Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Security Research Group

A Generalised Implementation of Symbolic Execution Using the Z3 Theorem Prover

Master Thesis

Amber Schippers

Supervisors:
prof. dr. Jurgen Vinju
Kevin Valk

Eindhoven, October 2022

Abstract

Due to a shortage in security analysts, a company called Codean aims to build a Review Environment (RE) to make security analysts more productive. The objective of this thesis is to create a tool applicable to multiple programming languages to be implemented in the RE. It makes use of symbolic execution to find a set of values for variables that satisfies all constraints to reach a particular line of source code. Tree-sitter is used to parse source code from different programming languages to an intermediate representation. The constraints imposed on the reachability of each line are extracted from the resulting syntax trees. The Z3 Theorem Prover computes a model that satisfies the constraints imposed on a line. A subset of the C preprocessor language is utilised as a test subject for the tool. While the tool seems to be extendable to other programming languages, the results indicate the possibility of the tool giving back false positives and false negatives. A more detailed evaluation is needed in future work. In order to know the severity, the tool needs to be applied to real use cases. To properly evaluate whether the tool can be extended to other languages, additional languages need to be implemented.

Contents

Chapter 1

Introduction

Where there is technology, there are hackers. This means that anyone who deals with valuable digital information needs to properly protect it. It is becoming more difficult now than ever, due to the increase in software complexity and number of criminal hackers [65]. Along with this comes an increase in cost: Based on historical cyber crime figures, it is expected that over the next four years, global cyber crime costs will grow by 15 percent per year [47]. By 2025, the global costs are estimated to reach over 10.5 trillion euros. Therefore, preventing cyber crime is becoming increasingly important.

Security vulnerabilities stem from the lack of, or incorrect verification of, input and output data. A security vulnerability in a program allows for someone to make this program behave in unintended ways. This can have drastic consequences such as allowing this person access to read and alter sensitive information. These mistakes in source code are easily made, as many programmers have never been taught how to prevent them: Out of the 24 top universities in the United States, only one undergraduate program of computing science requires the student to take a security course [8]. Besides the great number of security vulnerabilities, there is a shortage of people in cyber security to mitigate the vulnerabilities: It is estimated that, globally, there are over 1 million unfilled cyber security jobs [14].

When analysing source code for security vulnerabilities, it is paramount to ensure that each verification step is done correctly. In the case of large code bases or complex software, these verification steps can be scattered throughout the code base, which makes it difficult for a human to keep track of and verify that all individual verification steps together properly protect the input and output data. Automatic tools for vulnerability testing intended for software developers, while fast, require human involvement to interpret the results [54]. This means that automatic tools cannot replace a human tester. They can, however, be of support for security analysts.

Codean is a company developing a toolbox named the Review Environment (RE) [20]. The RE's purpose is to improve on the traditional Integrated Development Environment (IDE) when it comes to analysing the security of code. Indeed, the IDE is designed specifically for writing code instead of reviewing its security. The RE will be optimised for security code review with tools that aid the security analyst in reviewing more code in less time by automating tedious tasks. It is estimated that only 15.74 percent of programs are written in the most popular programming language, Python [61]. Hence, it is essential for the RE to support multiple languages.

To aid security analysts in code reviews, certain software testing techniques can be utilised. Symbolic execution is one of these techniques that helps identify software vulnerabilities by analysing the effect certain inputs have on how a program executes [39]. This technique abstractly represents variables as symbols, without requiring specified input values. The result is a set of constraints for each unique control flow path in the analysed code to be executed. From these constraints, a constraint solver can be used to form instances for each variable that would produce property violations. This helps the security analyst identify states that one should not be able to reach with a certain input, that can in fact be arrived at.

There are several existing implementations of analysis tools using symbolic execution. However, these implementations have one single target language, focus on several instruction sets like x86, or support an intermediate representation like LLVM. The aim of this paper is to present a more generalised implementation for a subset of the C preprocessor (CPP) language, that could be expanded to include other imperative programming languages.

This results in the following question: Can we construct symbolic execution in a way to make it work for multiple programming languages? This challenge consists of three parts, namely

- converting source code to an intermediate representation in a way that can be applied to other programming languages;
- extracting a set of symbolic constraints for a given line of code from this representation; and
- solving these constraints.

To simplify extracting the constraints from the textual source code, a parser-generator tool named Tree-sitter is used to parse it into a machine-readable data structure [62]. To generate satisfiable values for variables, Microsoft's Satisfiability Modulo Theories (SMT) solver Z3 is utilised [48]. It is expected that this will result in a complete implementation of symbolic execution, that can be expanded to programming languages that are supported by Tree-sitter.

Chapter 2

Problem description

Keeping in mind the importance of preventing cyber crime and the shortage of people that prevent it as motivated in Chapter 1, the security analyst needs to become more efficient. This chapter introduces the security analyst as well as the specific domain knowledge and theory that will be employed in this paper.

2.1 Security analysis

Software security is concerned with the software behaving as intended when under attack by a malicious attacker. Security testing is not the same as software testing. While both are concerned with ensuring the software functions as intended, software security is concerned with the software behaving as intended specifically when under attack by a malicious user. Software testing only tests for the normal use of the program.

The goal of security analysis is to identify flaws that could be exercised or exploited to cause harm to a system. Such a flaw is called a vulnerability, and is formally described as an "internal fault that enables an external fault to harm the system" [3]. There are multiple classifications of vulnerabilities that can be exploited in different ways.

To mitigate a vulnerability, ideally one should avoid it. However, as discussed in Chapter 1, software developers lack the knowledge to do so. If a vulnerability is not avoided, it can still be found, fixed and monitored. This is where security analysis comes in.

Security analysis can often be split into two parts, namely black box testing and white box testing [49]. In black box testing, the internal structure and code of a program are not used in the analysis. Only the external aspect of the program is tested, which means the behaviour of the program in response to abnormal inputs is observed. However, in white box testing, the tester can make use of the implementation of the program in the form of source code in order to spot vulnerabilities. Instead of output, the internal operations are analysed as well. Mixing these techniques, where the implementation of the program is partially known, is called grey box testing.

The more information the security analyst has, the more informed the assessment will be. Unfortunately, white box testing has its cons. Besides requiring a skilled tester to identify vulnerabilities from code, this testing technique is time intensive, especially when dealing with a large code base. Reviewing projects consisting of over 300,000 lines of code is not unusual at Codean.

A security analyst can use tools to automate some tasks. These tools can be divided into two groups. Dynamic analysis involves evaluation of a system during runtime. Examples of this are fuzzing, logging and tracing, or using an interactive debugger. Static analysis entails evaluating a system by examining its source code without executing it. A simple example is style checking, where warnings are given when one is in violation of naming conventions.

