Assignment of bachelor's thesis

Title:	Symbolic execution for R
Student:	Petr Šťastný
Supervisor:	Pierre Donat-Bouillud, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2024/2025

Instructions

Symbolic execution is a program analysis technique that explores all the paths of a program and solve the various conditions along the paths using a SMT solver. It can create concrete examples of what leads to a particular path, what leads to some bug. Building a symbolic execution engine is usually a long and complex endeavour, especially for interpreted languages. However, in "Prototyping symbolic execution engines for interpreted languages", Bucur et al. 2014, authors present a simple but powerful idea that enabled them to implement symbolic execution for Python and Lua in respectively 5 and 3 days in a tool called "Chef".
The idea is to use an existing symbolic execution engine for binary code, and to run it on the interpreter of the targeted language, e.g. the Python interpreter. It also requires to make the symbolic execution aware of the high-level CFG of the program. To get acceptable performance, the Python/Lua interpreter also has to be unoptimized!
The goal will be to use the same approach for the R language.

1) Research the principles of symbolic execution
2) Describe the key techniques used by symbolic execution engines Klee and S2E, and Chef
3) Familiarize yourself with the R programming language and its uses
4) Design and implement a symbolic execution for the R language using Chef
5) Verify its functionality
6) Discuss the results

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 27 March 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor’s thesis

Symbolic execution for R

Petr Šťastný

Department of Theoretical Computer Science
Supervisor: Pierre Donat-Bouillud, Ph.D.

May 16, 2024

Acknowledgements

I want to thank my supervisor, Pierre Donat-Bouillud, Ph.D., for his support, invaluable advice and for making this project possible.

Furthermore, the entire time, endless support has been coming from my friends and family, who stood with me during my entire studies and helped me focus on this thesis, and I would like to thank them for everything they did for me.

Last but not least, I would like to mention Jan Liam Verter and Andrew Kvapil, who showed me the beauty of functional languages and are the reason why I decided to study this field, and Štěpán Pechman, who aided me with getting access to the hardware I needed and tolerated my use of FIT Gitlab's disk space.

Declaration

In Prague on May 16, 2024

Citation of this thesis

Šťastný, Petr. *Symbolic execution for R.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstract

Symbolic execution is a technique that enables one to test programs and prove nontrivial properties about a program.

Making a new symbolic execution engine is challenging. Instead, a technique is used where an interpreter of the target language is symbolically executed itself, simplifying the symbolic execution of its programs.

In this work, I show how to apply this to the R language. The resulting symbolic execution engine can be used to, for example, test library correctness or generate type annotations.

Keywords Testing, Symbolic execution, R language, S^2E, Chef

Abstrakt

Symbolic execution je technika, která umožňuje testovat programy a dokazovat jejich netriviální vlastnosti.

Vytvoření nového systému pro *symbolic execution* je náročné. Místo toho je použita technika, kdy je symbolicky spouštěn samotný interpret cílového jazyka, díky čemuž je jednodušší symbolicky spustit i programy v daném jazyce.

V této práci aplikuji tuto techniku na jazyk R. Výsledný program je možné použít například na otestování korektnosti knihoven nebo na generování typových anotací.

Klíčová slova Testování, Symbolické spouštění, jazyk R, S^2E, Chef

Contents

List of Figures

Introduction

R is an interpreted language used for statistics and data analysis [1] and has a significant number of third-party libraries. In order to keep things performant, many[1] of those libraries are written in compiled languages, primarily C, C++ and Fortran [2].

If we consider automated analysis of R programs, these characteristics make the language rather hard to reason about. Its dynamic nature makes static analysis complicated, while the inclusion of libraries written in compiled languages means that one needs to analyze both the user source code and the behaviour of compiled libraries and combine that information together. This requires one to reason about the C API of the R interpreter as well. These reasons underpin the decision to explore the symbolic execution approach, which can be implemented in such a way that high coupling among multiple components is not an issue.

Symbolic execution is a technique that allows one to test large projects without having to spend long hours writing tests manually or collecting test cases from real-life usage. Instead, the symbolic execution engine reasons about the program itself based on its source code or its compiled representation. It can generate test cases that might cause problematic behaviour in the program, and because it is guided by the program itself, it tends to be able to find edge cases better than techniques that do not take that into account, such as random fuzzing.

For example, given the following code, a symbolic execution engine would typically generate two test cases, one with $i = 42$ and one with i being set to some other number. It would find an input such that the code fails without the need to find test cases manually and without using completely unguided, random inputs, which could completely miss the edge case presented below.

```
int i = symbolic_input();
if (i == 42) {
    fail();
}
```

Symbolic execution is viable even for very large systems. One fascinating example is a work done by Yang et al. [3], where they checked the Linux kernel

[1]About one-fourth of all packages released in 2020 [2].

using symbolic execution. The symbolic execution engine managed to generate concrete examples of *ext2*, *ext3* and *JFS* filesystems that, when mounted, caused a kernel panic or buffer overflow. This furthermore shows that even in the context of heavily used and well-tested programs, symbolic execution can uncover bugs and vulnerabilities that were not found before.

Most symbolic execution engines currently widely used, such as S^{2}E, Klee or SAGE, process either machine code or LLVM IR. Therefore, those engines are suitable for languages that can be compiled into machine code or LLVM IR, such as C or C++, but not so much for interpreted languages, such as R.

Writing my own symbolic execution engine would be a lengthy endeavour. It would require implementing all the instructions, modelling how environments are handled in R, and implementing complex optimizations, to reach an acceptable level of performance.

But what if I used an existing symbolic execution engine on an executable representation of the semantics of the language? That is exactly what Chef does. Chef is an S^{2}E plugin and essentially works by symbolically executing the interpreter – in R case, conveniently written in C – using S^{2}E, which is fed the actual R program as input. This practically guarantees bug-for-bug compatibility and is much easier to implement. The Chef authors claim that writing a new symbolic execution engine this way should be possible in a rather short time frame [4] while reusing the optimization techniques already made by others.

Goals

The goal of this thesis is to implement a symbolic execution engine for R using the technique presented by Chef authors, leveraging an already existing symbolic execution engine without the need to reimplement it from scratch.

I will describe the key principles of symbolic execution engines Klee and S^{2}E[2] in chapter 2 and focus on the way Chef works. After that, in chapters 3 and 4, I will provide an overview of the inner workings of the R interpreter and modify it to enable symbolic execution. Finally, in chapter 5, I will verify the functionality and show an example of using the symbolic execution engine to test real-life R libraries and generate type annotations.

Thanks to this tool, one will, for example, be able to test specific software and libraries by writing symbolic tests, do a large-scale analysis of the R ecosystem or generate test cases to further enhance manual testing already present in many R libraries.

[2]Which itself is a platform built upon Klee, extending it and adding more capabilities.

CHAPTER 1

Background

In the first part of this chapter, I explain the basics of symbolic execution, emphasise how it differs from other testing techniques and show a few interesting examples where symbolic execution was used with great success.

In the next section, I present the R language and focus more on what makes it an interesting choice for some kind of analysis. I also highlight some relevant parts of the inner design of the interpreter that will be important later on.

1.1 Symbolic Execution

Symbolic execution is a technique used to analyze programs in a way that may uncover bugs that might otherwise remain unseen. Some commonly-used techniques, such as unit tests and fuzzing, tend to be rather time-consuming or may not uncover all edge cases. For example, unit tests heavily depend on the ability of a programmer to exhaust all interesting cases. At the same time, fuzzing is often more or less randomized and might not hit rare edge cases at all. Symbolic execution, on the other hand, is actively guided by the code of the program [5] and will enumerate all inputs, thus testing every single edge case, assuming it is given enough time and computing resources.

Symbolic execution works by representing the values of program variables as symbolic values. Let us denote a *symbolic store* of variables as $\sigma : I \mapsto V$, where I is the set of identifiers of program variables and V is the set of symbolic values, denoted $\alpha_1, \ldots, \alpha_n$.

For example, consider the following program written in C:

```c
int i, j;
i += 4;
i *= 2;
i += j;
```

An initial symbolic value α_i is assigned to every given variable. Later, when the variable is modified, the symbolic value is modified equivalently.

The symbolic store will have the following values as the program is executed:

Statement	σ
1	$\{i \mapsto \alpha_1, \quad j \mapsto \alpha_2\}$
2	$\{i \mapsto \alpha_1 + 4, \quad j \mapsto \alpha_2\}$
3	$\{i \mapsto 2 \cdot (\alpha_1 + 4), \quad j \mapsto \alpha_2\}$
4	$\{i \mapsto 2 \cdot (\alpha_1 + 4) + \alpha_2, \quad j \mapsto \alpha_2\}$

Suppose a conditional statement is encountered later in the program.

```
int i, j;
i += 4;
i *= 2;
i += j;
if (i < 0) {
    // Branch 1
} else {
    // Branch 2
}
```

In this case, branch 1 is reachable if and only if there exists some assignment of concrete values to α_1, α_2, such as the value of variable i is negative, so $\sigma(i) = (\alpha_1 + 4) \cdot 2 + \alpha_2 < 0$. Branch 2 is reachable if the variable's value is not negative. Whether some α_1, α_2 exist, satisfying either of the conditions (or both) can be solved by a satisfiability modulo theories (SMT) solver. Using that, the symbolic execution engine can determine whether a branch can be taken and ignore it if it cannot be executed given the current variable mapping σ.

When a symbolic execution engine is executing the code and comes across a conditional jump,[3] it has to decide what to do next. Take the true branch or the false branch? The symbolic execution engine will do both. Until now, there was exactly one *state* containing, for example, the mapping σ. When a situation like this happens, the engine forks the current state into two: one that represents the program taking a true branch and the second one representing the contrary. From then on, the states will be processed independently of each other, even when the two branches merge.

However, there is one important distinction between Branch 1 and Branch 2. In the first branch, we know that `i` is negative, while in the second branch, the opposite is true. This knowledge should be encoded into the symbolic states as well in order to properly resolve any further queries.

This is where *path constraints* come into play. It is a set of conditions that has to be true since the branch has been taken. This set limits the values of symbolic variables. In this case, for branch 1, the path constraints are represented by a set of equations: $\pi = \{2 \cdot (\alpha_1 + 4) + \alpha_2 < 0\}$. When additional code is executed in the branch, these conditions limit the values of symbolic variables.

The biggest advantage of storing π is the ability to reason which values variables can or cannot have. For example, there could be an assertion somewhere in our code. Should a symbolic execution engine come across it, it can simply run an SMT solver and determine whether, given current *path constraints* π, is there some initial assignment of values to variables $\alpha_1, \cdots, \alpha_n$, such that the

[3]For example, an `if`.

condition in assert evaluates to be false. Similar conditions can be checked, for example, during array access, to verify that the program will never access elements out of bounds. Thanks to this, we can prove many interesting properties about the program.

While useful, the extensive use of SMT solvers makes symbolic execution quite slow [6], especially when non-linear operations are encountered. Those generally cannot be solved quickly by current SMT solvers. One example of non-linear operations is hashing [4], typically used in *set* and *map* implementations in modern programming languages.

Another large problem is *state space explosion*[4] [5] – common language constructs, such as loops, increase the number of states exponentially, making it very costly to analyze any programs heavily utilizing them. There is not a single silver bullet for addressing these issues; some sacrifices on soundness or completeness may be necessary in order to be able to analyze larger programs.

As an example, suppose there is a loop in the code:

```
int i;
int sum = 0;
while((i = input()) != 4) {
    sum += i;
}
// end
```

In every iteration, the current state is branched into two. One where $i = 4$ and one where $i \neq 4$. This results in an infinite number of states generated – every time the loop is executed, two states are generated: one that falls out of the loop and another one that returns to the beginning of the loop. See Figure 1.1 for a visualization that might help build up some intuition regarding the symbolic execution of that loop.

This brings another interesting problem to the table. Suppose there are multiple states that the symbolic execution engine needs to explore. How to decide which one to explore first? There are many strategies that tackle this problem. One particularly interesting strategy in this case is class-uniform path analysis (CUPA), as it is used by the Chef itself [6].

CUPA works by grouping the states by a heuristic and randomly selecting a group to be executed rather than a state [4]. As an example, suppose states are grouped by the function they belong to. If CUPA was not in place, functions with a higher number of states would be more likely to be explored. However, with CUPA, all functions have an equal probability of being chosen.

Writing a custom symbolic execution engine itself is not hard. When one does not care about optimization, interaction with the environment,[5] a symbolic execution engine can be written in a short amount of time. However, once one begins to tackle real-life issues, such as speed, path explosion, or the fact that many programs interact with the environment, even if it is just writing and reading files, the amount of work required begins to grow tremendously.

[4]Also called *path explosion*

[5]As an example, some symbolic execution engines have their own version of standard library functions or attempt to model, for example, writing and reading to a file. This is further discussed in sections 2.2 and 2.3, which describe the way this is handled by two symbolic execution engines.

