about fuzzing and tight branch conditions

Before concluding this report, we would like to briefly discuss some interesting aspects of the current research on fuzzing and symbolic execution.

We have seen that SymQEMU is used to help fuzzers overcome tight branch conditions, by providing to the fuzzer an example of input that leads the execution to the path that is hidden by this tight branch condition.

However, symbolic execution is not the only approach that is used to address this issue.

8.2.1 Overcoming tight branch conditions without symbolic execution

Researchers proposed alternative techniques for overcoming some common tight branch conditions in fuzzing, without the help of symbolic execution.

A possible approach for overcoming comparisons with magic numbers is based on the following observation : in many cases, the program input is not modified before being compared to a magic value. A typical example of this situation is the small program code that we used to introduce the concept of tight branch conditions in 1.2.5

In such a situation, it is easy for a coverage guided fuzzer that instruments the target program to recognize when this program performs a comparison between a value equal to the program input and another value. The fuzzer can then simply try to use the second value as input, which is the magic number. It will then discover that this input leads to execution of new code.

Of course, this approach does not work if some complicated computation is performed on the input before performing the comparison. However, this technique and some slightly more advanced variants of it happened to give good results when testing real-world software [5]. For this reason, simple comparisons to magic numbers like the one presented in 1.2.5 are not an obstacle anymore for modern fuzzers.

8.2.2 Fast generation of mutated inputs while respecting path constraints

Although modern fuzzers can overcome tight branch conditions in some simple situations, symbolic execution with a tool like SymQEMU is still useful to overcome tight path constraints that involve complex computations on the program inputs.

However, an important problem in those situations is that the fuzzer has no “understanding” of the constraints that made SymQEMU generate the input. By blindly applying random mutations on it, the fuzzer may never get new inputs that pass the tight branch condition.

A simple example is a situation where a program receives as input a series of bytes followed by

a corresponding checksum, and executes the interesting part of its code only if the checksum is valid. In this situation, SymQEMU can easily generate an example of input that passes this tight branch condition. However, it is of no use for the fuzzer, because variations of this input obtained by random mutations will most probably not pass the tight branch condition anymore. To take advantage of the input coming from the symbolic executor, the fuzzer would need to know that the last part of the input is a checksum and recompute it when generating input mutations.

To solve this problem, new approaches are needed to allow rapid generation of mutated inputs while satisfying the appropriate path constraints [12].

8.2.3 Acknowledgements

I would like to thank Alexandre Duc, who supervised me throughout this project, and Aurélien Francillon, who made this internship possible by welcoming me to EURECOM.

Bibliography

[1] American fuzzy lop - whitepaper. `lcamtuf.coredump.cx/afl/technical_details.txt`. Accessed: 2023-12-14.

[2] Gcc documentation. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`. Accessed: 2023-12-14.

[3] Qemu translator internals documentation. `https://www.qemu.org/docs/master/devel/tcg.html`. Accessed: 2023-12-14.

[4] Tcg documentation. `https://www.qemu.org/docs/master/devel/tcg-ops.html`. Accessed: 2023-12-14.

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[6] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. FUZZOLIC: mixing fuzzing and concolic execution. *Comput. Secur.*, 108:102368, 2021.

[7] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, 2012.

[9] Nassim Corteggiani and Aurélien Francillon. Hardsnap: Leveraging hardware snapshotting for embedded systems security testing. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 294–305. IEEE, 2020.

[10] Alexandre Duc. High-level software security. University Lecture, 2022.

[11] Stephane Duverger. Qemu internals blog articles. `https://airbus-seclab.github.io/qemu_blog/`. Accessed: 2023-10-1.

[12] Andrea Fioraldi. personal communication.

[13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[14] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[15] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[16] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426, 2007.

[17] Sebastian Poeplau and Aurélien Francillon. Systematic comparison of symbolic execution systems: intermediate representation and its generation. In David Balenson, editor, *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, pages 163–176. ACM, 2019.

[18] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 181–198. USENIX Association, 2020.

[19] Sebastian Poeplau and Aurélien Francillon. SymQEMU: Compilation-based symbolic execution for binaries. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[20] Jonathan Salwan and Florent Saudel. Triton: Framework d'exécution concolique et danalyses en runtime. In *Proceedings of the 2015 Symposium sur la Securite des Technologies de lInformation et des Communications, SSTIC*, volume 15, 2015.

[21] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 571572, New York, NY, USA, 2007. Association for Computing Machinery.

[22] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5):263272, sep 2005.

[23] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

[24] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pages 138–157. IEEE, 2016.

[25] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[26] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 745–761. USENIX Association, 2018.

List of Figures

List of Tables