Automatic testing can be used as support for the security analyst. It can save the security analyst time by performing repetitive and time-consuming tasks. Automatic testing tools cannot, however, replace security analysts. In their current form, the tools work fine for simple tasks. Nonetheless, they have not tackled complex tasks [45]. Besides, the vulnerabilities the tool finds need to be interpreted and often mitigated by a human. Furthermore, the tool can present false positives, meaning it reports a vulnerability where there is none.

2.2 Symbolic execution

For certain sections of code, restrictions may be imposed on input values for the code to execute. Think of an IF statement, for instance. The conditions imposed by this IF statement need to hold true in order for the code block contained by the statement to execute. Either this code block always executes, it never executes or the result of the conditional statement is dependent on input.

In nested conditional statements, there could be a number of constraints on a variable. These constraints on the input accumulate into what is called the *path constraint*. An example in pseudocode is illustrated in Figure 2.1. In line 4, a constraint is placed on the variable X. This means that lines 5-11 can only be reached if X satisfies this constraint. In other terms, lines 5-11 are control dependent on line 4. Line 5 presents another constraint for X. This constraint's domain is line 6. This makes the path constraint for line 6 the accumulation of the constraint imposed on line 5 and the constraint imposed on line 6. If a path constraint is satisfied, an execution can follow the particular associated path. Symbolic execution offers an overview of what the restrictions are on input values for a certain program path. Instead of the input consisting of fixed values as is the case in concrete execution, in symbolic execution input values are represented by symbols denoting arbitrary values. These symbols are called symbolic inputs. An interpreter walks through the program using these symbolic inputs, and each input-dependent variable it encounters is expressed in terms of the symbolic inputs. It also collects constraints on the possible outcomes of each conditional branch the program can take. The set of constraints for a certain path is known as the *path condition*.

These different branches or paths a program can take can be represented in a tree. If you add the constraints to reach the paths, one gets what is called a control flow graph or execution tree. Figure 2.2 depicts the execution tree for the code in Figure 2.1. A node shows a conditional statement and the outgoing branches show what occurs when the condition holds and when it does not.

On each level you go deeper into the tree, a constraint is added. These constraints can be solved in order to have an example of what possible inputs will cause a certain branch to be executed. Line 12 in the pseudocode example in Figure 2.1 is unreachable, since X would need to be smaller than 3 and equal to 3 at the same time. Due to this contradiction, this line is dead and it is not represented in the execution tree.

Surprisingly, symbolic execution does not involve the execution of a program. It is a form of static analysis. Symbolic execution has uses including program proving, software testing (symbolic debugging), generation of test data and in proving the program quality [23]

Unfortunately, symbolic execution has its limits. Most importantly, as programs get larger, the number of feasible program paths can grow exponentially and may cause the tool to run out of memory. If a program contains unbounded loops, this leads to path explosion and symbolic execution will follow all of these paths, causing it to never terminate [2]. There are tricks to alleviate path explosion like merging similar paths into one [41].

Another weakness of symbolic execution is its inherent complexity. It is hard to symbolically reason about calls to operating-system, library functions and program statements like pointer manipulations and floating-point operations [2]. This issue can be mitigated by using concrete values to simplify constraints. The result is a simplified, partial symbolic execution [33].

Symbolic execution is different from unit testing. In a unit test, concrete values are presented to the program to inspect whether the unit they affect behaves as expected. However, symbolic execution aims to explore the paths of the entire program by using symbolic values and concludes the values that lead to those paths. Unit testing will not catch system-level errors or errors on the integration between the units.

Figure 2.1 Pseudocode example

```
$I =< input >$;
$X = I - 6$;
$y \leftarrow 0$;
if $X < 3$ then
    if $X = 1$ then
        $y \leftarrow 10$;
    end if
    if $X = 2$ then
        $y \leftarrow 20$;
    end if
    if $X = 3$ then
        $y \leftarrow 30$;
    end if
end if
```

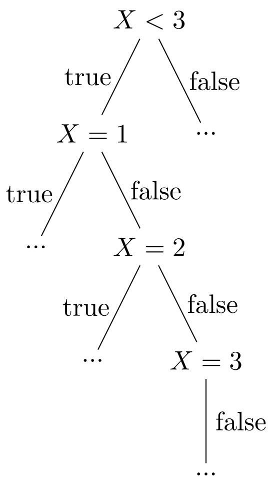


Figure 2.2: An execution tree illustrating the control flow of Figure 2.1

Another similar, but different, static analysis tool is abstract interpretation. It can approximate all possible runtime states of a program. Like symbolic execution, it maintains constraints (in the form of invariants or path conditions) during execution [1]. What it cannot do, however, is explore all possible execution paths of a program.

2.3 The C preprocessor language

A preprocessor is used by a compiler to modify a program before compilation. It is, however, not a part of the compiler itself: It is a separate step in the compilation process. The compiler is instructed by the preprocessor to carry out necessary preprocessing before the compilation of the next language occurs.

The CPP is a language-independent tool that serves as the macro processor for the C, C++ and Objective-C programming languages [58]. The CPP is often called a macro processor since it allows the user to define macros. A macro is a brief instruction, which automatically expands into a larger set of commands to carry out a particular task. It can be seen as an abbreviation, providing a convenient way to represent common programming idioms.

Besides being a macro processor, the CPP has further uses. Some of its primary capabilities are inclusion of header files, conditional compilation, line control and diagnostics.

- Header files are files with extension *.h* that can be substituted into your program. The files contain C function declarations and macro definitions. Header files allow for easy sharing of its contents between source files.
- Conditional compilation allows the user to include or exclude blocks of code in the program by setting various conditions.
- Line control permits the user to set the line number and file name of a source file, which will result in more meaningful error messages from the subsequent compiler.
- The CPP supports the use of diagnostics, meaning it allows for problems to be detected at compile time instead of runtime by printing error or warning messages.

The CPP is widely used in practice and it can be a great aid in reducing programmer effort, improving portability, performance and readability when used in a disciplined matter [28]. However, its users believe the CPP makes C programs more difficult to understand and modify [28]. Its lack of structure, while increasing flexibility, may complicate the comprehension of the C source code by both programmers and tools. Even the designer of the C++ language has some critiques about the CPP language: "Occasionally, even the most extreme uses of CPP are useful, but its facilities are so unstructured and intrusive that they are a constant problem to programmers, maintainers, people porting code, and *tool builders*" [59]. Nevertheless, the CPP is heavily used [28].