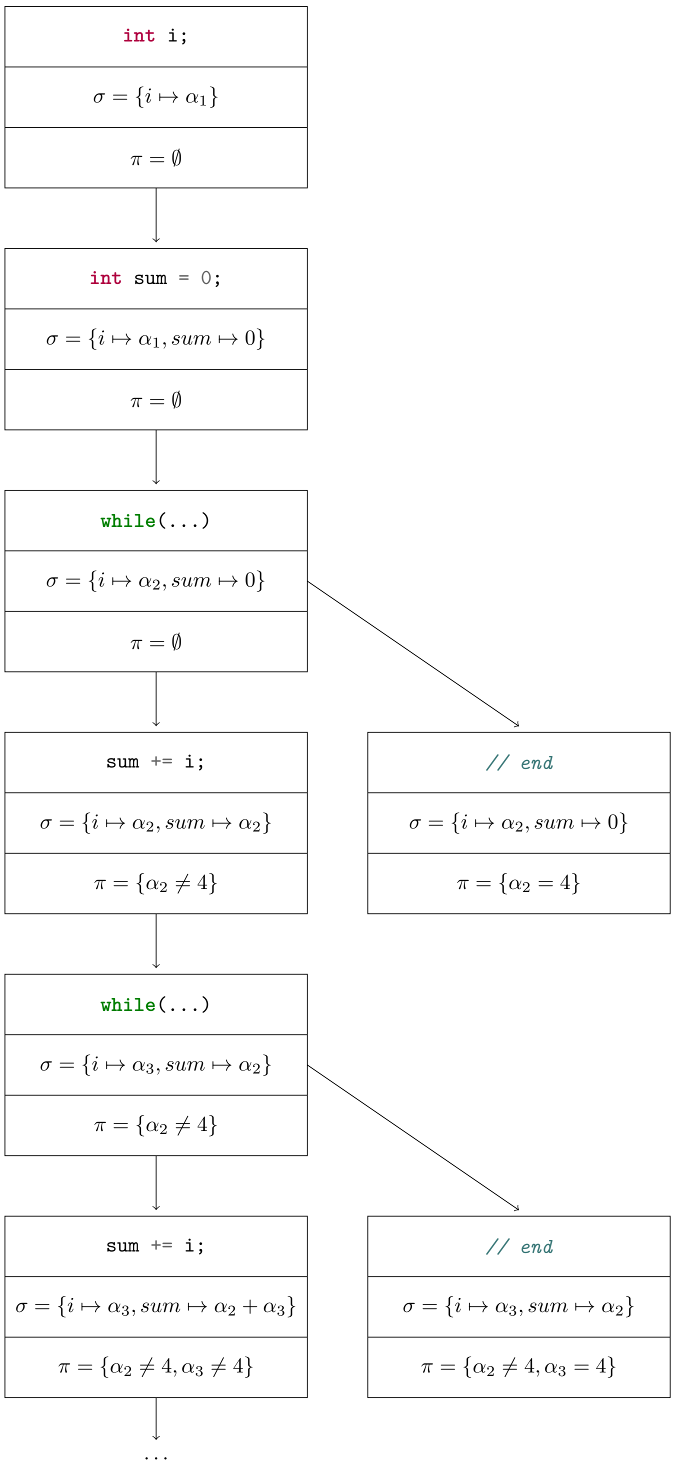
int i;
$\sigma = \{i \mapsto \alpha_1\}$
$\pi = \emptyset$
int sum = 0;
$\sigma = \{i \mapsto \alpha_1, sum \mapsto 0\}$
$\pi = \emptyset$
while(...)
$\sigma = \{i \mapsto \alpha_2, sum \mapsto 0\}$
$\pi = \emptyset$
sum += i;
$\sigma = \{i \mapsto \alpha_2, sum \mapsto \alpha_2\}$
$\pi = \{\alpha_2 \neq 4\}$
// end
$\sigma = \{i \mapsto \alpha_2, sum \mapsto 0\}$
$\pi = \{\alpha_2 = 4\}$
while(...)
$\sigma = \{i \mapsto \alpha_3, sum \mapsto \alpha_2\}$
$\pi = \{\alpha_2 \neq 4\}$
sum += i;
$\sigma = \{i \mapsto \alpha_3, sum \mapsto \alpha_2 + \alpha_3\}$
$\pi = \{\alpha_2 \neq 4, \alpha_3 \neq 4\}$
// end
$\sigma = \{i \mapsto \alpha_3, sum \mapsto \alpha_2\}$
$\pi = \{\alpha_2 \neq 4, \alpha_3 = 4\}$
...

Figure 1.1: Symbolic execution tree showcasing a loop

Each box is a symbolic execution state at some point in time. The first line is the relevant line of code, followed by a set of symbolic variables σ and a set of path conditions π on the next lines.

Because of this, there are many symbolic execution engines available that can be used and extended, for example, Klee and S^{2}E,[6] which are the foundation that Chef builds upon, and I focus more on them in sections 2.2 and 2.3.

It is also important to note that symbolic execution can be used not only to test software but also to extract its behaviour in general. A great example of that is work done by N. Hasabnis and R. Sekar [8], who managed to extract assembly instruction semantics from the code generation implementation in various compilers.

The key thought was that the instruction semantics are already present in a well-defined form: in the source code of various compilers. The code generators of compilers in question perform the task of translating IR[7] to assembly. All that is needed is inverting the code generator to generate IR out of assembly instead, thus retrieving precise semantics of instructions that the code generator is able to emit. The authors indeed managed to accomplish this and extracted semantics of all instructions supported by the *GCC* code generator for both *x86* and *AVR* architectures.

1.2 R

R is a language used widely in the field of statistics and data analysis; the *Introduction to R* lists good support for data manipulation and analysis as one of its core features [1]. However, the language itself contains some quirks and behaviours that might be surprising to a novel programmer and may often cause issues [9].

One of the features that might be surprising to a programmer is the various ways of indexing collections in R. There are three subsetting operators (`[`, `[[` and `$`) [10] that may be used. Each of them behaves in a slightly different way, even depending on which collection is being used [11, Chapter 4].

In R, there are multiple data collections, as described in [1]:

- *Vectors* are collections of basic data types – booleans, numbers and strings. There are ways to store more complicated types, such as dates, in vectors as well.
- *Matrices* (or, more generally, *arrays*) are multidimensional vectors that can be indexed by two or more indices.
- *Factors* are vectors that only contain predefined values. One can think about it as a collection of enum values.
- *Data Frames* are essentially matrices, but the columns may have different types.
- *Lists* are collections that allow any type of value, even different ones, in the same collection.

[6]Which itself is built upon Klee and enhances its abilities. S^{2}E authors put a large amount of work into the symbolic execution engine to handle programs interacting with the environment. As a result, it is able to even symbolically execute large programs such as MS Office [7].

[7]LLVM intermediate representation, a language similar to assembly used by LLVM-based tools due to it being relatively easy to process.

- *Environments* are essentially hashmaps, used primarily for the purpose of scoping. Each scope has an associated environment with a link to an environment of a parent scope. When the programmer tries to access a variable, its name is searched for in the environments.

For example, consider the behaviour of the `[` operator when the item does not exist in a collection:

```
> list(1,2,3)[4]
[[1]]
NULL

> c(1,2,3)[4]
[1] NA
```

Elements can be accessed not only by indices but also by names (strings). For example, the last item in the list can be accessed both by index `4` and name `cd`:

```
> x <- list(1,2,3,cd=4)

> x[4]
[1] 4

> x["cd"]
[1] 4
```

The main difference between `[` and `[[` is that the former returns possibly more than one element; the latter can only return one. The latter also drops the names of the values if there are any; the former does not. Furthermore, the result of `[` will be of the type of the original object (in the example below, a list), while the result of `[[` will be an atomic vector. See the following example:

```
> x <- list(1,2,3,cd=4)

> x[4]   # returns value 4 named "cd" (in a list)
$cd
[1] 4

> x[[4]]   # returns value 4, not named (in an atomic vector)
[1] 4
```

Both `[[` and `$` can be used to get a single value from a collection. The difference is that `$` does not work on vectors, allows only indexing by names and performs *partial matching* – it selects an item if the prefix of the name matches, as long as there is precisely one match. `[[`, on the other hand, selects a value only if the name matches exactly. For example, consider the following:

```
> x <- list(1,2,3,cd=4)

> x[["cd"]]
[1] 4

> x$cd
[1] 4

> x$c
[1] 4

> x[["c"]]
NULL
```

Note that operator `[[` returns *NULL* if the value was not found. However, lists can contain valid *NULL* values. It is not enough to check if a name exists in a list by checking whether `[[` returned *NULL*. The programmer has to use, for example, `"c" %in% names(x)`.

Another feature that might cause problems is the C API of the R Interpreter. As I mentioned earlier, about one-fourth of all packages ever released (and a great number of the most popular ones) are written in some other language, most often C/C++ or Fortran [2]. Writing those libraries means that the developer has to interact with the C API. Even when one ignores all the potential issues that might occur as a consequence of writing lower-level code, the internals of the R Interpreter are sometimes hard to understand, even with the help of official documentation.[8]

Due to the interpreter being written in C, there is little actual compile-time type-checking going on. For example, every value in R is composed of one or more `SEXP`s, which is similar to abstract syntax tree (AST) nodes found in other languages. There are many types of nodes – be it a logical value, vector or an `S4` class. There are functions and macros that are designed to work with a specific type of `SEXP`. However, because R is written in C, the type checker does not guard against passing incorrect `SEXP`s to a function. This has the potential to create obscure bugs that might be quite challenging to find. Furthermore, some of the macros can be challenging to understand due to the hard-to-find documentation, and their usage might have to be inferred from the usage in the source code of the R interpreter, which may cause the programmer to use them incorrectly. This can have far-reaching consequences regarding the safety and correctness of the code.

Another feature that might be tricky to tackle is the R garbage collector. Essentially, the R interpreter has a list of objects that are not to be removed. Any objects found in that list (or objects that are transitively referenced by those ones) are not removed when the garbage collector is called, which may happen any time an R interpreter internal function is called.

This means that the user is required to call the `PROTECT` and `UNPROTECT` macros whenever they are working with temporary values that have not been assigned anywhere yet – those macros place or remove the objects from the

[8]It should be noted that there exist projects such as `Rcpp` that offer their own interface, so the developer might not need to interact with the C API of the interpreter in all cases.

aforementioned structure, protecting them from the garbage collector. If one were to forget to call `PROTECT`, a value might be garbage-collected, and the user might attempt to use freed memory. I present the overall design of the R interpreter and offer a more detailed explanation of the garbage collector in section 3.2.

All those minor differences and edge cases can make for surprising behaviour when a programmer is coming to R from a different language and has not read the language reference properly. This makes R quite an attractive target for code analysis tools that could find mistakes and hidden bugs, especially if the tools could operate on a large scale.

Furthermore, it is challenging to try to explore R with a purely static analysis approach. It is a dynamic language, offering ways to manipulate the AST programmatically and having an `eval`, which is used in many libraries[9] and makes static analysis very challenging. As R lacks macros, `eval` is often used for various purposes that could be handled by macros in other languages due to the ability to construct AST and later execute it at runtime.

Code containing an `eval` can be especially hard to analyze statically due to the fact that `eval` can not only perform side effects, but it can even access environments across the entire call stack – in other words, it can modify variables even in scopes of functions that called the current one [12]. x All these reasons underpin the motivation to explore the symbolic execution approach on R. Since it essentially executes the code, it can handle quite well even highly dynamic features such as eval or modification of various scopes throughout the program.

[9]As of 2021, 22.6% of R packages use `eval` [12].

Chapter 2

Related work

Symbolic execution is conceptually similar to property-based testing[10] – it executes a program with many computer-generated inputs and tries to find one that will cause the program to fail or violate a human-written property the code should uphold.

Thus, in this chapter, I will focus not only on symbolic execution tools but also outline the basics of property-based testing, a technique that is used in a very similar way to the way symbolic execution can be used. In section 2.1, I will highlight Hypothesis, a Python testing tool that recently introduced support for using symbolic execution during property-based testing itself. Due to all of these, I believe this tool deserves mention in this chapter.

After that, the reader will be presented two symbolic execution engines – Klee and S^2E in sections 2.2 and 2.3, which are crucial for Chef itself, as Chef is an S^2E plugin, and S^2E itself builds upon Klee and enhances its abilities. When the reader has enough context about those symbolic execution engines, I will finish this chapter with an overview of Chef in section 2.4.

2.1 Property-based testing

Property-based testing is a technique simmilar to fuzzing. It runs the target program with random input. Still, unlike plain random fuzzing, the user is expected to write down properties of the system being tested, and the property-based testing software attempts to find an input that breaks those properties. The key feature is that after finding a bug, it further attempts to refine the input found in order to reduce its complexity and find the smallest breaking example possible [13], which allows the user to efficiently understand the bug.

One of the well-known tools used is Hypothesis [14], a Python tool, which I decided to use here as an example to show the usage of property-based testing. First of all, let me outline the usage of Hypothesis on an example.

Since Hypothesis essentially ”fuzzes” functions, one needs tests that do not check specific input/output pairs but rather general properties.[11] For example, one can test a pair of encoding and decoding functions and verify the property

[10]And fuzzing in general.

[11]While fuzzers might typically verify general properties such as correct memory usage, Hypothesis uses user-defined properties describing expected behaviour of the program.

that the decoding function is the inverse of the encoding function, that is $\forall x \in D_{\text{encode}} : \text{decode}(\text{encode}(x)) = x$.