The grammar of the CPP is related to the grammar of the C programming language. The main dissimilarity is the use of the hash symbol (#). All operations in the CPP language are triggered by preprocessing directives. In the CPP language, directives start off with this hash symbol. The hash is followed by the name of the directive. Some directives are followed by arguments. The CPP language consists of 25 directives. A list can be found in Table 2.1. It is good practice to start directives in the first column to maintain readability.

$\#assert$	deprecated alternative to macros
$\#define$	define a certain macro
$\#elif$	stands for else if, when the $\#if$ does not hold
$\#else$	when the $\#if$ and $\#elif$ do not hold
$\#endif$	end a conditional
$\#error$	stop compilation
$\#ident$	a string constant is copied into a special segment of the object file
$\#if$	test whether the expression that follows it is true
$\#ifdef$	return true if a macro is defined by $\#define$
$\#ifndef$	return false if a macro is defined by $\#define$
$\#import$	deprecated alternative to $\#include$
$\#include$	insert a header from another file
$\#include\_next$	insert file with this name that occurs after current file in the directory
$\#line$	specifies the original line number
$\#pragmaGCCdependency$	issue a command to the compiler
$\#pragmaGCCerror$	issue a command to the compiler
$\#pragmaGCCpoison$	issue a command to the compiler
$\#pragmaGCCsystem\_header$	issue a command to the compiler
$\#pragmaGCCsystem\_header$	issue a command to the compiler
$\#pragmaGCCwarning$	issue a command to the compiler
$\#pragmaonce$	issue a command to the compiler
$\#sccs$	no documentation of what it should do
$\#unassert$	undo $\#assert$
$\#undef$	cancel the definition of a macro
$\#warning$	continue compilation, but issue a warning message

Table 2.1: A list of all C preprocessor directives [30]

2.4 Tree-sitter

Code written in a form readable by humans first needs to be translated in order to be understood by a machine. The translation is performed by an interpreter or compiler. One component of this translator is called a parser. Parsers break the input they get into smaller elements, preparing it for further translation.

Tree-sitter is an open-source parsing toolkit developed at GitHub and written in C and C++ [62]. Where each programming language has a parser to go from source code to machine-executable code, Tree-sitter can serve as a parser for multiple programming languages. This allows the transformation of source code from different languages to machine-readable data structures with the same interface. The generated data structure is in the form of a concrete syntax tree. As a parser generator with tree matching support, it is comparable to Rascal[40], StrategoXT[11], DMS[7], TXL[21] and ANTLR[51].

Figure 2.3 shows an example of a syntax tree generated by Tree-sitter. The example is based on the source code given in Listing 2.1. The output given by Tree-sitter was converted to the graph description language DOT and visualised by GraphViz [26].

```
#if X < 3
    #if X == 1
        ...
    #endif
#endif
```

Listing 2.1: A simple example written in C

Each node from the syntax tree holds some information. A node has a type, text, start point, end point, list of children and a parent. Each of these properties can be visualised in the syntax tree as seen in 2.3.

This means the syntax tree can be traversed like a traditional tree: Given a node, one can access its parent, siblings and children. One starts at the root node. Additionally, Tree-sitter allows querying the syntax tree. Such a query must contain one or multiple patterns and will return any matches found. The pattern denotes the node's type, and optionally, the types of the node's children.

Tree-sitter offers incremental parsing, which means that once a file is parsed, it will not need to be re-parsed again entirely when the file is edited. Instead, the edited information is added on or subtracted from the previous parsing. This increases speed and uses less memory than similar parsers which do not offer incremental parsing, since parts of the old and new file are shared. Additionally, Tree-sitter offers error recovery. This means that even if the source code is invalid by containing errors, Tree-sitter will not abort and give an error message. Instead it inspects the code and finds the start and end of the invalid code section and give a useful syntax tree regardless.

Tree-sitter has language bindings in 11 different languages and it can currently parse 45 languages, with an additional 18 in development.

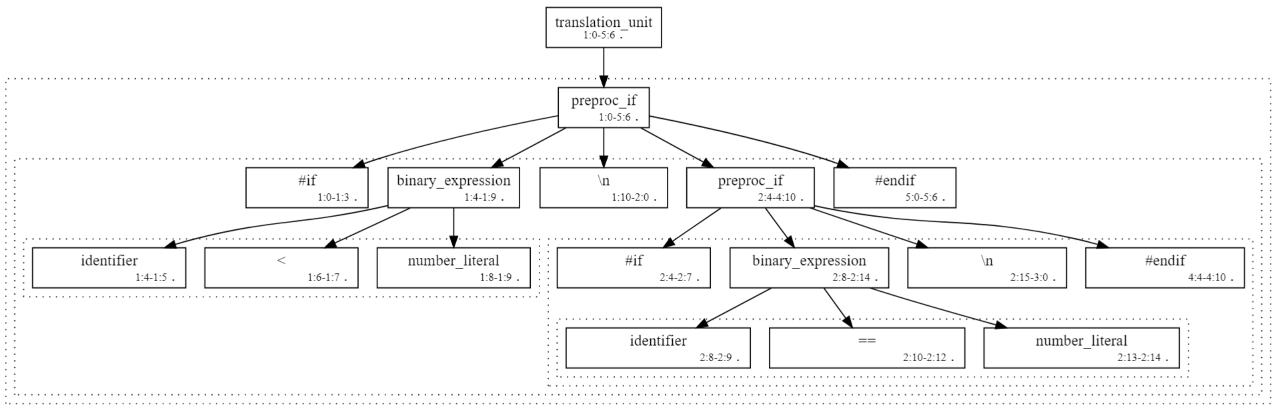


Figure 2.3: An example of a syntax tree produced by Tree-sitter

2.5 The Z3 theorem prover

The boolean satisfiability problem (SAT) determines whether an interpretation exists such that a certain boolean formula is satisfied. If we generalise this to mathematical formulas instead of boolean formulas, we get a more complex problem involving equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories [48]. An SMT solver is a tool that solves SMT problems. Microsoft Research has developed such an SMT solver named the Z3 Theorem Prover (Z3). It was designed specifically for program verification and dynamic symbolic execution, although today it is utilised more generally in software verification and analysis applications. Z3 has language bindings in C, C++, .NET, Java, Python and ML/OCaml.

To use Z3, one creates variables of certain data types in Z3 that do not contain a value. Constraints can be added onto these variables. For example, some integer has to be larger than a certain number. When constraints are created, we say the constraints have been asserted in the solver. The solver can check whether the mathematical formula is satisfiable, and if so, suggest valid values for the variables. This solution is also called a model or witness for the set of asserted constraints. A model is an interpretation that makes each asserted constraint true.

Chapter 3

Preliminaries

3.1 Symbolic execution

It is a popular concept since introduced in the 1970s, with some of its fundamental papers describing different systems like the Select system [10], the EFFIGY system [39], the DISSECT system [35] and Clarke's system [18]. Since then, symbolic execution has been mentioned in over 32.000 papers, of which a little over 2000 were published in the last year [1].

Different adaptations of symbolic execution have been developed. To name some examples, differential symbolic execution reduces cost of symbolic execution and improves the quality of analysis results by making use of the fact that different versions of a program are largely similar [53]. Selective symbolic execution is a technique that aims to create the illusion of full-system symbolic execution, yet it symbolically runs only the code that is of interest to the developer [16]. Directed symbolic execution can be used when one is looking for program executions that reach a particular target line, instead of the entirety of the source code [44]. Another example is probabilistic symbolic execution, which estimates the probability of executing portions of a program [32]. However, a language-generalised adaptation of symbolic execution has not been developed before.

Concolic execution is a testing technique based on, and similar to, symbolic execution [55]. Concrete execution, that is "regular" execution with concrete values, is combined with symbolic execution. The idea is to simplify implementation of symbolic execution by running it alongside concrete execution, each receiving feedback from the other. Similar to the goal presented in this paper, the aim of concolic testing is to generate inputs that would exercise all the feasible execution paths of a sequential program [55]. However, these inputs are not generated for one line in source code only. Additionally, the method of reaching the goal differs as well: The first path that gets explored is random. The next path follows the previous one, except the result of one conditional argument gets flipped with the help of symbolic execution. This means that in the worst case, all paths would need to be computed to find the one we desire. Concolic execution is an extremely expensive but also valuable method as it can show the absence of security issues next to their presence. Large companies like Microsoft employ it in their vulnerability analyses of high-profile software like Microsoft Office and Windows [38].

3.2 The C preprocessor language

Similar to the aim of this paper, the goal of Hu et al. is to use symbolic execution to find the conditions for any given preprocessor directive or C/C++ source code line to be compiled [36]. Nevertheless, this is not the objective of the paper you are reading. To reach the goal in this paper, one must first implement a tool which finds the conditions for any given line of code to be

[1] as found on Google Scholar.

executed. The key is that it needs to be generalised for as many languages as possible, and the CPP language is merely a guinea pig. The tool implemented by Hu et al. is not suitable to be extended to multiple languages easily, since CPP specific extractors and analysers were developed and used.

A more scaleable approach for symbolic execution in the CPP language was implemented by Latendresse [42]. This implementation can be brought into play with large C/C++ software. It utilises symbolic execution under the term "symbolic evaluation". For the symbolic representation, the term "conditional values" is coined. The notion of conditional values used in the paper avoids the combinatorial analysis of paths that occurs in traditional symbolic execution, making for a faster version. The author went on to write a more complete rendition of the same tool [43]. Where the first implementation lacked in its interaction with macro expansion and evaluation, the latter implementation transforms them into Boolean expressions to be handled by the symbolic evaluator.

3.3 Tree-sitter

The design of Tree-sitter was inspired by six papers [68] [64] [67] [66] [60] [52]. While these differ in their area of research, the most prominent area is generalised left-to-right derivation parsing (GLR parsing), which Tree-sitter is built on top of [19]. What sets this parsing algorithm apart from others is its capability to parse any context-free grammar [25].

Tree-sitter is integrated in the Atom text editor as well as the GitHub website to handle syntax highlighting [62]. However, GitHub also uses it for a form of static analysis similar to symbolic execution, namely symbolic code navigation. Symbolic code navigation allows the developer to select a named identifier in source code to navigate to the definition of that entity, and it allows the opposite as well: given a definition of some identifier, it can list all the uses of that identifier within the project [19].

3.4 The Z3 theorem prover

The idea of using the Z3 theorem prover to implement symbolic execution is not a new one. This is no surprise, as fundamental papers of symbolic execution hint at the use of a theorem prover [39] [23]. Additionally, Z3 is the leading solution for SMT solving [4]. Z3 has been used for symbolic execution in popular analysis platforms like Mayhem [15], SAGE [34] and Angr [56].

Other notable SMT solvers include cvc5 [5], Bitwuzla [50], Boolector [12], CVC4 [6], MathSAT [13], OpenSMT2 [37], SMTInterpol [17], SMT-RAT [22], STP [31], veriT [9] and Yices2 [24].

Chapter 4

Methods

The objective of this paper is to find a method to implement a tool that can find a set of values that satisfies the conditions imposed on any given line of code to be executed, for as many languages as possible.

The full process can be split into three main parts.

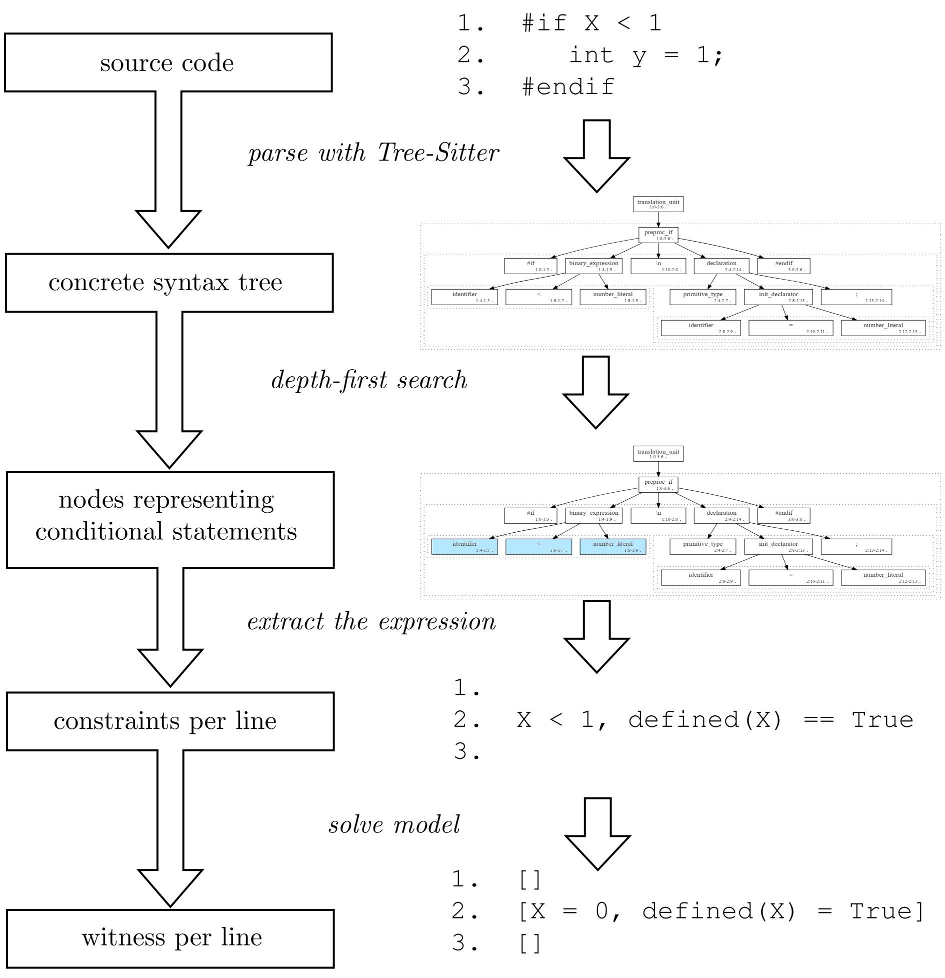
- The first part involves parsing the source code into an intermediate representation so that source code from distinct programming languages is portrayed in the same manner. In this paper, the intermediate representation is materialised in the form of a concrete syntax tree. The syntax trees are generated from the source code with the help of Tree-sitter. This parser was selected due to its wide-reaching scope of languages it can parse, as well as its efficiency in doing so. Additionally, the concrete syntax trees produced by Tree-sitter are easy to traverse as well as easy to comprehend when visualised.
- In the second part, the aim is to go from the intermediate representation to a list of constraints for any given line of code to be executed. These constraints can come from multiple lines of code, that is, multiple positions and nodes in the tree. The tree needs to be traversed and the constraints need to be appended to a list.
- Finally, for each variable in the path constraint for any given line, a value needs to be given so that the list of constraints for this line is satisfied. This set of values that satisfies the given constraints is called a witness. A witness can be found using an SMT solver. The selected SMT solver for this implementation is the Z3 theorem prover, as it is the leading solution for SMT solving [4].