Quoting the Hypothesis docs [15] on this example, one can write the following to test the property highlighted above:

```python
from hypothesis import given
from hypothesis.strategies import text


@given(text())
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```

Hypothesis fuzzes the test function with random strings. If it finds an issue, it starts to simplify the found counterexample. It does so by following a number of reduction rules, essentially deleting various parts of the input until it can no longer find a smaller input that serves as a counterexample for the test. This way, Hypothesis[12] can be used to easily fuzz code and find counterexamples that are as small as possible.

This way of testing is simultaneously very similar to the way symbolic execution can be used to test programs since it essentially inputs random values into a program as well. The difference is that symbolic execution uses the program source code to guide its exploration, while property-based testing is randomized.

Furthermore, Hypothesis recently introduced an experimental feature that allows one to use custom backends, such as one that utilizes symbolic execution [15, Projects extending Hypothesis → Alternative backends for Hypothesis]. This has the potential to bring symbolic execution of an interpreted language to a widespread audience and definitely deserves a mention in this section.

It is still in a very early stage of development as of now, but it is still quite interesting to look at. It would essentially mean that Hypothesis would no longer pick inputs entirely at random, but those could be chosen according to the program that is being tested itself.

2.2 Klee

Klee is a symbolic execution engine designed for symbolic execution compiled programs. It comes with support for modelling the Linux environment, allowing it to be easily used even for programs that interact with the environment. It has been successfully used to test GNU coreutils and uncover 56 bugs [16]. Klee has a good track record with its proven capability and all its achievements, while being rather easy to read and understand, which makes it perfect for building upon and learning more about real-world symbolic execution engines.

Klee works by modelling the whole program and acting as an interpreter of the compiled code. For each state, Klee models registers, stack, heap, PC and a set of path conditions.[13] An interesting problem is modelling memory.

[12]Or pretty much any other property-based testing library

[13]The reader can refer to section 1.1 for a quick overview regarding states and path conditions. Klee works in a very similar way to the simple example described there.

One of the straightforward implementations would simply be a single array of bytes, which closely mimics the semantics of real programs. However, as Klee authors discovered, the SMT solver of their choice was not able to solve resulting queries.[14] So Klee works by simply having an array of objects – so, in principle, whenever a symbolic code works with some object, the solver only handles the memory space of the given object and does not need to worry about the rest of the memory. This was found to drastically improve performance [16].

During interpretation, Klee automatically checks for the validity of insecure operations. For instance, in the case of load or store instructions, it checks whether the address is within a valid object. During division, it verifies whether the divisor is always non-zero. If any of these checks were to fail, Klee would generate a test case that triggers the error. Thanks to this, Klee can validate certain properties of a program without the need for the user to do anything.

As described in section 1.1, one of the biggest challenges of symbolic execution is the need to solve SMT formulas, often complex ones or a large number of them, which tends to be quite slow. Klee authors discovered a few techniques that can drastically reduce the runtime of the solver they used. They do so by simplifying the queries according to those heuristics: They simplify simple equations, such as $2 \cdot x - x = x$, or remove obviously truthful conditions (or conditions that are not relevant for the current query) from the set of path conditions.[15] Similarly, over time, some variables will have known concrete values; there is no need to represent them as symbolic, and all equations containing those variables can be simplified. Furthermore, the fastest solver is the one that doesn't run, so Klee authors keep track of known-unsatisfiable queries and do not attempt to run the solver when the result is already known.

Readers are encouraged to read the Klee paper [16] to learn more about the optimization techniques employed. It is worth noting that despite their apparent simplicity, these techniques significantly enhanced performance. Notably, solver runtime decreased from 92% to just 41% of the total runtime.

Another recurring challenge encountered is the state selection process. This is a topic that will persist through the discussion about Klee, S^{2}E and even Chef itself later. As I mentioned earlier, there are often many states active at one time, even when symbolically executing relatively simple and small programs, let alone large ones like the entire R interpreter. State selection is thus an important topic that can heavily affect the results of the experiments.

Klee handles this by implementing strategies for choosing which states to execute [16].

Random Path Selection stores states in the form of a binary tree. When some state is to be selected for execution, Klee traverses the binary tree randomly, selecting a state to execute. This has the property that the sizes of a subtree of each state have no effect on whether a state is selected or not during the traversal of the tree. For example, if there were two states, one containing a loop that generates a large number of states, the first state would be selected just as often as the loop, which makes the execution less likely to get stuck in a heavily forking code.

[14]At least not efficiently enough so it is usable. And neither was any other SMT solver they were aware of [16].

[15]For instance, in the context of $x = 5$, the condition $x < 10$ becomes redundant and can be safely removed.

Coverage-Optimized Search chooses a state that is likely to cover new code soon. The likeliness to cover new code itself is a heuristic that Klee computer based on the distance to an uncovered instruction, the call stack and whether the state recently uncovered a new code. Based on this, Klee assigns each state a weight representing the likelihood. After that, a state to execute next is selected randomly based on that.

The last thing I would like to cover in Klee is environment modelling. Based on the text written so far, the reader should know how pure functions are symbolically executed. However, the sad state of affairs is that real-life code rarely consists of pure functions only, and some modelling of the outside world is required for symbolic execution to be truly practical.

Klee authors have chosen to model system calls required to run coreutils. They could have modelled a subset of the C library itself, but that would be a nontrivial endeavour that might introduce more bugs. Simply modelling syscalls, which is much simpler, has the advantage of less code being written and, thus, being easier to do. The result is that the authors only had to implement about 40 syscalls[16] to test the GNU coreutils [16].

Despite Klee's relative simplicity, the authors found ten bugs in coreutils, all while keeping the codebase clean and easy to read, which is a great accomplishment.

2.3 S^{2}E

S^{2}E [7] is a symbolic execution tool built upon Klee. However, unlike Klee, S^{2}E has quite a different way of working and, thanks to that, enables much more interesting analysis than Klee did. Most importantly, it is able to work with huge codebases. People even managed to analyze large programs such as Microsoft Office with it.

S^{2}E no longer executes just the program it is supposed to test – it executes the whole operating system. Thanks to this, worrying about the environment modelling I covered in the previous section is no longer necessary. The operating system itself is the model. This has the advantage of being able to test software with its real dependencies – be it other programs, libraries or drivers. Even proprietary programs with no source code available can be easily tested this way. This works by executing the whole operating system within a customized QEMU.

However, as the reader might have already thought, this seems utterly unreasonable performance-wise. It is very costly to symbolically execute even simple programs, let alone the whole operating system. The reader is correct with this thought – S^{2}E solves this problem by symbolically executing only some parts of the system, those that are interesting enough to reason about.

Essentially, S^{2}E switches between concrete and symbolic execution as it traverses the code. The key observation is that the program launched by the user is interesting, and we want to explore it thoroughly. Still, the OS kernel or libraries that it uses are of no particular interest to us. So when S^{2}E runs the program of interest, it begins executing it symbolically. When calling into a library or the kernel, it stops executing the program symbolically and begins

[16]Which took only about 2,500 lines of code.

executing it concretely. After the execution returns to the program, it starts executing symbolically again. S^{2}E authors call this technique *selective symbolic execution* [7].

In S^{2}E, there are multiple modules (and more can be easily written) that affect which code is to be executed symbolically. However, the default one[17] defines the binary that the user requested to be analyzed as viable for symbolic execution and everything else as non-viable.

There is one problem here, though, that the reader might think about. What about losing precision when switching from symbolic execution to concrete and back? It is necessary to understand how S^{2}E implements switching between symbolic and concrete execution first.

Symbolic to concrete transition happens when, for example, a library is called from within an application that one wants to observe. Let x be a symbolic variable. Calling a function foo(x), which should not be executed symbolically, requires x to be concretized.[18] S^{2}E chooses some value according to current path conditions, for example, $x = 4$. The path constraints are updated to reflect that $x = 4$. When the function finishes, the return value is saved. Since the path constraints now contain the condition $x = 4$, both the output of the function and any side effects that might have happened are perfectly valid.

The obvious problem is that this just limited the possible range of values that x might have to just a single one. This is solved by dividing path conditions into two categories. Path conditions that were added into path constraints based on the process described above are called *soft constraints*, the other ones[19] are called *hard constraints.* The difference is that whenever S^{2}E would not enter a path because of a *soft constraint*, it knows that it could have gone there if it had chosen a different concrete value during some concretization process in the past. So, in this case, S^{2}E does just that: it backtracks to the execution state where concretization happened and chooses a different value[20] [7].

Concrete to symbolic transition is quite intuitive. Consider a call bar(y) from a concrete context into a symbolic one. The function itself is executed symbolically, with y being considered a symbolic value. In parallel to this, the function is executed with the original concrete argument as well in order for S^{2}E to be able to continue the concrete execution later when the function returns.[21] The result of this is that concrete execu-

[17]It is called `ModuleExecutionDetector`. It is present and turned on in the configuration file that is generated automatically when a new S^{2}E project is created. It defines modules that are to be executed symbolically. By default, it is set to only one that matches the binary to be analyzed.

[18]Note that S^{2}E uses *lazy concretization.* If the function does not touch the symbolic variable at all, and it only, for example, forwards it somewhere else, there is no need for concretization and S^{2}E does not do that.

[19]As described in section 1.1.

[20]To be perfectly clear, let me reiterate that the previous one was $x = 4$, as described in the previous paragraph. The next one can be chosen, for example, given the path conditions of the new path, in order for S^{2}E to be able to enter it.

[21]Only the result of execution of the function with concrete parameter is considered for subsequent execution of the rest of the caller of *bar.*

tion behaves as expected, while S^{2}E simultaneously explores the function symbolically as well, as was intended [7].

Furthermore, S^{2}E has a complex but very well-working plugin system, which allows developers to easily write new plugins that can significantly alter S^{2}E capabilities. Together, we will dive deeper into the plugin system in the following section, where I talk about one of S^{2}E plugins that is crucial for this thesis, and then later in section 3.1.

2.4 Chef

Chef is the main focus of this thesis. Before I begin explaining what it is and how it works, it is important to note that this thesis is based on Chef as described in the original paper from 2014 [4]. This section is about the paper itself and the tool described there and is not necessarily completely accurate in all cases with respect to the tool as implemented in this thesis. Details about the implementation and the differences compared to the original version can be found later in section 4.1. This is mainly because Chef, as described in the original paper, is based on S^{2}E from the year 2014. The current version, which the tool developed as part of this thesis uses, is different in some parts. For example, the CUPA feature, implemented previously in Chef itself, is now part of S^{2}E. I will focus on the reasoning later in chapter 4.

To begin this section, first consider the motivation for building Chef. There are already some symbolic execution engines designed for the execution of statically compiled code. The previously mentioned Klee and S^{2}E are two good examples of those. There are many languages that compile into some form of assembly, be it actual instructions or LLVM IR. Some symbolic execution engines, such as S^{2}E or Klee, target these instructions and so are able to work with many compiled languages by default.

This is, unfortunately, not the case for interpreted languages. They are not compiled, so S^{2}E or Klee cannot be used directly. Furthermore, writing a custom symoblic execution engine would be challenging. The interpreter is generally being developed rather quickly. New language features are added constantly, existing bugs and quirks are getting fixed or changed, and present behaviour gets deprecated or removed. Furthermore, the specification, as written in the docs, often does not fully represent the language, especially with respect to implementation-defined behaviour, which cannot be ignored by symbolic execution engines as long as one wishes to model the language perfectly.

If one were to write a symbolic execution engine for an interpreted language from scratch, not only would one have to reinvent all the optimizations already done by others, but it would require sacrificing utility by supporting only one version of the language or constant modifications in order to keep it up to date with the official interpreter.

Chef, an S^{2}E plugin, works by symbolically executing the interpreter itself, with the source code of the program to test as an input. This requires some slight modifications of the interpreter and writing some reusable code on the S^{2}E side, but that is just about it [4]! The whole endeavour is much simpler and faster and reuses existing optimization techniques of S^{2}E. Another benefit is that the semantics of the language are essentially taken from the interpreter being executed, which is, by its nature, the most accurate description of the language. The Chef authors showcased this on Python and Lua – they managed to develop a symbolic execution engine for them in just three and five person-days [4].

First of all, it is important to note the difference between low-level and high-level paths. The former are paths in the interpreter,[22] as seen by S^2E. However, because of the nature of the problem Chef is trying to solve, it might be useful to have some information about the latter as well. High-level paths are paths made from high-level instructions, which correspond to individual program statements in the language that is fed to the interpreter. As an example, consider the following Python program:

```
x = [i**2 for i in range(5)]
x.append(x)
```

Each line of the program corresponds to multiple high-level instructions. For example, the list comprehension contains many high-level instructions that will be executed. Furthermore, for each high-level instruction, there are many low-level instructions that will be executed in the interpreter. If one were to write even a simple addition (such as `i += 4`), the interpreter would have to search for a variable named i, get the object that the name is bound to, and increment it, maybe even calling user code, depending on the object. All this would be implemented in C or C++, and a compiled version of the interpreter would be the low-level instructions that S^2E sees.

The low-level instructions are the level S^2E operates on but are close to meaningless to the end user. When one is interested in when a program fails and what input is needed to crash it, the user is interested in the line of code in the high-level source code and the actual variables as they input it, not bytes in the memory of the interpreter.

Because of this, Chef requires the developer to alter the interpreter of the language as well. This is because an easy way to propagate information about the high-level instructions to S^2E is simply to send a notification about each high-level instruction encountered by the interpreter.

There is another benefit to having information about high-level instructions that are currently being executed, and that is performance, as CUPA can be based on them [4].

State selection

As mentioned before, one important thing is choosing which states to execute, highlighted particularly in section 2.2. Default S^2E state choosing techniques and heuristics don't work that well in Chef's case. For example, a single high-level instruction can translate into hundreds of low-level instructions with plenty of different paths. Using, for instance, a coverage-seeking heuristic might cause a thorough exploration of the single high-level instruction. However, that is not the goal here. A better heuristic would guide the execution into exploring as many high-level instructions as possible [4].

These heuristics can be made and used precisely because the information about the high-level instruction currently being executed is passed into S^2E. Chef uses CUPA to choose these states [4]. CUPA can even have multiple

[22]In the compiled representation of the interpreter, to be precise.

layers, allowing one to use more than one heuristic. This works by executing the first heuristic first. This groups states into one or more groups. One group is selected randomly, and a second heuristic is called, which groups those states into their own groups. This process continues for each heuristic function or until a single state remains.

Interpreter optimization

Another important thing is handling some patterns in interpreters that tend to slow down their symbolic execution. While S^{2}E can handle very large projects, a few techniques can still speed up the execution by a significant margin.

One of the most significant ones, and often the easiest one to implement, is neutralizing hash functions. Interpreters often use hash-based data structures for performance reasons. However, this is not ideal for symbolic execution, particularly as these tend to generate constraints that essentially require SMT solvers to reverse the hash functions. Even if they do not, they often cause a path explosion, forking into each possible hash bucket a value could fall into [4].

Both of these are not ideal, and one simple way to fix this is to rewrite the hash functions to return a constant number, for instance. This works very well, for example, with a hash table implementation that handles collisions by creating a linked list (or a similar structure), as all properties of hash functions are still kept intact, and the only difference is that hash lookup is turned into list traversal, which is manageable to execute symbolically.

Another feature encountered by Chef authors that is used to speed up execution but often has the opposite effect for symbolic execution is when a function, for example, a string compare, contains multiple different paths based on the input. For example, a function might return `false` when it is asked to compare two strings for equality with known different lengths. However, this causes path explosion with symbolic execution. Chef authors found out that better performance can be achieved if those fast paths are removed [4].

Chapter 3

Analysis

In this chapter, I introduce the way S^{2}E and S^{2}E plugins are designed in section 3.1, and the R Interpreter in section 3.2.

3.1 S^{2}E plugin design

S^{2}E is designed with extendability in mind. It is heavily plugin-oriented, and most of its features are implemented via plugins. This gives developers of new plugins great power. Thanks to this design, the plugin interface was designed in such a way that provides a lot of flexibility to the developer, allowing modification of the behaviour of S^{2}E.

When the symbolic execution of a program starts, that is exactly one symbolic state. A state has its own memory, PC, values of CPU registers, and other information necessary to operate.

When the symbolic execution encounters a branching instruction, a second state may be created. Both states will proceed with the execution in parallel, while their PC values of registers and memory will diverge. This effect can be thought of as similar to forking a process, which might be easier to imagine and understand.

Now that the reader has a solid understanding of S^{2}E states, it is crucial to explain how plugins store data.

Each plugin is defined by a C++ class. S^{2}E spawns one instance of said class when the plugin is to be created and initialized. Note that there is only one plugin instance at all times, regardless of the number of symbolic states. All the data stored are shared, with no regard to the symbolic state.

However, there are situations where one would want to store information per symbolic state. This is solved in S^{2}E by not having just one class per plugin, there is an option to declare a plugin state class, which is instantiated per-state. S^{2}E contains macros that will retrieve the correct plugin state instance automatically, so it can be conveniently and easily used to store data per symbolic state. This way, the developer can work with per-state data easily.

Furthermore, another important feature of S^{2}E plugins is the ability to communicate with each other. Such inter-plugin messaging is used not only by various core plugins but also by Chef. A plugin may either declare dependent plugins, which are required by the plugin in order to function well, or not do that at all and communicate with plugins opportunistically when they are available. S^2E provides functionality to retrieve instances of other plugins, if available. Plugins can then work with each other in the same way as one would use any other ordinary C++ object.

However, while this might be used, this is not how plugins in S^2E work by convention and it is rare for a plugin to expose some kind of public interface. Instead, plugins in S^2E often utilize callback-based message sending.

This uses the `libsigc++` library. It provides a way for one actor to define a connection and send notifications to it. Any number of actors can subscribe to any given connection to receive all notifications emitted. In the S^2E world, this is used, for example, with the `LinuxMonitor` plugin. It defines the `onSegFault` connection, where the plugin sends a notification in case a segmentation fault occurs in the program being executed. Other plugins can listen to the events and take action when they happen.

The last thing that remains to be mentioned is the mode of communication between S^2E, respectively its plugins, and the program that is being executed. Since S^2E runs the target program inside QEMU, it has the option to implement the communication in a rather unusual way. Instead of some standard method of IPC communication, S^2E and the tested program transfer data using custom processor instructions emitted by the tested program. Those instructions are captured by QEMU and passed to S^2E, which executes some action based on it.

The communication works both ways. Usually, when S^2E is expected to act on the target program, the program sends a pointer to its memory over to S^2E using a custom instruction. For example, when the program asks for a symbolic variable to be made, it sends a pointer to the memory that is to be marked as symbolic. The system works the same way when communicating with plugins. There is an instruction called `s2e_invoke_plugin` that is used to transmit a pointer to a C-style string representing the name of the plugin, a pointer to the date to send and an additional number representing the size of the data. The data received will be passed to the corresponding plugin by S^2E, allowing communication in between plugins and the symbolically executed program.

3.2 R Interpreter

The R interpreter is a huge C codebase. To understand how it works, it is first necessary to explain core internal structures.

SEXPs

The core of the interpreter is a data structure representing R values, which was already briefly mentioned previously in section 1.2. The structure is called `SEXPREC`.[23] In the codebase, `SEXP` is often used instead. This type is a pointer to `SEXPREC`.

Since the codebase is in C, no compile-time type checking is present while manipulating the values. There is a field in `SEXPREC` representing the type of the value and plenty of macros that can be used for easy data manipulation,

[23]There is also `VECTOR_SEXPREC` used for vectors, although one does not come across it most of the time

but it is the responsibility of the developer not to confuse the types. The types rather closely mimic R types the user is familiar with – there are integer vectors, numeric vectors, environments and many others. There are, however, a few types that one does not come across often in regular R code but are important to the thesis, so they deserve a mention here.

`LISTSXP` is a pairlist, a list of values. It is often used for function arguments.

`SYMSXP` is a symbol – a name referring to a variable.

`LANGSXP` is a language object, typically calls. To be precise, they are pairlists, where the first element refers to a function to be called, and the rest are function arguments.

`EXPRSXP` are expressions. It is structured as a vector of `LANGSXP`.[24]

There are more types, but they should be either well-known to an R developer, such as vectors and lists [17],[25] or are not interesting to our case and occur only marginally in the parts of the codebase that will be modified later for the sake of implementing Chef.

The `SEXPREC`s are connected in the form of a tree. Each `SEXPREC` contains three pointers regardless of its type – a pointer to attributes, previous and next node.

The next and previous nodes are most often used by the garbage collector. For regular purposes, such as traversing a list, another structure in the `SEXPREC` is often used. It contains[26] three pointers that are used to represent compound structures. The meaning is slightly different for each type, but for example, for `LISTSXP`, the first two pointers are `carvalue` and `cdrvalue`,[27] allowing list traversal [17].

Many types come with convenience macros that allow the developer to easily work with this structure. For example, for lists, there are macros `CAR0`, `CDR`, `CADR` and many others, which can be used to access various elements of the list.

Another mention-worthy feature is attributes. Attributes can be set on many objects, modifying their behaviour. For example, a matrix is a vector with a `dim` attribute describing the number of dimensions of the matrix.

As mentioned earlier, every `SEXPREC` contains a pointer to attributes. It is either a `R_NilValue` (meaning that the object has no attributes) or a pairlist. Pairlist is special. Inside its data structure, it contains not only `carvalue` and `cdrvalue` (as mentioned earlier) but also a `tagval`, which is again a pointer to a `SEXPREC`. This is used to represent attributes. They are stored in a pairlist, where each element of the pairlist contains a TAG (`tagval`), which stores the name of the attribute. This is shown more intuitively by figure 3.1.

[24]Almost always `LANGSXP`, although it is not guaranteed to be so.

[25]They were described in section 1.2.

[26]Usually. The data is a union of multiple structures, although all but one contain three pointers to another `SEXPREC`.

[27]This is a convention that should not be too surprising to anyone that has worked with a LISP-style language. `CAR` is an operation that returns the first element of a list, while `CDR` returns the tail of a list. So, in order to retrieve the second element of a list, one would call `CAR(CDR(list))`, or a shorthand, `CADR`. This is explained in great detail in the BI-PPA course at FIT, CTU, which I can strongly recommend to any reader without knowledge of LISP.

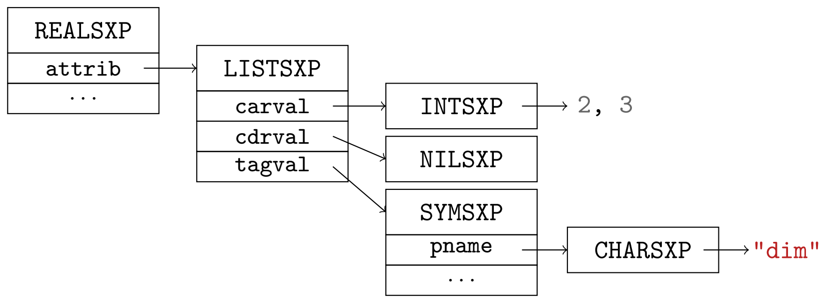

Figure 3.1: Attributes section of a `matrix(c(1,2,3,4,5,6),nrow=2)`

Environments

Variables in R are symbols. When a user attempts to access a value of a variable, the interpreter has to retrieve a value bound to the name of the corresponding symbol.

The place where variables are looked up is called *environment* in R. It is a scope in which variables can be found, and those scopes are linked to previous ones. Environments have their own type, `ENVSXP`. It contains a pointer to its parent and a hashtable with the symbol-value pairs. The hashtable is implemented as a `VECSXP` filled with either `NILSXP` or `LISTSXP`. A `LISTSXP` contains the actual values, with each element of the pairlist having a tag containing the variable name itself.

As environments contain a pointer to their parent, values from parent scopes can be accessed in R. There is even an operator called *super assignment* that allows one to modify an existing variable in a parent environment.