The full pipeline can be seen in Figure 4.1. The security analyst can use the witness of a line to visualise its otherwise long list of constraints. Additionally, the witness can be used as input to see how the program behaves before and on this particular line.

To illustrate an example and assess how well this pipeline could work, the CPP language is used. To create a minimal viable prototype, the implementation assumes variables to be of the integer type. Additionally, expressions containing logical operators (&& and ||) are ignored. Unsupported types trigger an error in the implementation.

The tool is implemented in Python as both Tree-sitter and Z3 have a Python API. Codean had a preference for Python, since Python is known for its active and large community.

Due to the fact that security analysts inspect all lines in source code instead of just one, the choice was made to compute a witness for all lines in a single run of the tool. This is feasible as the source code does not change often, meaning the tool would not have to recompute frequently. Additionally, this means the tool would only need to be run at start-up and the security analyst will not need to wait for the tool to compute a witness each time they require one for a certain line.



```
1.   #if X < 1
2.       int y = 1;
3.   #endif
```

```
1.
2.   X < 1, defined(X) == True
3.
```

```
1.   []
2.   [X = 0, defined(X) = True]
3.   []
```

Figure 4.1: The full pipeline of the tool.

4.1 Parsing the source code

The Tree-sitter documentation lists detailed instructions to build the library and use the parser [63]. Once the library is installed and imported, one can create a parser, set its language, and build a syntax tree based on the source code. The source code is be passed to Tree-sitter as a string. The root note can then be obtained from the syntax tree. Tree-sitter allows the tree to be printed as an S-expression, in case one wants to inspect it.

By default, the syntax tree produced by Tree-sitter is a concrete syntax tree. This means that for every individual token in the source code, a node is created. If details like commas and parentheses are not needed, Tree-sitter can produce an abstract syntax tree instead. Even if the type of tree would not matter for analysing source code written in the CPP language, concrete syntax trees are used in this implementation. This is done due to the intention of expanding to other languages. Using an abstract syntax tree creates the possibility of important details being left out.

4.2 Extracting constraints

Now that the syntax tree for the entire source code is obtained, the constraints imposed on each line need to be found. These are computed in the order in which the lines appear in the source code. To do this, the tree is iterated over fully while keeping track of the current line. Once the last line is handled, this step is complete.

Using Z3, a general purpose solver object can be created. Constraints can be added to this solver at any time. When a constraint is added, we say it has been asserted in the solver.

The constraints need to be expressed as Z3 variables in order to add them to the solver. For an integer, this is as easy as using the Int('x') function [27]. The 'x' is the name of the Z3 variable. If there are multiple constraints on the same variable, this variable should have the same name in the Z3 variables as well.

If a solver holds and solves the constraints, and each line of code has different constraints imposed on it, we need a solver for each line of code. Luckily, many lines of code share the same constraints. We say lines for which the constraints are identical belong to the same scope if they are enclosed by the same conditional statement(s). Consult Listing 4.1. Clearly, lines 3-5 belong to the same scope. One can also plainly see that lines 7-9 belong to a different scope than line 3-5 do. Finally, even though lines 13-15 have the same simplified constraints imposed on them as lines 3-5 do, they belong to a different scope as the lines are not enclosed by the same conditional statements.

```
 1  #if X < 3
 2      #if X == 1
 3          ...
 4          ...
 5          ...
 6      #elif X == 2
 7          ...
 8          ...
 9          ...
10      #endif
11  #endif
12  #if X == 1
13      ...
14      ...
15      ...
16  #endif
```

Listing 4.1: A section of C code, serving to explain scopes

Lines belonging to the same scope only need one single solver to compute a witness that suits all of them.

The nodes are handled on a case by case basis. A node is only of interest when it imposes a constraint. In the CPP language, constraints are imposed by the directives #ifdef, #ifndef, #if, #elif and #else. When a node of any of these types is encountered, it means a constraint is imposed on the set of lines it encapsulates.

To find the set of nodes that affect a line, the information each node contains is utilised. As raised in Chapter 2.4, each node has a start- and end point. The start point consists of two numbers separated by a colon (:). The first number denotes the line on which it is called. The column on which it is called is indicated after the colon. The end point works similarly. The range of a line, that is, the lines from the start point up to and including the end point, showcases the lines it affects.

As mentioned earlier, the constraints on the lines are computed in the order of which the lines appear in the source code. This means we start at line 1. Depth-first search (DFS) is used recursively to traverse the tree. This ensures the nodes are visited in the same order as the lines they represent.

If the start point of a node is higher than the number of the line we are currently working on, it means all nodes that affect this line have passed and the list of constraints is complete. When a constraint is added, a new scope is entered. A copy of the current constraints is made, and given to a new solver to add on to. In the CPP language, one exits a scope when an #endif , #elif or #else statement is encountered. It can also be derived from the end points given in the nodes.