```r
dice_roll <- 4

innocent_function <- function() {
    # Notice the super assignment here
    dice_roll <<- 42
}

innocent_function()

print(dice_roll)  # 42
```

Built-ins: `names`

The R contains a number of built-in functions. Some of the ones that the reader is surely familiar with are, for example, `+` or `<-`.

They are recorded in the file `names.c`. There is an array `R_FunTab` that contains the name of the function, a pointer to the function implementation and some flags, such as the arity of the function, associativity or whether the arguments of the function should be evaluated before the function is called.

When a function is called in the eval loop, this array is searched for a matching function to call.

Garbage collection

R uses a garbage collector, so the user does not need to manage memory. Every once in a while,[28] the garbage collector marks all objects that are currently reachable and removes everything else. This means that when writing C code interfacing with the interpreter (or modifying the interpreter itself), extra caution has to be given to handling R objects, such as `SEXP`s.

If one creates an R object in C, there is currently no reference point to the object from anything! So, if the garbage collector is triggered, the object might be collected immediately. In order to suppress garbage collection of certain objects until they can be returned to the user or attached somewhere, the R interpreter offers the `PROTECT` and `UNPROTECT` functions.

The `PROTECT` function takes a `SEXP` and places it onto a `PPStack`.[29] When the garbage collector runs, it checks the stack and does not delete any `SEXP` found, neither any `SEXP`s referenced by any protected one, recursively.

This means that while it is wise to use `PROTECT` for any `SEXP` created, it has to be removed from the `PPStack` later or else a memory leak would occur. Because of this, R has a convention where a caller is responsible for protecting arguments to functions (so any arguments a function receives are always protected) and that the `PROTECT`s and `UNPROTECT`s have to be balanced – so every `SEXP` that has been protected has to be unprotected before the function returns. The caller can `PROTECT` the returned `SEXP`s if they desire to, but it is their responsibility to do so [18, Section 5.9.1].[30]

[28]This can generally happen whenever the R interpreter allocates. As there is no guarantee whether a given interpreter function/macro allocates or not, one should assume that the use of any function or macro can trigger the garbage collector.

[29]The pointer protection stack.

[30]There are tools that check whether `PROTECT`/`UNPROTECT` as used correctly, such as `rchk` by Tomáš Kalibera.

Chapter 4

Design and Implementation

In this chapter, I will elaborate on the approach to making Chef. I will highlight the requirements on the S^{2}E plugin and the way it is implemented. After that, I will focus on the R interpreter modifications, explain why they are needed, and what requirements they fulfil. Finally, the set of tools used to analyse the resulting output will be discussed.

However first and foremost, I would like to explain one obstacle I spent quite a lot of time working around. It heavily changed the course of development and affected the design of Chef.

The paper describing Chef is ten years old at the time of writing; thus, the tool they developed is of considerable age as well. This caused numerous problems that I spent weeks trying to resolve. From old S^{2}E that had build script, which downloaded dependencies from code forges that no longer exist[31] to issues building QEMU at all and having to manually edit the source code to make it compile. Another problems I ran into was unavailability of images that S^{2}E used to run and having to search for old Debian images on the internet archive, installing packages that are several years old[32] and the absence of libraries with modern TLS. This proved to be hard to solve and was very time-consuming.

After this experience, I considered the pros and cons of this approach. Ultimately, I decided not to use Chef as it was provided, as further development would be very slow, documentation scarce, and there was no longer any community that would be able to help neither me nor the user with any issues, should they arise. Furthermore, any potential further development after this thesis would be rather unlikely, and any development of new features would be pretty hard.

I decided to alter Chef and port to modern S^{2}E, running on a modern system and using up-to-date dependencies. This allowed me to create a much more pleasant experience for both the user and any potential further contributors to this tool. It also enabled Chef to take advantage of new features in S^{2}E, for instance, CUPA, that was not implemented in early releases of S^{2}E that the

[31]Fortnuately, many of links to the source forges contained the name of the project together with the hash of a specific commit, which allowed me to recover the exact version from a different source forge.

[32]One interesting problem, in particular, was expired PGP keys, so installation was not as simple as simply changing the mirror to Debian package archive.

original Chef used, and thus Chef authors had to implement it entirely from scratch.

The new version of Chef consists of mostly copy-pasted code from the original Chef, altered to work well with newer versions of C++ and modern S^{2}E. However, thanks to the new capabilities of the ecosystem around Chef, I managed to remove large chunks of the original code since it was not needed for this purpose.[33] The resulting code is smaller, and much of the complexity has been offloaded to S^{2}E. It now works in a modern ecosystem, allowing one to use familiar, up-to-date tools and benefit from the community around them. All things considered, I believe it was a good call to alter Chef in this way, and it now should be much more pleasant to both use and develop.

4.1 Chef

The S^{2}E plugin, Chef, communicates with the R interpreter using custom CPU instructions, as I mentioned in section 3.1. It works by sending over a pointer referencing the following structure:

```
enum S2E_CHEF_COMMANDS {
    START_CHEF,  // start chef session
    END_CHEF, // end chef session
    TRACE_UPDATE, // new high-level instruction was executed
    ABORT_STATE // this state should not be considered ->
                // kill state without generating a testcase
};