Once all nodes in the tree have been visited, this step is complete.

4.3 Finding a witness

When the constraints for each scope have been extracted and asserted in the solver, all that is left is to find a witness for each line of code. This is handled by the Z3 theorem prover.

The Z3 method *check*() solves the asserted constraints [27]. If a solution is found, this method will return *sat* for "satisfiable". This means a set of numbers exists for which all constraints hold, that is, a witness can be found. If no witness exists for this set of constraints, the method returns *unsat* for "unsatisfiable". In other terms, one can say the system of asserted constraints is infeasible. Besides these two possible results, it is also a possibility that the solver is unable to solve for this set of constraints. In that case, the *check*() method will return *unknown*.

The *model*() method can produce a witness. In this context, "model" is a synonym for "witness", that is, an interpretation that makes each asserted constraint true [27]. This method will result in an error if the *check*() method gave anything but *sat* as its output. If not, it produces a list with each item containing the name of a Z3 variable that was used in any number of constraints, along with a value for that variable that makes the constraints satisfiable.

Chapter 5

Results

In order to estimate whether this tool could be useful for Codean, we need to evaluate whether this implementation...

- ...can be extended to other languages.
- ...can be valuable to the security analyst.

5.1 Extension to other languages

Adding a language to the tool should be realistically possible to do without an excessive amount of effort.

5.1.1 Implementing a new language step by step

The individual steps are as follows.

Step 1: Parse Add the desired language to the Tree-sitter parser.

Step 2: Identify Identify any statements or expressions in the language that impose a condition.

Step 3: Visualise and explore Visualise example trees for each type of conditional statement, in order to get a grasp of names and structures in the syntax tree.

Step 4: Locate conditional statements When traversing the tree using depth-first search as already implemented, ensure that when a conditional statement is encountered, additional steps are performed.

Step 5: Locate expressions Iterate from the node representing the conditional statement to the expression contained within it.

Step 6: Locate elements From the expression, extract the identifier(s), operator(s) and literal(s). These can also include other types, depending on what the language allows in an expression, like function calls.

Step 7: Infer types Infer the type of the identifier(s) from the literal(s) and operator(s).

Step 8: Form constraints Form the constraints by creating a Z3 variable for each identifier and assigning it a constraint using the operator(s) and literal(s). Add the constraints to the solver.

Step 9: Negations If any conditional statements indicate the negation of any previous statement, this negation should be added to the solver as a constraint.

Step 10: End of scope If any conditional statements indicate the end of any previous statement, an older solver should be used. This is already implemented.

5.1.2 Implementing the C preprocessor language step by step

As an example, this is what each step would look like for the CPP language.

Step 1: Parse The desired language can be added to a list. The parser checks the extension of the source code and selects the appropriate language from the list.

Step 2: Identify The conditional statements in the CPP language are #ifdef, #ifndef, #if, #elif and #else.

Step 3: Visualise and explore An example tree for an #if statement can be seen in Figure 5.1.

Step 4: Locate conditional statements The representation of an #if statement in the syntax tree is called 'preproc_if', so if a node matches this type, additional steps need to be taken.

Step 5: Locate expressions In the example, the expression is of the type 'binary_expression', which is in all cases the direct child of the conditional statement. Longer expressions are split into multiple binary_expressions. Recursion is used to formulate the entire expression.

Step 6: Locate elements Iterate through the expression's children to find its components. This step has already been implemented. However, it does not work for logical operators.

Step 7: Infer types The type of the node that is not an identifier, is of the type 'number_literal'. For the CPP language, this means the identifier is an integer.

Step 8: Form constraints To form the constraint, a z3.Int() variable is created and the operator and 'number_literal' are used to form the constraint. Add the constraint to the solver using '*solver*.add(*constraint*)'.

Step 9: Negations #elif and #else statements indicate that the previous conditional statement must not hold. The previous conditional statement of #elif and #else statements is its parent in the syntax tree. The negation of the constraints imposed by the parent node is added to the solver.

Step 10: End of scope #endif, #elif and #else statements indicate that the previous statement no longer holds.

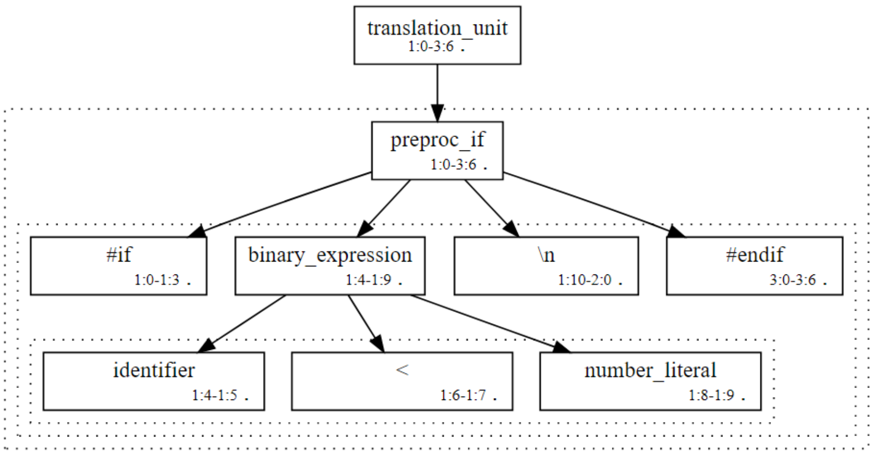


Figure 5.1: An example tree of the #if conditional statement

5.1.3 Analysing each step of the CPP case

Each individual step is analysed for effort and likelihood of error. A measure used in statistics is the number of Type I and Type II errors, also known as 'false positives' and 'false negatives', respectively. In this scenario, a false positive occurs when the tool incorrectly lists a witness that could never occur. A false negative entails the tool missing a witness that should have been there.

Step 1: Parse Adding the desired language to the Tree-sitter parser is done by simply creating a Language object from the Tree-sitter library in the list of languages that can be used. The languages are already implemented by Tree-sitter and thoroughly tested for errors. This step takes minimal effort and will not impact the accuracy and precision of the tool.

Step 2: Identify Identifying the conditional statements of a language should be done by looking at its documentation. It is often clearly defined and, again, should be simple to do correctly.

Step 3: Visualise and explore Visualising the way the conditional statements are represented in the syntax tree is a way of preventing error later on. It gives an overview of different structures that can occur so that one can take it into account before starting the implementation of the language. Therefore, the more time spent on it, the better. Missing a possible structure could result in false positives, as a constraint could be missed and a false witness would be given. However, new structures can be added at any time. This step is very time intensive.

Step 4: Locate conditional statements Finding lines that impose a constraint is done by checking the type of each node when the tree is being iterated. This step is simple and quick.

Step 5: Locate expressions Once a line that imposes a constraint is found, the corresponding expression can be found in its children. The type of the nodes can be checked to see if it matches any type of expression, be it unary, binary, or other if the language allows it. Binary expressions have already been implemented. If something was missed in the visualisation step, an expression could be overlooked which could result in missing constraints entirely. This gives rise to false negatives. If the visualisation step was done correctly, however, this step does not take much effort.

Step 6: Locate elements Extracting the elements of the expression is more strenuous, as one does not know what to expect. Depending on the language, expressions could have numerous different structures. One might have to iterate the tree in complex ways to access each element. The visualisation step is, again, fundamental to this step. It could take long to explore all structures and make sure that in each expression, all elements are extracted. Mistakes here are easily made, but can also easily be found by testing the code from the visualisation step. Layered binary expressions (for example, $X * 2 + Y == Z$, which can be divided into multiple binary expressions) are already handled in this implementation.

Step 7: Infer types Once all the literals and operators are known, one can infer what type the identifier is. If there are no literals or arithmetic operators in the expression ($X == Y$, for example), one does not know the type. Giving them the wrong type may result in false positives and false negatives, as one constraint that should not be there could be added, and another missed. A constraint can still be formed as long as one sets all identifiers as the same type. Within one expression, all identifiers and literals are of the same type. Expressions separated by logical operators (&& and ||) can contain different types. Depending on the types that can occur in an expression in the language, the type might be difficult to keep track of. Giving the wrong type only occurs in the absence of arithmetic operators and literals as described above.

Step 8: Form constraints Creating a Z3 variable from an identifier can be done in a single statement. This part is already implemented for integers and linear arithmetic. Every type that can occur in a conditional statement in the language needs to be implemented, along with all operations that could be performed on it. For each element, the type is found either by inspecting the type of its node (for operators and literals), or the type of the nodes surrounding it (for identifiers). Then, the identifiers and literals need to be translated into Z3 types or 'sorts'. If that the sort is not supported, Z3 allows one to define their own sorts [46]. Once the constraint is complete, it is added to the solver automatically. If one needs to define new sorts, this process can get difficult. However, once all the sorts needed are available, this step is simple and fast. If no mistakes were made in the previous step, the implementation of this step will not affect the accuracy and precision of the tool.

Step 9: Negations In conditional statements that negate the previous statement, one needs to iterate to the node of the previous statement, compute its constraints like described in the steps above, and add the negation of the constraints to the solver. This should take minimal effort, as computing the constraints is already implemented. False positives and false negatives can only arise if mistakes were made in the previous steps.

Step 10: End of scope In conditional statements that end the previous statement, a scope is exited and the previous version of the solver needs to be used. This is already implemented, but one needs to indicate the statements this is triggered for. If one knows which statements this applies to, it is easy to implement.

To illustrate the claim that switching to a new language is facilitated by using a parser generator like Tree-sitter, we demonstrate the commonalities and differences between comparable code snippets in different languages in Figures 5.2 to 5.5. However. this can not be taken as a complete or sound analysis. It is meant to illustrate the expectation that adding the next language may be an effort comparable to the steps taken for the CPP.

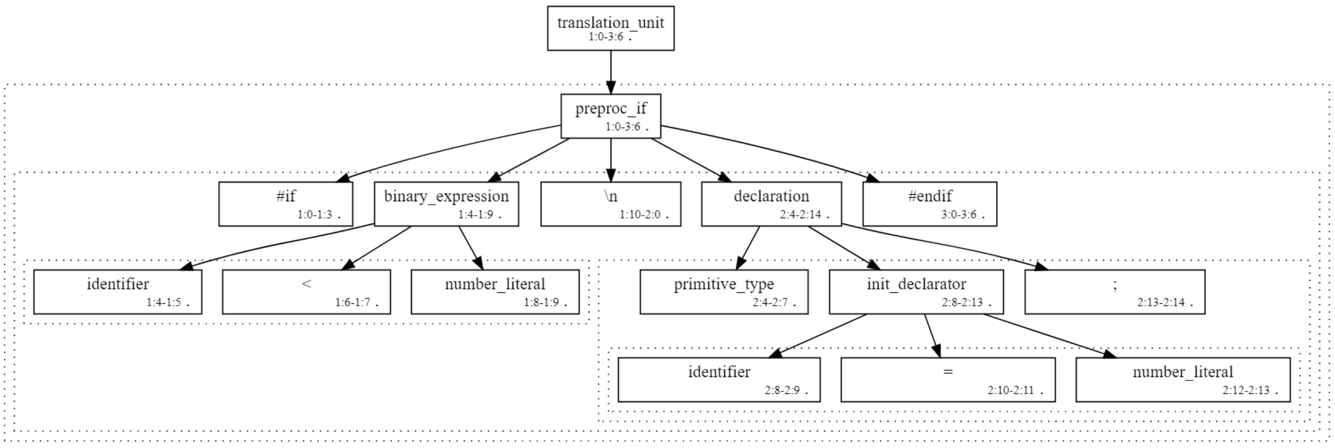


Figure 5.2: A syntax tree of a simple if statement in the CPP language. The corresponding code can be found in Listing 5.1.

```
#if X < 1
        int y = 1;
#endif
```