struct S2E_CHEF_COMMAND {
    S2E_CHEF_COMMANDS Command;
    union {
        struct {
        } start_chef;
        struct {
            uint8_t error_happened;
        } end_chef;
        struct {
            uint32_t op_code;
            uint32_t pc;
        } trace;
    } data;
};
```

Chef is listening to these commands.

`START_CHEF` and `END_CHEF` start or stop an analysis. This means whether Chef is listening to `TRACE_UPDATE` and managing a record of the instructions.

`TRACE_UPDATE` causes Chef to record the new high-level instruction that was just executed. As of now, it is used for some nice-to-have information in test cases and, more importantly, CUPA.

[33]For example, I removed the support Chef had for CUPA in favour of S^{2}E-powered CUPA with HLPC/HLOP support added.

ABORT_STATE is a signal that causes Chef to terminate a state without generating a test case. Under normal circumstances, a state that has terminated will produce a test case documenting the input needed to reach that state of execution. However, it is not needed at all times.

A good example might be a user who wants to generate a positive symbolic integer. Chef does not have a special command to tackle this kind of request. Instead, the user might create a symbolic integer and restrict it to just positive values by calling this ABORT_STATE if the value does not conform to the user's request.

However, one may question the whole existence of this command. After all, S^2E has a similar command built in, called s2e.assume. One can just call it from the guest code and thus enforce that only the valid values will be considered. The issue with that is that it is an error to use s2e.assume with a condition that cannot be satisfied, which can happen. This ABORT_STATE solution does not have such a restriction. Furthermore, it is guaranteed to never generate a test case in case the condition does not hold, which is not true for s2e.assume, which kept generating new test cases since the state ended and Chef reacted by generating a test case.

A test case is generated by iterating through all symbolic variables and getting some concrete value satisfying all conditions encountered. This test case generation is run when a state exits. This can happen, for example, via an END_CHEF command, which, as presented above, does provide an error_happened parameter.

However, one might also be interested in test cases that cause the program to segfault. In order to handle this, Chef will depend on a plugin LinuxMonitor. It is not necessary for Chef to use it. It will function without it as well. But if the plugin is enabled, Chef will hook on its onSegFault connection. When the LinuxMonitor sends a notification that a segmentation fault happened, Chef will output a test case, assuming the analysis is turned on.

Finally, the last feature that Chef provides is CUPA optimization. I already talked about CUPA in previous chapters. It essentially prioritizes states based on some heuristic. In this case, the target criterion for prioritizing state will be PC and opcode. The way this is used in the R Interpreter means that S^2E will try to make each high-level instruction take about an equal amount of processing time. So if there is a single high-level instruction that causes a path explosion, S^2E will keep exploring other instructions as well, about as often as the problematic one. It will not hoard most of the processing time for itself.

In Chef, this is done by extending the S^2E CUPASearcher plugin. It is implemented by extending the CUPASearcherClass. The target is to have two criteria – high-level PC and high-level opcode. So, two new classes are created: CUPASearcherHlpcClass and CUPASearcherHlopClass. These classes implement an interface that S^2E CUPASearcher plugin requires. An instance of each of the classes will be added to the CUPASearcher, which will then take care of calling the classes and actually scheduling the states.

The classes need to receive data from Chef in order to function properly. Namely, they require to know the last execution that was executed for any given state. This is implemented by utilizing connections. Chef exposes an `on_interpreter_trace` connection. Recipients registered will receive a high-level instruction each time Chef receives one, which is every time the interpreter sends a `TRACE_UPDATE` command. The CUPA implementations will store it in a mapping from state ID to the latest high-level instruction, so they can answer queries from the `CUPASearcher` plugin.

Memory Usage

During testing, it became apparent that memory usage might be a major problem with a larger analysis. The problem is the fact that as time goes on, new states are forked constantly, typically at a much higher pace than they are finished. This causes a constant increase in the number of states and, thus, higher memory usage. This can be visualised by an experiment that was left to run overnight and generated a high number of states.[34] The correlation between a number of states and memory consumption can be seen in figure 4.1.

The solution to this is `MemoryForkLimiter`. An additional plugin that tracks memory usage of both the process and the overall system memory usage and disables forking and spawning of new states should some configured limit be exceeded.

It is turned off by default because of its impact on the analysis and the speed of completing it. Still, it can be enabled if the user is constrained by memory usage or attempts to execute an experiment that causes a great number of state forks. The effect of enabling this can be seen in figure 4.2. It can be seen that memory usage started climbing much slower once was the limit on state forking enforced. It still rose, due to the fact that many of the states were just created, but not explored in any way, so it is not a way to limit memory usage up to a limit. Rather it is a notification to Chef that it should try to limit memory consumption as much as possible.

4.2 R Interpreter

A key part of getting Chef to work is altering the interpreter in order for it to invoke S^{2}E. There are a few necessary features.

First of all, it is sending information about executed high-level instruction. This is needed for Chef to correctly guide execution via CUPA. This is achieved by creating a structure in memory and sending a pointer to that structure over to S^{2}E, as explained earlier. Since S^{2}E has the same structure defined as well, it suffices to read the pointer sent over. It will be able to read the structure and extract the data.

The solution to that is rather simple: it suffices to locate the eval loop[35] and insert a single call that will send the high-level instruction notification over to S^{2}E, as seen in figure 4.3.

Second, the interpreter has to send a few basic commands over to Chef. For instance, the user might not want Chef to process everything but is interested only in some parts of the program. Sending regular updates over to S^{2}E might be costly performance-wise, especially if there is more processing of high-level instructions implemented on the S^{2}E side. For this purpose, the R interpreter

[34] Intel Xeon Gold 6140 (72) @ 2.294GHz, 188GiB of RAM.

[35] Conveniently located in a file called `eval.c`, in a function called `eval`.

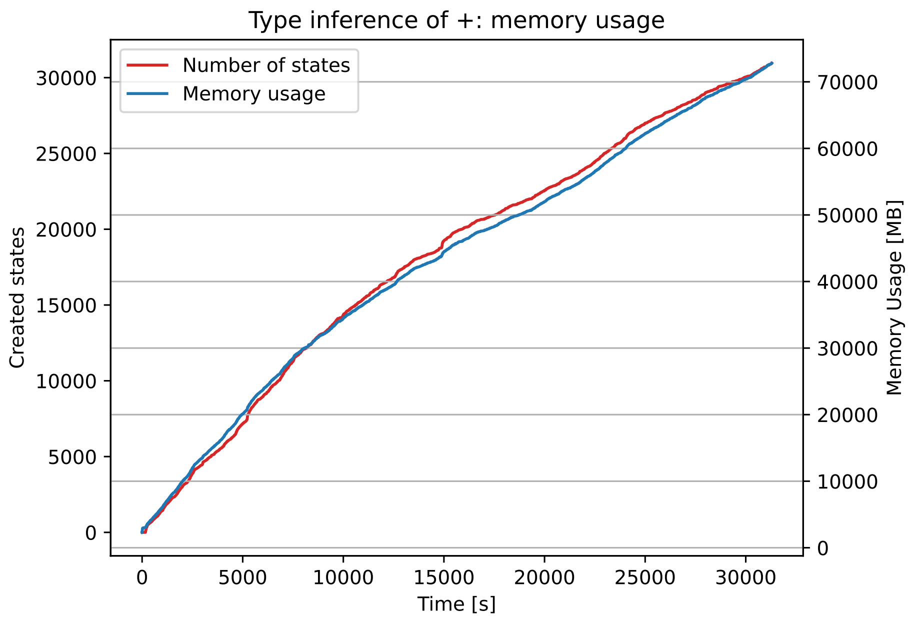

Figure 4.1: Memory usage and states count of a long-running analysis

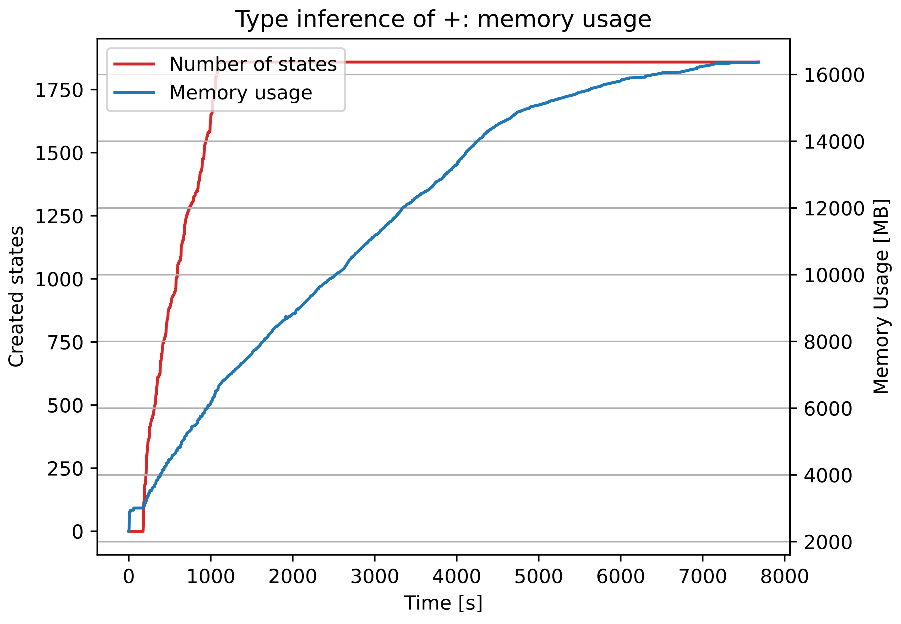

Figure 4.2: Memory usage with restricted state forking

```
SEXP eval(SEXP e, SEXP rho)
{
// If envvar R_SYMBEX is set to "1"
if (R_SymbexEnabled()) {
    // Send update to Chef about currently executing instruction.
    // Note that multiple calls per one line of code
    // may occur. While this does not affect Chef at all,
    // it might be surprising if one does not anticipate it.
    R_UpdateHighLevelInstruction(e->sxpinfo.type, (u_int32_t)(size_t)e);
}

// (...)

switch (TYPEOF(e)) {
    case NILSXP:
    case LISTSXP:
    case LGLSXP:
    // (...)
}
}
```

Figure 4.3: Modification of the R eval loop that sends high-level instruction updates to Chef.

has to be able to not only send a high-level instruction update but also a notification to start or end an analysis or simply terminate a state, as was described in the previous section.

And finally, it has to offer some way for the user to control the communication with S^2E in a program. For example, the user should be able to create new symbolic variables and make assertions.

Currently, the modified interpreter provides the following functions:

`chef.start()` starts Chef tracing.

`chef.end(LOGICAL)` ends Chef tracing. If the first parameter is set to `T`, it means that an error has occurred, and a test case will be dumped into an error test cases file. It will go to the successful test cases file if it is set to `F`.

`chef.debug(STRING)` sends a debug log over to S^2E. The message will appear in `debug.txt`. External tools may utilize this to receive additional information about the execution.

`chef.int(STRING)`, `chef.numeric(STRING)` create a symbolic variable with a given name.

`chef.raw(STRING, INTEGER)`, `chef.string(STRING, INTEGER)`, `chef.vec(STRING, INTEGER)`, `chef.any(STRING, INTEGER)` all create a symbolic variable with the given name and a length. Note that `chef.vec` creates a symbolic type of a vector. Not all are supported, nested vectors for example, but many of them are. The `chef.any` creates either a `NULL` or any of the other symbolic variables mentioned.

`chef.list(STRING, INTEGER)` and `chef.matrix(STRING, INTEGER, INTEGER)` create either a list of given length or a matrix with given dimensions. The list is filled with symbolic data of variosu type, while the matrix contains either integers or floating point numbers.

`chef.attach(EXPR)` attaches a symbolic attribute with a symbolic value to any given `SEXP`. This may be tricky to use, but could be useful for trying to crash badly-written libraries that do not verify attributes properly.

`chef.asserts(LOGICAL)` checks whether a condition is truthful and generates an erroneous test case if not. Note that there are other cases when an erroneous test case is generated, namely when an uncaught error is thrown or when a segmentation fault occurs. The first one is done by modifying the error handling in the R interpreter, and the second one in the S^{2}E plugin itself.

`chef.assume(LOGICAL)` terminates all states that do not fulfil a certain condition without generating a new test case. Since the state is deemed invalid or out of scope by the user, it is not expected for Chef to generate a test case describing how to reach it.

The functions are defined as builtins in `names.c`, as explained in section 3.2.

To showcase their usage, they could be used to mimic the example in section 1.1 like so:

```
chef.start() # Start Chef
i <- chef.int()
j <- chef.int()

i <- i + 4
i <- i * 2
i <- i + j

if (i < 0) {
    chef.debug("Branch 1")
} else {
    chef.debug("Branch 2")
}
chef.end(F) # No error happened
```

The symbolic execution engine manages to infer that in order to reach the first branch, one can for example set `i <- -64; j <- 0`, and the second branch can be reached with `i <- 0; j <- 65793`.

Symbolic variables

Internally, the functions that generate symbolic variables build an R `SEXP` and send a pointer to data of that structure over to S^{2}E to mark the buffer as symbolic. The useful `s2e.h` library is used for that. It contains functions that send opcodes to S^{2}E that support many actions, marking buffers as symbolic among them.

```
// s2e.h

#define S2E_INSTRUCTION_REGISTERS_COMPLEX(val1, val2)    \
        "pushl %%ebx\n"                                  \
        "movl %%edx, %%ebx\n"                            \
        S2E_INSTRUCTION_COMPLEX(val1, val2)              \
        "popl %%ebx\n"

#define S2E_INSTRUCTION_REGISTERS_SIMPLE(val)            \
    S2E_INSTRUCTION_REGISTERS_COMPLEX(val, 00)

// (...)

/** Fill buffer with unconstrained symbolic values. */
static inline void s2e_make_symbolic(void *buf, int size,
                                      const char *name)
{
    __s2e_touch_string(name);
    __s2e_touch_buffer((char*)buf, size);
    __asm__ __volatile__(
        S2E_INSTRUCTION_REGISTERS_SIMPLE(03)
        : : "a" (buf), "d" (size), "c" (name) : "memory"
    );
}
```

Strings are, however, quite difficult to tackle. One cannot simply use the `mkChar` family of functions that R provides, as R caches strings. This is not desirable in this case, so one has to implement it in a way that avoids caching.

S^{2}E also expects a name of symbolic variables – this is later used, for example, in generated test cases. This is the reason why the functions that provide symbolic variables take the name of the variable as a parameter. This is later passed over to S^{2}E. However, there is one more thing to consider.

It is often useful to know the types of variables. This powers tools that are able to generate tests out of test cases – without types, it would be hard to know how to parse bytes into some relevant value and output it as a literal. Furthermore, tools that provide some form of type annotation of functions based on symbolic execution could be made, which is later presented in the following section.

In order to gather this information, one needs to pass the type of every symbolic variable from the interpreter to S^{2}E, which will later be included in the generated test cases. One could send them via a special command and store them in some mapping. However, this would require nontrivial support in multiple places.

A more straightforward solution, and one that I chose to go with, is to prepend the names of the symbolic variables with the type. So, for example, a symbolic integer called `foo` would become `int__foo`, and similarly, the name of a string could look like `str__bar`. This has the advantage of being easy to implement in both the R interpreter and the tools and simultaneously being completely transparent to S^2E. It allows the addition of new types easily and is generally quite simple to work with.

Performance

In order to keep symbolic performance manageable, some changes had to be made to the interpreter, mostly relatively small and easy to make. Taking inspiration from the Chef authors, I removed any hashing present in the interpreter.

I did so by simply modifying the hashing functions to always return a constant, zero, in my case. This removes the danger that the SMT solver would be asked to solve constraints that would be essentially equivalent to reversing a hashing function, a task that can be assumed to have a rather concerning runtime cost. This works due to the fact that R hash-based data structures are implemented as a linked list, so degenerating the hashing function does not break the algorithm.

To showcase this change, a rather good example is environments. They contain a hashmap of symbols and their values and can be easily visualized by using the internal function `inspect`.

```r
library(rlang)

.Internal(inspect(env(w=1,x=2)))
```

In stock R, it might print something like this:

```
@11eb52078 04 ENVSXP g0c0 [] <0x11eb52078>
ENCLOS:
  @134849f88 04 ENVSXP g0c0 [MARK,REF(65535),GL,gp=0x8000] <R_GlobalEnv>
HASHTAB:
  @13400ab30 19 VECSXP g0c7 [REF(1)] (len=29, tl=2)
    @13480dae0 00 NILSXP g0c0 [MARK,REF(65535)]
    @13480dae0 00 NILSXP g0c0 [MARK,REF(65535)]
    @13480dae0 00 NILSXP g0c0 [MARK,REF(65535)]
    @11eb55c40 02 LISTSXP g0c0 [REF(1)]
      TAG: @124010818 01 SYMSXP g0c0 [MARK,REF(625)] "w"
      @12413b0c0 14 REALSXP g0c1 [REF(65535)] (len=1, tl=0) 1
    @11eb55c08 02 LISTSXP g0c0 [REF(1)]
      TAG: @123809ce0 01 SYMSXP g0c0 [MARK,REF(58872)] "x"
      @12413b088 14 REALSXP g0c1 [REF(65535)] (len=1, tl=0) 2
    ...
```

While in the modified R, the output was the following:

```
@563ea8dae7f8 04 ENVSXP g0c0 [] <0x563ea8dae7f8>
ENCLOS:
  @563ea635e8c0 04 ENVSXP g0c0 [MARK,REF(65535),GL,gp=0x8000] <R_GlobalEnv>
HASHTAB:
  @563ea6672170 19 VECSXP g0c7 [REF(1)] (len=29, tl=2)
    @563ea8dae478 02 LISTSXP g0c0 [REF(1)]
      TAG: @563ea638c188 01 SYMSXP g0c0 [MARK,REF(14186)] "x"
      @563ea92b6ae0 14 REALSXP g0c1 [REF(65535)] (len=1, tl=0) 2
      TAG: @563ea643f340 01 SYMSXP g0c0 [MARK,REF(411)] "w"
      @563ea92b6b18 14 REALSXP g0c1 [REF(65535)] (len=1, tl=0) 1
    @563ea6325e50 00 NILSXP g0c0 [MARK,REF(65535)]
    @563ea6325e50 00 NILSXP g0c0 [MARK,REF(65535)]
    @563ea6325e50 00 NILSXP g0c0 [MARK,REF(65535)]
    @563ea6325e50 00 NILSXP g0c0 [MARK,REF(65535)]
    ...
```

Notice that both variables have been hashed into a different bucket in the original version of R; however, in the modified version, all ended up in the same bucket.

While this obviously degrades performance in the case R is used in the usual way, it can speed up SMT queries considerably [6].

Another thing that was done is disabling implicit printing of variables. This caused complex SMT queries that took hours to resolve. Other optimizations may be possible to discover and implement as well in the future.

User experience

While this setup works quite well, there was one major flaw in the realm of user experience. The user had to specify the name of the variable when creating a symbolic variable. However, this is completely unnecessary and is forcing the user to do a meaningless action that might cause bugs and confusion should the user provide an incorrect name.

There are cases when the name is rather easy for the interpreter to retrieve. In a lot of cases, the user may write something like this:

```r
firstNum <- chef.int("firstNum")
someRaw <- chef.raw("someRaw", 10)
```

In this specific case, the interpreter can infer the name. The goal is to provide such implementation so that the user can simply write the following code with the same effect:

```r
firstNum <- chef.int()
secondNum <- chef.raw(10)
```

The question remains: how can that be done? The code that automatically detects the name should not be in the chef function itself since it does not know the AST, and while it could be retrieved, the resulting code that retrieves it would be quite ugly and error-prone.

An approach that worked better for me is extracting the variable in the place where the AST is straightforward to retrieve: the `eval` function.

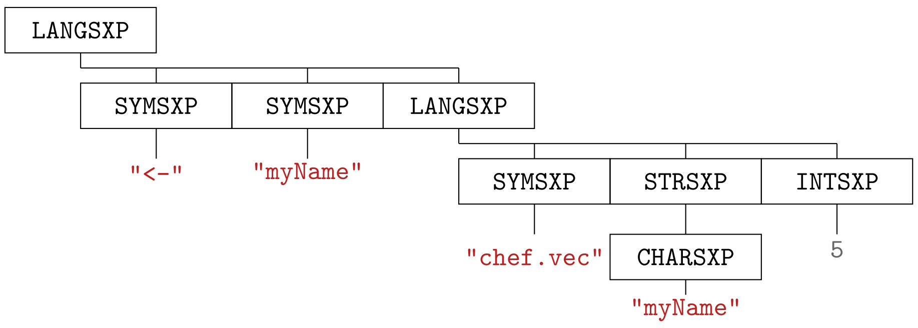

Figure 4.4: AST diagram of `myName <- chef.vec("myName", 5L)`

It can be seen that when a code follows this structure, it is not hard to extract the variable name from the AST, allowing the user to omit the first argument of `chef.vec`.

The solution is to monitor the usage of the `<-` function in the eval loop. Once it is detected, it suffices to check for the left and right sides of the function (to be more precise, its first and second arguments). Visualization of this is provided as figure 4.4.

If the left side consists of a single `SYMBOL`, it is a variable that the user is trying to assign to. It might be something else if the user did, for example, `vector[[5]] <- chef.int()`; however, those examples are not trivial, and name generation might be obscure. For these reasons, I decided to only proceed with the simple case, where the user is assigning to a variable.

Next, if the right side consists of a call to a chef function, and that function is missing exactly one argument, it is assumed that the function call is missing the name of the variable. In this case, the name of the symbol on the left side of the `<-` is extracted, and the AST is modified – the name is inserted as the first argument of the chef function being called.

This ensures that the chef functions receive all the arguments the usual way, even if the user does not specify the name, in case the interpreter can easily infer it. Ultimately, I find this to have significantly increased the user comfort while writing scripts utilizing Chef.

4.3 Chef Tools

`Chef tools` is a collection of scripts that aim to aiding the user in processing the output returned by Chef. A tool that will process the output files has been deemed useful, despite efforts to make the output as clear and readable as possible, because the files can be quite large.

The package is designed to be easily extendable – the user should have no problems adding their own tools for their own use cases. This is important since the Chef itself is a symbolic execution runner; it can be utilized for a number of different actions, so the tooling should be as easy to make as possible.

This is one of the main reasons for writing the tools in Python. It is a language I am quite familiar with, easy and quick to write, pretty popular, and most likely already available on your device. Furthermore, the performance concerns are not too relevant here since the files that Chef produces tend to be just a few tens of megabytes at most in the scenarios I tested. Python has no issues with processing files of this size.

To support this use-case,[36] this tool is split into two separate packages.

- `chef-testcase-parser`, a Python library that can parse test cases generated by Chef.
- `chef-tools` is a Python command-line utility containing tools that process the result. As of the time of writing, there are currently two tools available.
    1. `testgen` that can turn an R script used for testing and a list of (erroneous) test cases into a `testthat`-compatible file that can be run outside Chef. This is designed to be easily integrated into existing test suites.
    2. `typeinfer` that analyzes which types can a tested function can process. It also produces mapping in the form of inputtype* $\rightarrow$ outputtype, as many R functions produce outputs with different types in response to different inputs. An example of this is presented later in section 5.2.

Testcase parser

I made the decission to separate the logic that parses test cases[37] from the tools. It contains some non-trivial logic and is likely to be updated in the future, while the tools themselves are quite simple and will most likely receive little to no updates once they are made. It is, therefore, beneficial for this library to be updated separately.

Furthermore, not all users who will make their own tools will be willing to participate in the development of chef tools, and there is no need to force them to modify the code I have written. With the existence of this separate library, they will be able to write their own code much more easily.

The problem itself is not something extraordinary. There is a `json` file as an input, containing an array of test cases. Each test case, among some metadata, contains variable assignments. The library cannot simply parse the JSON and return the result for a simple reason – the assignments correspond to low-level C calls to S^2E, not to the R assignments that the user is interested in.

To make this clear, the generation of some variables has $1:1$ correspondence between C and R assignments. For example, if one created an integer symbolic variable in R, it would generate one symbolic variable visible to S^2E. However, some symbolic variables in R – such as vectors – generate more symbolic variables. The vector, for example, generates one symbolic variable that defines the type of the vector and a second one that represents the data. An example can be seen in figure 4.5.

The resulting design is relatively simple. For each possible R assignment, which might be, for example, an integer, `NULL`, or a vector, there will be a

[36]Particularly the extendability requirement.

[37]These are the files called `successful_test_cases.json` and `err_test_cases.json`.

```
{
"timestamp" : 4759,
"pc" : 4110240480,
"stateId" : 14,
"inputs": [
   { "name" : "v0_any__lhs_0",
        "bytes" : "0x3 0x0 0x0 0x0",
        "i32" : 3,
        "string" : "\\x3\\x0\\x0\\x0"}
   ,{ "name" : "v1_vec_type__lhs_1",
        "bytes" : "0xe 0x0 0x0 0x0",
        "i32" : 14,
        "string" : "\\xe\\x0\\x0\\x0"}
   ,{ "name" : "v2_vec_data__lhs_5",
        "bytes" : "0x0 0x0 0x0 0x0",
        "string" : "\\x0\\x0\\x0\\x0"}
   ]
}
```

Figure 4.5: Raw Chef test case consisting of three symbolic variables that map to a single R value.

Python class that can parse a given C assignment (or assignments) into this R assignment. It will consume the proper amount of C assignments and convert the data into a proper format.

During the parsing, the library will go through all assignments in any given test case and, for each of them, will find a class that can parse the assignment[38] and feed it the required number of them.

Chef-tools

`chef-tools` is a simple collection of Python tools, utilizing the `click`[39] tool for the command-line interface. The choice of `click` itself is due to its popularity and ease of use. It fulfilled all the requirements: automated generation of help texts and input validation, which `click` handles quite well.

Other than that, it is a rather standard Python command-line utility, currently containing two tools that perform data transformation utilizing the aforementioned test case parser.

The key was keeping the source code of chef tools as simple as possible so as not to hinder potential users from writing their own tools in tandem with the existing ones. The only thing the user is required to add a new tool is to add a new command, which, thanks to `click`, is a matter of defining a single function and creating a new module that does the data transformation required.

[38]Each assignment contains a name of the variable. In the R script, the user specifies the name of the variable. However, as the R interpreter prefixes it with its type before sending it to S^{2}E, the type of any given variable is known, and a correct class to parse it can be selected.

[39]`https://github.com/pallets/click`

Test Generator

The test generator is a utility that can produce `testthat`-compatible file that runs tests generated. It works by parsing the source code used to run the Chef experiments. It can not handle every possible way to run Chef, but as long as the user conforms to the usual way of invoking Chef, the test case generator should cover them without the need for modification. For example, let us consider the following simple example:

```r
chef.start()

a <- chef.int()
chef.assume(a > 10)

b <- chef.int()
chef.assume(b > 10)

chef.assert(a + b > 22)
chef.end(F)
```

This obviously fails for a certain combination of `a` and `b`. Running Chef generates test cases; some of them will be successful, and some of them will not. After the run, the user can feed the data to the test generator script.

First of all, it takes the source code, cleans Chef-specific functions and either removes them or replaces them with native or `testthat` equivalents. After that, it wraps the rest of the code in a function accepting the arguments that were previously symbolic variables.

Afterwards, the test cases provided by the user and generated by Chef are parsed, and for each test case, the function containing the user source code will be called.

In this example, it provides something similar to this:[40]

```r
# chef-tools testgen err_test_cases.json source.R
library(testthat)

chef_fun_to_test <- function(a, b) {
    expect_true(a + b > 22)
    expect_false(F)
}

context("Tests generated by Chef")

test_that("Test case 0", {
    chef_fun_to_test(2146435080L, 1048704L)
})
test_that("Test case 1", {
    chef_fun_to_test(11L, 11L)
})
```

[40]It took 899 seconds and 3.2 GB of RAM to run this on single core, Intel i5-8250U.

As one can see by running this program, the symbolic execution run actually found two errors: one that I deliberately put in; summing 11 + 11 does not result in something higher than 22. However, it also found a second one that I did not anticipate: summing 2146435080 and 1048704 results in `NA` due to integer overflow.[41]

Type inference

When the `chef.any` function is called in R, it generates a symbolic variable with any type – it can be a string, a number, or perhaps a symbol. Whenever a test case is recorded, it provides not only the value of the variable but the type of it as well.

The type inference tool takes advantage of this to retrieve the set of types that the function that is being tested managed to process without an error. For example, consider the following:

```
chef.start()
tryCatch({
  var <- chef.any(5)
  functionThatIsBeingTested(var)
}, error = function(e) {
  chef.end(T)
})
chef.end(F)
```

The function is deemed to accept any type that was successfully used to invoke the function at least once. So, it would suffice for the tool to look at successful test cases and extract a list of types of the variables.

```
$ chef-tools typeinfer successful_test_cases.json
integer
nil
logical
```

However, one might also want to get the whole mapping from input to output types. The type inference tool was developed with this in mind. In order to use it, the user has to specify the return type of the function in the log of the execution.

```
chef.start()
tryCatch({
  arg1 <- chef.any(5)
  arg2 <- chef.any(5)
  result <- functionThatIsBeingTested(arg1, arg2)
  # The typeinfer tool will know that the type of the result produced
  # was caused by lhs and rhs variables
  chef.debug(paste("TYPEINFER", typeof(result), "arg1", "arg2"))
}, error = function(e) {
```

[41] R integers are 32 bits long, however, the lowest value (−2147483648) is used to represent `NA` instead, a value that is created as a result of integer oveflow/underflow, for example.

```
  chef.end(TRUE)
})
chef.end(FALSE)
```

The tool will search for the execution log for names of the arguments and a return type. Each log line contains state ID, which is also included in the test case metadata. The tool will then pair test cases with specific state IDs to the corresponding entry in the log and produce a mapping like so:

```
$ chef-tools typeinfer successful_test_cases.json debug.txt
integer    integer    -> integer
nil        logical    -> logical
```

Furthermore, it should be noted that in R, even scalars are vectors.

```
> typeof(1L)
[1] "integer"
> typeof(c(1L))
[1] "integer"
> identical(1L, c(1L))
[1] TRUE
```

So even when the user creates a symbolic vector and passes it as an input to the tested function, or when the function returns a vector of multiple scalars, the type inference tool will report it as an `integer` (or another type). It does not explicitly distinguish between vectors of length one (scalars) and vectors that are longer.

If the user encountered a use case where the length of a vector is vital knowledge, for example, if one wanted to explore how functions behave when given inputs of a specific size, they should be assured that S^{2}E and Chef already capture this information, and the test case parsing library already does handle that. It should be relatively easy to patch the type inference tool to print the length of the vectors alongside their type.

Chapter 5

Evaluation

The symbolic execution engine is implemented and ready to use. However, there are two questions remaining: how can it be used, and how fast is it? I attempt to answer these questions in this chapter. I showcase a few applications where a tool like this could be used and highlight some performance characteristics. If the reader wants to follow along and try out the experiments themselves, the appendix A contains instructions on running Chef.

5.1 Testing library correctness

One of the most natural uses of symbolic execution, in the area of verifying library correctness, closely mimics property-based testing, as already explained previously in section 2.1.

If one wanted to evaluate the correctness of a library, a good test would not only try to get the implementation to crash but would also verify the results. How to do that when inputs are generated during the runtime?

One way basically mimics property-based testing. One can verify the correctness of results in some cases. For example, it is not hard to verify whether a tree is balanced. An AVL tree balancing algorithm could be tested by being fed a random (or symbolic) tree and just verifying whether it is balanced afterwards. In this simple case, I verify whether the `base64encode` and `base64decode` functions output sensible results by trying to encode and later decode a symbolic buffer and verifying, whether the result was identical to the input.

```r
library(base64enc)
chef.start()
input <- chef.raw(100) # limit input length

# Encode and decode it
encoded <- base64encode(input)
decoded <- base64decode(encoded)
# Compare whether the result is identical to the input
chef.assert(identical(input, decoded))
# If we got here, there was no segfault, no error and the assert passed
chef.end(F)
```

The test did not find any output for which the condition would not hold. After terminating, S^{2}E reported peak memory consumption of about 4 GB and took about four and half hours to run.

This example, while useful, it may not detect bugs in the implementation of `base64enc` itself. The encoded buffer could be just plain wrong, and while it might decode correctly, it still does not prove corectness.

One solution is comparing the result of `base64enc` to another library or to a trusted reference implementation. In this case, I chose `jsonlite`.

```r
library(base64enc)
library(jsonlite)

for (strlen in 1:100) {
    chef.start()

    str <- chef.string(strlen)
    raw <- charToRaw(str)

    # Encode and decode it
    encoded <- base64encode(raw)
    decoded <- base64decode(encoded)

    # Compare whether the two strings are identical
    chef.assert(identical(raw, decoded))

    # Get encoded string from jsonlite
    jsonlite_encoded <- base64_enc(str)
    chef.assert(identical(jsonlite_encoded, encoded))

    chef.end(FALSE) # No issue found
}
```

This code could be used with other libraries, even using 3rd party tools outside of the R environment, due to the way S^{2}E works – once one has a program with reference implementation, testing the R libraries become relatively simple.

In this case, running the code actually yields a discrepancy between those two libraries once `strlen` is big enough! It turns out that `jsonlite` inserts newline characters into long base64 strings, while `base64enc` does not. However, this might not be wrong. While newline characters in the encoded strings are forbidden by the RFC 4648 [19, Section 3.1], which is referenced in the documentation, it is not explicitly said anywhere which base64 standard is implemented [20]. Furthermore, this has been reported before and the author hinted that this is not an issue.[42] Regardless, I believe this highlights the ability to use Chef to compare the correctness and the behaviour of various solutions.

[42] `https://github.com/jeroen/jsonlite/issues/305`

5.2 Type checking

The tool made as part of this thesis strive to be general and allow various kinds of analysis to be performed, not only bug finding or some kind of white-box fuzzing. A good example for that is performing type checking.

R is a language that is not statically typed and as such, it is often not known upfront which functions accept or output which types. Furthermore, it might be hard to read from the source code or not available in the documentation in some cases. Due to this, a tool that could infer type information about various functions might be useful in some cases.

I demonstrate this on the `base64encode` function[43]. Consider the following:

```r
library(base64enc)

chef.start()

tryCatch({
  # Try to create a value with variable type
  any <- chef.any(5)
  # Attempt to execute it.
  base64encode(any)
}, error = function(e) {
  # Signal an error
  chef.end(TRUE)
})

# No error happened
chef.end(FALSE)
```

After running this, one can examine the list of successful and failed inputs. It can be presumed that if for given type there exists a value that does not trigger an error, the function accepts the type. If there exists no such value, the function either does not accept the type assuming the analysis finished and all paths were taken.

A simple test like this managed to infer that, for example, passing any `SYMSXP` to the function causes an immediate error,[44] while the function even accepts (and encodes!) logical vectors for instance.[45] It requested about 4.5 GB of RAM.

```
$ chef-tools typeinfer successful_test_cases.json
complex
nil
double
logical
```

[43]Part of the package `base64enc`, which was already used in other tests before.

[44]Which should not be surprising but is a good example of this test providing an information that a type is likely not compatible with the function.

[45]It should be noted that while there are coercion rules in R that can implicitly change type of a value to another one, such as logical to a numeric [21, pg. 54], the effect seen here is not caused by coercion. It was verified by running `base64encode(I(c(T,F,F)))`, as the `I` function inhibits implicit coercion.

Testing the result shows that logical vectors indeed work with the base64 encoding.

```
> library(base64enc)
> base64encode(c(T, F, F))
[1] "AQAA"
```

While this might not be surprising to an experienced R developer, I personally did not anticipate this and it was not mentioned explicitly in the documentation [22]:

```
Usage:

     base64encode(what, linewidth, newline)
     base64decode(what, output = NULL, file)

Arguments:

    what: data to be encoded/decoded. For 'base64encode' it can be a
          raw vector, text connection or file name. For 'base64decode'
          it can be a string or a binary connection.
```

Another example is type checking the builtin `+` operator. The tool can not only provide various types that the function can be run with, but it can also infer what is the output type based on the inputs.

This is done by utilizing the type inferring tool mentioned before in section 4.3. To produce a mapping of input/output types, one has to not only know which types can be passed to the function, but also which types does the function produce. This is done by signalling the type in the source code like so:

```
chef.start()
tryCatch({
  lhs <- chef.any(5)
  rhs <- chef.any(5)
  result <- lhs + rhs
  # The typeinfer tool will know that the type of the result produced
  # was caused by lhs and rhs variables
  chef.debug(paste("TYPEINFER", typeof(result), "lhs", "rhs"))
}, error = function(e) {
  chef.end(TRUE)
})
chef.end(FALSE)
```

After running the type inference tool introduced in section 4.3, the user will be presented with the following:[46]

[46] I did not let this experiment finish on my laptop due to diminishing returns, but it requested around 15.3GB of virtual memory. After a while, the analysis tends to produce drastically less test cases than it was in the beginning, as can be seen in figure 5.2.

```
$ chef_tools typeinfer successful_test_cases.json debug.txt
nil     nil        ->  integer
double  complex    ->  complex
complex double     ->  complex
nil     complex    ->  complex
nil     logical    ->  integer
nil     double     ->  double
nil     integer    ->  integer
(...)
```

I believe that this tool could be used for this purpose, as it should be able to automatically process pretty much any library without the need for human interaction after the initial setup.

5.3 Large-scale testing

One big advantage of symbolic execution is that it is unguided and does not require human presence to run. This allows one to perform large-scale testing. Chef is well-suited for that, given one has the required computing power.

For example, one could run the aforementioned type inferring tool on a large number of functions and packages in order to generate type information for the R ecosystem.

Another thing that could be done would be checking whether there is a package that causes a segmentation fault. One can write an R wrapper that will attempt to symbolically call every function in a given package.

This section presents a simple R script that can be used as a reasonable starting point to such an endeavour. It iterates on every function in a given package, finds it arity and, in this example, tries to call it.

The snippet below[47] tries to find out whether the is a function that triggers a segmentation fault when given unexpected input. The function is called, the result is discarded, as well as an error, should one occur. Errors are expected, and it would be, on the contrary, quite surprising if there were a bigger number of functions that do not throw them at all. After all, the functions are called with a generic symbolic input – any type that Chef supports may be passed to it. Because of this, errors are ignored.

The only moment where Chef would report an error would be in case a segmentation fault occurred. This is thanks to a modification I made. In case the `LinuxMonitor` plugin is turned on, Chef hooks on its segfault monitor, receiving a notification each time a segmentation fault occurs. Should one happen, Chef will generate a test case that leads to that and terminates the state.

[47]The snippet is incomplete for brevity, it does not include the `arity` function not important to this example. This should illustrate the point, but in case the reader is curious about the full code, it is available in the `chef-nextgen` repository (see appendix A) as file `main/utils/fuzzer.R`.

```r
ignore_errors <- function(expr) {
  tryCatch(
    {
      eval(expression)
    },
    error = function(e) { }
  )
}

fuzzPackage <- function(package) {
  functions = ls(paste('package:', package, sep=''))

  for(function_str in functions) {
    fun <- get(function_str)
    ar <- arity(fun)

    chef.start()
    chef.debug(paste("Started testing", function_str))

    # Fill each argument with 10-bytes-long symbolic value
    # of any type
    ignore_errors({
      args <- sapply(1:ar,
        function(x) chef.any(paste(function_str, "_", x, sep=''), 10))
      do.call(fun, as.list(args))
    })

    chef.end(F)
  }
}

library(jsonlite)
fuzzPackage("jsonlite")
```

Such code could be used for variety of cases, not limited to hunting segmentation faults or type information generation, after some modifications. I hope this highlights the possibilities that Chef offers and its flexibility regarding adapting to different kinds of analysis with ease. While a test like this would be unlikely to catch more delicate errors, it could be used to massively test large part of the R ecosystem for low-hanging fruit, given one has the computing power required.

This test, albeit quite simple, actually managed to find a double-free bug in one of the eleven tested libraries, as discussed further in section 5.5.[48]

[48]It took about eight hours of running the affected library and consumed 14GB of RAM at the peak. It was tested on Intel Xeon Gold 6140 (72) @ 2.294GHz, 188GiB of RAM.

5.4 Performance

Thankfully, a lot of work was poured into S^2E and other projects used transitively by Chef, which translated into a quite good performance. S^2E is even able to symbolically execute large projects, such as Microsoft Office [7]. Chef inherits these great performance characteristics, making running the whole interpreter symbolically viable.

I decided to measure the overhead Chef has over executing R code natively with a stock interpreter.[49] This is likely to have a large impact on which codebases are deemed testable with the symbolic execution engine. I've chosen several popular R libraries[50] of various sizes, extracted example usage and tests, and ran them with and without Chef. This should give the reader an overview of how well Chef scales with bigger libraries. Note that no symbolic variables were defined during the experiments. This is because their choice heavily depends on the selected project, and depending on their choice, might not ever terminate in practice – such as when arrays with unbounded lengths are tested. What I am interested in this section is merely the overhead over running R natively.

The results shown in figure 5.1 were all run on a server[51] used only for this purpose, single-core. Median time was used for comparisons. Time spent is recorded by `hyperfine`,[52] 10 runs. It is either spinning up the whole S^2E within Qemu (estimated overhead can be seen at the `empty` test results) or running the R interpreter directly on the source code. The numbers in the table below were determined by running the test suite of each project.

Package	KLOC[a] R	KLOC C/C++	KLOC tests	Chef	Plain R	Slowdown
Nothing[b]	0	0	0	198.6s ± 4.5s	0.2s ± 0.0s	993×
`magrittr`	0.6	0.8	0.4	848.8s ± 21.6s	0.8s ± 0.02s	1061×
`glue`	1.2	0.4	1.2	893.4s ± 25.1s	1.8s ± 0.03s	496×
`withr`	0	2.4	2.0	821.2s ± 46.3s	4.0s ± 0.1s	205×
`stringr`	3.4	0	1.1	1038.1s ± 65.6s	3.4s ± 0.05s	305×
`jsonlite`	2.2	5.7	1.1	888.0s ± 26.2s	3.5s ± 0.07s	254×
`digest`	1.1	9.1	1.4	536.7s ± 14.5s	0.6s ± 0.2s	895×
`tibble`	4.8	0.4	4.2	1246.9s ± 89.7s	19.7s ± 0.2s	63×
`sf`	13.9	0.1	5.6	445.0s ± 11.4s	0.3s ± 0.02s	1483×
`cli`	17.1	15.2	7.9	850.0s ± 34.4s	28.2s ± 0.2s	30×
`vctrs`	15.0	42.7	22.4	29683.9s ± 1390.8s	52.1s ± 0.5s	570×

[a] Thousands of lines of code
[b] No libraries, empty source code. Used for measuring overhead on spinning up the VM with S^2E.

Figure 5.1: Benchmarking results

One can see that although there is a significant slowdown, the results are not terrible at all! Chef is often only a few hundred-times slower than the R implementation run natively without a virtual machine, which is, in my

[49]The R interpreter was used without any modifications, version `4.1.2`, built for Ubuntu 22.04.4 LTS.
[50]Chosen thanks to `https://www.r-pkg.org/`
[51]Intel Xeon Gold 6254 (16) @ 3.099GHz, 24GB RAM.
[52]A benchmarking software, `https://github.com/sharkdp/hyperfine`.

opinion, not a bad result. The average runtime was 10.4 seconds for plain R and 56 minutes for Chef, which is 327-times slower. Furthermore, regarding simple overhead of running Chef, I did not uncover any correlation between code size and the Chef slowdown.

Regarding the performance with symbolic inputs, it is rather hard to draw a single conclusion. The runtime depends heavily on the code that is being tested and varies greatly. Furthermore, it is not needed to actually finish the analysis and explore all paths of a given program for the analysis to yield interesting and useful results. The rate of discovery of new test cases can be inspected in figure 5.2.

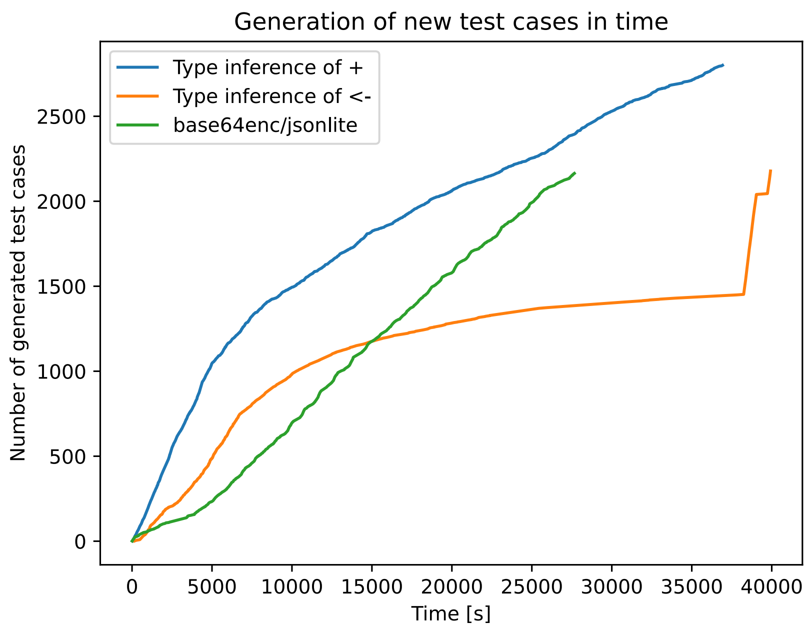

Figure 5.2: New test cases discovered by Chef over time

It can be seen that generation of new test cases is quite fast at the beginning, but begins to slow down after a while. There might be some spikes time to time, where the SMT solver resolves a long-running query and S^2E generates a bunch of states shortly after, as seen in the end of the `<-` type inference run.

5.5 Discussion

In the previous sections, I presented few categories of tests that I believe could be used with Chef.

Library corectness

Testing libraries is in a state where it could be used in practice as of now. The biggest downside is the human factor: someone has to write the symbolic tests. While it might be easy to test parts of the codebase with symbolic tests,[53] this is unlikely to be automated and requires human work. Still, this is in my opinion an area where it could be used. It should be noted that the tests take tens of minutes to hours of computing time in order to run and requires few gigabytes of RAM, depending on the test. Because of that, I see the potential usage as part of some continuous integration suite that is run overnight, and not on regular developer's computer.

One downside I experienced was generation of symbolic variables with variable length.[54] Currently, this is solved by having a loop representing the size of the symbolic variable, as seen in the example with `base64enc` and `jsonlite`. While it works, it can take a very long time to reach higher iterartions of the loop – how much, exactly, depends entirely on the individual test.

I see two viable solutions to this problem: one is offloading the problem to the user. Chef is capable of creating many different symbolic variables, and they can be combined together. If one wanted to, for example, create a symbolic string of variable length, it might suffice to create a raw vector of given size and then producing a string out of data of the raw vector, up to the first nullbyte. Or the user might generate a symbolic integer, constraint it to a specific interval using `chef.assume` and only use the first few characters of the string as the test. These will not be as fast as native symbolic variable generation, but could be used regardless.

The second solution to this problem would be adding a support directly to Chef. However, this is not trivial and would require some additional modifications that are not part of this thesis. It might improve performance in this scenario, if it is implemented in the future.

In general, I would recommend to use Chef for this kind of scenarios for software where testing is important and one has the required computing power, as long as they do not need to use symbolic variables with variable length or do not mind the performance loss.

Type checking

Type checking is, in my opinion, one of the most interesting features of Chef. It can find out accepted types of both libraries and builtin functions, and prove that by generating an example. It can infer those types in a matter of hours. I can see using Chef in a project that would, for example, be dedicated to providing typing information about the R ecosystem, or about the internal

[53]Or property-based tests, in general.

[54]One cannot simply pass a symbolic variable as a parameter to a `chef` function expecting size, for example the `chef.string`. The value will be concretized and only will be used, rendering this process ineffective.

builtins, where the types would otherwise often have to be inferred from the code itself.

The disadvantage is that unless it is left run to the end, it generates a potentially incomplete set of types. However, all of them should be sound, which is still an interesting property and something I can imagine some projects being able to work with.

Large scale testing

Mass-testing R libraries is in my opinion a functionality with great potential. One can automatically test large areas of the R ecosystem and discover bugs without the need of any human action. However, it is hard to write good tests that would be able to test any R package thoroughly. In section 5.3, I presented code that would check whether a given library crashes, which is applicable mostly to libraries written in a lower level language, such as `C` or `C++`.

I tested a small set of libraries. It was not a massive scan of the R ecosystem, I aimed at verifying whether that approach might be viable, with potential large-scale verification being done later.

I have searched `r-pkg.org` for recently updated libraries that require compilation, with the idea that this does not favour popular libraries and avoids old libraries that are not being used anymore. Out of these, I filtered them by hand to discard libraries that serve as database connectors for example, since running them without preparing a setup first would be likely useless and I did not anticipate any useful findings. I came up with list of 11 libraries[55] that were fuzzed with hope that a segmentation fault could be triggered.

Out of these, `kit` is vulnerable to the following memory bug:

```
> library(kit)
Attaching kit 0.0.17 (OPENMP enabled using 1 thread)
> clearData(c(NULL, NULL))
double free or corruption (out)
[1]    2963451 IOT instruction (core dumped)  R
```

From inspecting the source code, it looks like the library lacks good verification on what is passed into its cleanup function. It tends to assume that it is an internal object and runs various memory-freeing operations on it, generating variety of memory problems, depending on what the user passed there.

This shows that even simple tests like the one I came up with earlier can be useful and can find bugs, even in maintained libraries.

Memory consumption

The last thing I would like to discuss here is memory consumption. While the R interpreter is relatively lightweight and S^2E consumes well under a gigabyte of memory, the situation becomes worse when a code with heavy state forking is ran. As highlighted in section 4.1, the memory consumption grows lineary with

[55]`fmtr`, `jsonlite`, `tclust`, `transport`, `tinycodet`, `stringi`, `kyotil`, `kit`, `dbarts`, `sp` and `lda`.

the number of states. This is not manageable for larger test cases, for example, running eight tests concurrently overnight accumulated about 180 gigabytes of memory usage due to heavy state forking. As this amount of memory required can be a dealbreaker for some users, I created a workaround in the form of a plugin that tracks both local and global memory use, and can prevent new states from being created when the memory pressure is high. The user can learn more about configuration of this tool in appendix A.

Conclusion

This thesis aimed to implement a symbolic execution engine for R utilizing the Chef technique.

In order to achieve this, an analysis of existing symbolic execution engines – S^2E and Klee – was performed, researching the principles of symbolic execution as well as implementation of those two engines.

As Chef is an existing S^2E plugin that was to be altered together with the R interpreter, the inner functionality of both Chef and R was researched and analysed. Chef was then modified and parts of it were rewritten in order to function with modern S^2E. The R interpreter was altered to support symbolic execution with Chef, and some performance and user experience optimizations were made.

The result is a working tool consisting of multiple components packaged in a docker image with a focus on ease of use by the users. It supports generating several types of symbolic variables, segmentation fault detection and optimizes state selection using CUPA. Two tools are provided, one for generating test cases out of symbolic run and another one for automated generation of type annotations.

Several examples are provided on how to discover functionality discrepancies between libraries, generate type annotations of various functions or R builtins, and how to mass-analyse R libraries without user interaction.

An evaluation was done on the performance and usability of the solution. One discrepancy was found in the implementation of `base64` encoding in two libraries during the evaluation, as well as one use-after-free bug in another library.

The Chef authors, whose work this thesis is based on, claimed that it took them merely a few person-days to build a symbolic execution engine for Python and Lua. While the whole implementation took me much longer than that due to learning how to use the libraries, understanding the interpreter, setting up a reliable environment and modifying Chef to work with modern S^2E, I assume that with a working environment and experience with using both S^2E and Chef, making a new symbolic execution engine for another interpreted language could truly take me around five to ten person-days in the future, especially if I was already familiar with the language. This observation seems to be in line with the experience of the authors of Chef.

Overall, I personally find Chef to be quite useful in a variety of cases. While the experiments can take a while to run and tend to consume large amount of memory unconstrained, it still can prove interesting properties about the programs and I see the potential of this tool as part of some analysis run overnight or in a CI setup. While I find it unlikely for this to become part of a regular development routine, there are plenty of cases where it could be used, a notable one being type inference in my opinion.

Future work

The tool is just a starting point in the realm of symbolic execution of R. In order to enable widespread adoption, support for more symbolic variables could be added, supporting all values constructible in R in the ideal case. CUPA usage could also be improved with more heuristics and strategies in order to facilitate more efficient execution.

While large-scale analysis of the R ecosystem is already possible with this tool, adding the above features could make a non-negligible difference in real-life testing and large-scale analysis.

More thorough scan of the R ecosystem could also be done, utilizing the technique highlighted in this thesis or coming up with new, more advanced ones.

Bibliography

[1] W. N. Venables, D. M. S.; the R Core Team. An Introduction to R. 2023, accessed on 13.08.2023. Available from: `https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf`

[2] Bommarito, M. J.; Bommarito, E. An Empirical Analysis of the R Package Ecosystem. *SSRN*, 2021. Available from: `https://dx.doi.org/10.2139/ssrn.3788978`

[3] Yang, J.; Sar, C.; et al. Automatically generating malicious disks using symbolic execution. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, 2006, pp. 15 pp.–257, doi:10.1109/SP.2006.7. Available from: `https://doi.org/10.1109/SP.2006.7`

[4] Bucur, S.; Kinder, J.; et al. Prototyping symbolic execution engines for interpreted languages. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, Salt Lake City Utah USA: ACM, Feb. 2014, ISBN 978-1-4503-2305-5, pp. 239–254, doi:10.1145/2541940.2541977. Available from: `https://dl.acm.org/doi/10.1145/2541940.2541977`

[5] Baldoni, R.; Coppa, E.; et al. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, volume 51, no. 3, May 2019: pp. 1–39, ISSN 0360-0300, 1557-7341, doi:10.1145/3182657. Available from: `https://dl.acm.org/doi/10.1145/3182657`

[6] Cadar, C.; Sen, K. Symbolic execution for software testing: three decades later. *Commun. ACM*, volume 56, no. 2, Feb. 2013: p. 82–90, ISSN 0001-0782, doi:10.1145/2408776.2408795. Available from: `https://doi.org/10.1145/2408776.2408795`

[7] Chipounov, V.; Kuznetsov, V.; et al. S2E: a platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, volume 46, no. 3, Mar. 2011: p. 265–278, ISSN 0362-1340, doi:10.1145/1961296.1950396. Available from: `https://doi.org/10.1145/1961296.1950396`

[8] Hasabnis, N.; Sekar, R. Extracting instruction semantics via symbolic execution of code generators. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*,

Seattle WA USA: ACM, Nov. 2016, ISBN 978-1-4503-4218-6, pp. 301–313, doi:10.1145/2950290.2950335. Available from: `https://dl.acm.org/doi/10.1145/2950290.2950335`

[9] Burns, P. The R Inferno. 2011, accessed on 09.03.2023. Available from: `https://www.burns-stat.com/pages/Tutor/R_inferno.pdf`

[10] R Core Team. R Documentation (Extract). 2023, accessed on 13.08.2023. Available from: `https://search.r-project.org/R/refmans/base/html/Extract.html`

[11] Wickham, H. Advanced R. 2023, accessed on 13.08.2023. Available from: `https://adv-r.hadley.nz`

[12] Goel, A.; Donat-Bouillud, P.; et al. What we eval in the shadows: a large-scale study of eval in R programs. *Proc. ACM Program. Lang.*, volume 5, no. OOPSLA, oct 2021, doi:10.1145/3485502. Available from: `https://doi.org/10.1145/3485502`

[13] Kvapil, O. Properly based testing. 2023, a lecture on property-based testing given at Prague #lang-talk meetup. Available from: `https://github.com/lang-talk/meetups/blob/d4e3db0/README.md#lang-talk-meetup-vol-2`

[14] MacIver, D. R.; Hatfield-Dodds, Z.; et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, volume 4, no. 43, 2019: p. 1891, doi:10.21105/joss.01891. Available from: `https://doi.org/10.21105/joss.01891`

[15] Hypothesis documentation. 2024, accessed on 07.08.2024. Available from: `https://hypothesis.readthedocs.io/en/latest/`

[16] Cadar, C.; Dunbar, D.; et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA: USENIX Association, Dec. 2008. Available from: `https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems`

[17] R Core Team. R Internals. 2023, accessed on 13.01.2024. Available from: `https://cran.r-project.org/doc/manuals/r-release/R-ints.html`

[18] R Core Team. Writing R Extensions. 2024, accessed on 22.04.2024. Available from: `https://cran.r-project.org/doc/manuals/r-release/R-exts.html`

[19] Josefsson, S. The Base16, Base32, and Base64 Data Encodings. RFC 4648, Oct. 2006, doi:10.17487/RFC4648. Available from: `https://www.rfc-editor.org/info/rfc4648`

[20] Ooms, J.; Lang, D. T. jsonlite: A Simple and Robust JSON Parser and Generator for R. 2023, accessed on 4.5.2024. Available from: `https://cran.r-project.org/web/packages/jsonlite/jsonlite.pdf`

[21] Adler, J. *R in a Nutshell*. O'Reilly, 2010, ISBN 978-0-596-80170-0.

[22] Šimon Urbánek. base64enc: Tools for base64 encoding. 2015, accessed on 10.4.2024. Available from: `https://cran.r-project.org/web/packages/base64enc/base64enc.pdf`

Appendix A

Chef usage

First of all, let me clarify the source code organization. While not vital for usage from the point of view of a regular user, it is crucial for any kind of development.

Due to its nature, Chef is split into multiple repositories.[56]

- github.com/SoptikHa2/s2e (branch `chef`) contains modification of S^2E to include Chef plugin. This includes everything mentioned in this thesis that has something to do with S^2E, for example, CUPA.
- github.com/SoptikHa2/r-chef-symbolic-execution (branch `chef`) contains the R Interpreter modified to work with Chef and S^2E. This includes particularly calls to S^2E and custom `chef` functions.
- gitlab.fit.cvut.cz/stastpe8/chef-nextgen is a storage for docker images and packages. It includes Dockerfiles vital to building both development and production-ready images.
- gitlab.fit.cvut.cz/stastpe8/chef-testcase-parser is a Python library that parses test cases generated by S^2E. This is used by tools that work with the Chef's output in some way.
- gitlab.fit.cvut.cz/stastpe8/chef-tools contains some tools that automate the processing of Chef output. Two notable examples[57] are the test case generator, which turns Chef test case file into an R `testthat`-compatible file with tests, and the type inference tool that can extract type annotations of functions from a Chef run.

However, the only thing vital to the user is knowledge of how to run a Chef docker image and how to run the tools. If one wants to make their own tools, I suggest taking a closer look at the last two repositories in the list above.

[56]Historically, there was some work done even on the old Chef before the modifications to support modern S^2E. These are available at `https://github.com/SoptikHa2/s2e-old/tree/chef` and `https://github.com/SoptikHa2/chef-symbex-lua`. Note that those are in a proof-of-concept state and are not the result of this thesis one is likely looking for. I include them here for the sake of complexity, as they contain some fixes that might make running old Chef much simpler if one desired to do so.

[57]And the only examples, so far.

In order to run Chef, one has to create a file with R source code. For example, let's assume there is the following code in a file called `source.R`.

```
chef.start()
chef.debug("Running!")
number <- chef.int()

if (number < -4) {
    chef.debug("Something bad happened")
    chef.end(T) # An error occured
}

chef.assert(number != 14)

chef.end(F) # No error occured
```

After that, the only thing to remain is to run Chef. It took about 130 seconds to run on a powerful computer,[58] so one should expect this to take a few minutes.

```
docker run -v "$PWD":/data <docker-image> run <R-source> <out-dir> [libs]
```

The docker images available are in the `chef-nextgen` repository, but one can simply use `gitlab.fit.cvut.cz:5050/stastpe8/chef-nextgen/r-symbex:thesis`.

In this case, make sure you are in the directory where you put the `source.R` file. Run `docker run -v "$PWD":/data gitlab.fit.cvut.cz:5050/stastpe8 /chef-nextgen/r-symbex:thesis run source.R out-1`.[59] This will run Chef and put all experiment data into a new folder called `out-1`. This will take some time. You can end it prematurely by pressing `C-c`, but that might yield incomplete results.

In order to be able to process results, one might want to install the provided `chef-tools` command-line utility. To do so, follow instructions in the `chef-tools` repository or run the following:

```
pipx install chef-tools --index-url
https://gitlab.fit.cvut.cz/api/v4/projects/60767/packages/pypi/simple
```

Now, look around the generated `out-1` directory that was generated by Chef. Most of the files follow the generic S^{2}E structure. The chef-specific files are the test cases and the `output` directory with captured `stdout` of the R process.

One can explore the `debug.txt` file for detailed information about the execution. The captured standard output of the interpreter can be seen in the `outfiles` directory, and the `stats.csv` contains statistics about the execution, such as memory usage.

However, the most interesting files are `successful_test_cases.json` and `err_test_cases.json`. There are numerous successful test cases, but only two erroneous test cases, which is just what was expected.

[58] AMD Ryzen 9 5900x, 32GB RAM.

[59] The target architecture is currently set to `x86-64-v2`. If you encounter an error like `Illegal instruction`, consult `https://github.com/HenrikBengtsson/x86-64-level` and either buy a newer CPU or build the docker image yourself. The Dockerfile is in the `chef-nextgen` repository in the directory `main`.

```
[
{
    ...
    "inputs": [
    { "name" : "v0_int__number_0",
        "bytes" : "0xe 0x0 0x0 0x0",
        "i32" : 14,
        "string" : "\\xe\\x0\\x0\\x0"}
    ]
}
,{
    ...
    "inputs": [
    { "name" : "v0_int__number_0",
        "bytes" : "0xfa 0xff 0xff 0xff",
        "i32" : -6,
        "string" : "\\xfa\\xff\\xff\\xff"}
    ]
}
]
```

In order to see an example of one tool in the `chef-tools` utility, go to the `out-1` directory and run the following:

```
chef-tools testgen err_test_cases.json ../source.R > testthat.R
```

This will generate an R script that can be run with `testthat`.

```
library(testthat)

chef_fun_to_test <- function(number) {
    print("Running!")

    if (number < -4) {
        print("Something bad happened")
        expect_false(T) # An error occured
    }

    expect_true(number != 14)
    expect_false(F) # No error occured
}

context("Tests generated by Chef")

test_that("Test case 0", {
    chef_fun_to_test(14L)
})
test_that("Test case 1", {
    chef_fun_to_test(-6L)
})
```

Run it via `Rscript testthat.R` and verify that it indeed fails the test.

Configuration

The docker image contains sane defaults that most users do not have to touch. However, should one want to, there is an option to do that. The script that starts S^{2}E detects whether there is a file called `s2e-config.lua` in the root folder that the user passed to the docker image – so most likely the current working directory if the user followed the previous example to the letter.

The sample configuration can be found at the `chef-nextgen` repository.[60] One of the modifications the user might want to do is adding a memory limit.[61] It is advised that the user puts the following near the end of the file to the `User-specific scripts` section.

```
add_plugin("MemoryForkLimiter")
pluginsConfig.MemoryForkLimiter = {
    -- 70% of system memory usage used up (across all processes)
    -- maxGlobalMemUse = 700

    -- 5GB used up by this instance of S2E
    maxMemoryUseBytes = 5000000000
}
```

[60] `https://gitlab.fit.cvut.cz/stastpe8/chef-nextgen/-/blob/master/main/s2e-config.lua`

[61] It does not hard-limit S^{2}E memory usage, but rather stops generation of new states if the memory consumption rises to a certain level.

Appendix **B**

Contents of attachment

`readme.md`	an overview of contents of attachment
`src`	
`chef-nextgen`	infrastructure needed to build the thesis
`chef-testcase-parser`	library for parsing generated test cases
`chef-tools`	set of tools working with S^2E output
`r-interpreter`	modified R interpreter
`s2e`	modified S^2E
`thesis`	LaTeX sources of this thesis
`experiments`	experiment data from chapter 5
`text`	
`thesis.pdf`	compiled thesis in PDF format