Listing 5.1: A simple if statement in the CPP language.

```
if (x < 1) {
```

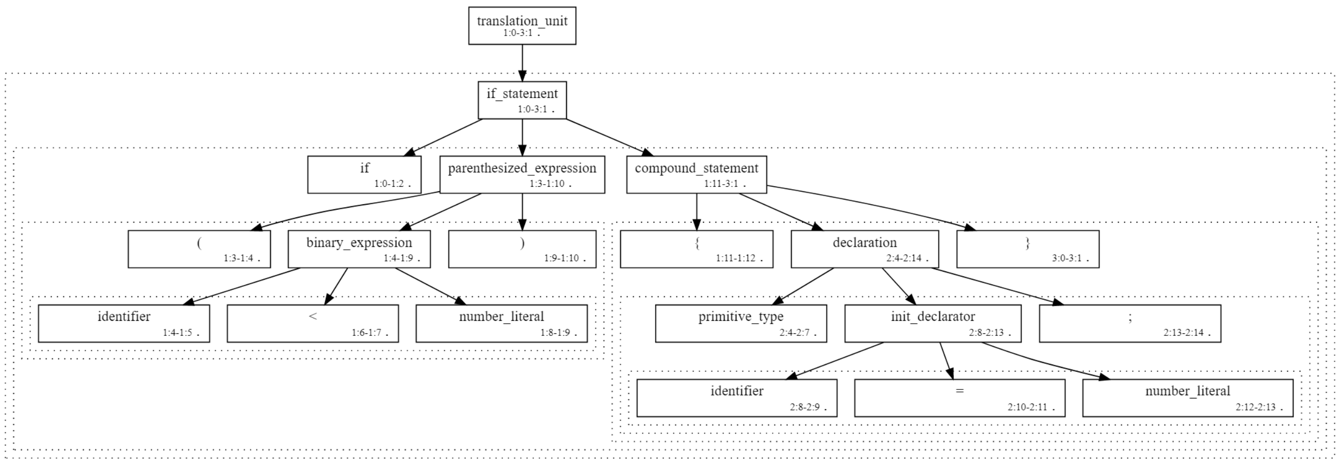


Figure 5.3: A syntax tree of a simple if statement in the C programming language. The corresponding code can be found in Listing 5.2.

```
    int y = 1;
}
```

Listing 5.2: A simple if statement in the C programming language.

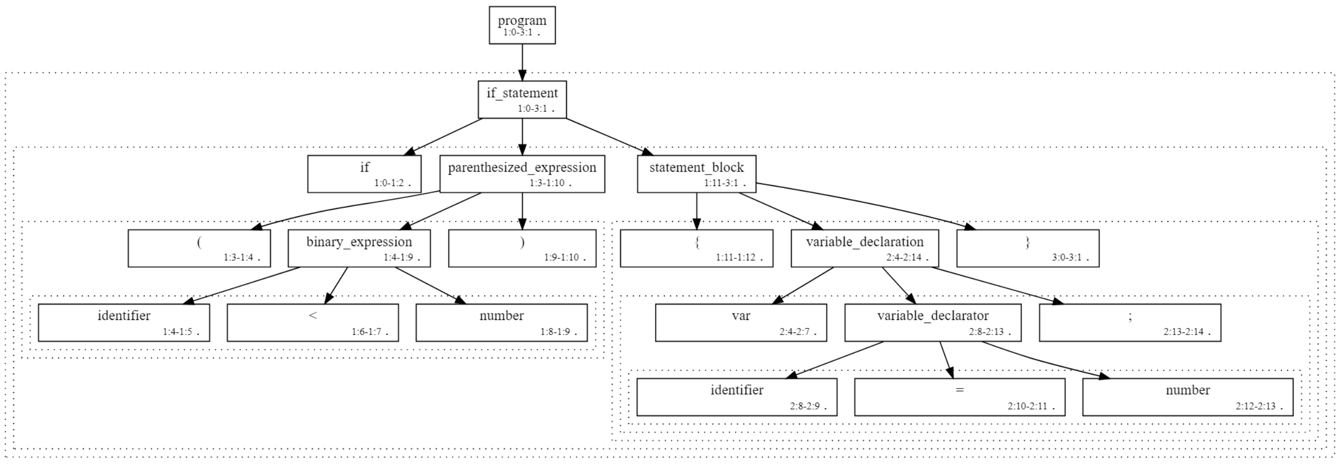


Figure 5.4: A syntax tree of a simple if statement in the JavaScript programming language. The corresponding code can be found in Listing 5.3.

```
if (x < 1) {
    var y = 1;
}
```

Listing 5.3: A simple if statement in the JavaScript programming language.

```
if x < 1:
    y = 1

```

Listing 5.4: A simple if statement in the Python programming language.

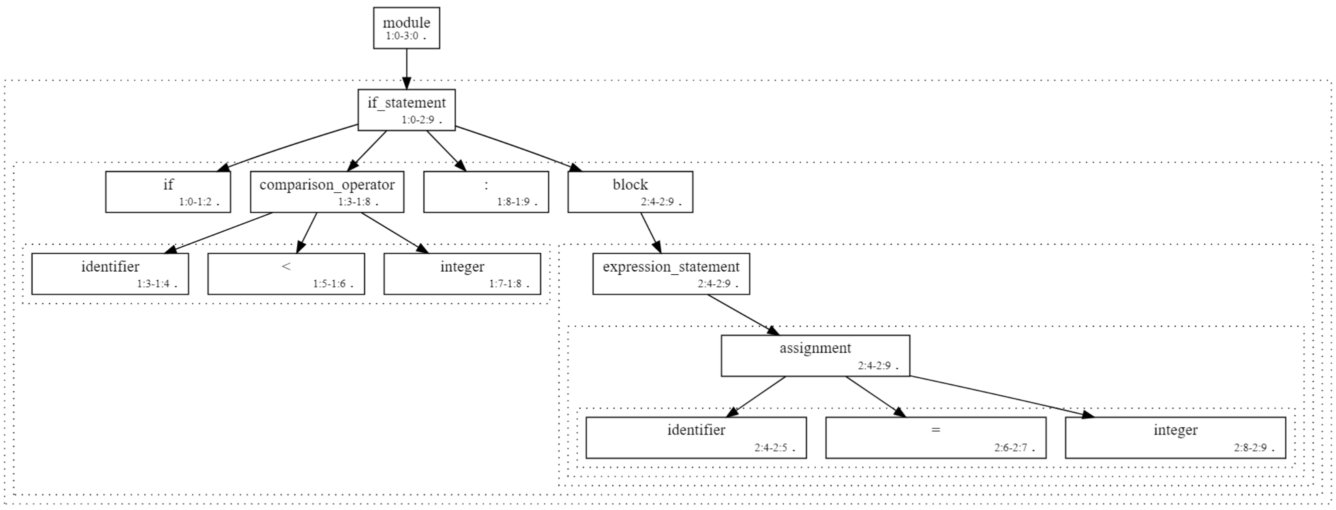


Figure 5.5: A syntax tree of a simple if statement in the Python programming language. The corresponding code can be found in Listing 5.4.

5.2 Value to the security analyst

In order for the tool to be valuable to the security analyst, it needs to do its job quickly and correctly.

5.2.1 Runtime

Compared to other SMT solvers, Z3 is relatively fast. The SMT-COMP 2022 mentioned in 3.4 measured the performance of Z3 and other SMT solvers making models for quantifier-free equality and linear arithmetic sequentially [57]. Out of 891 benchmarks, 852 were handled correctly and the remaining 39 benchmarks resulted in a timeout.

To test the scalability of the implementation, the runtime is measured against the number of conditional statements over 100 iterations. This is computed in different settings. The first is a setting in which the conditional statements are not nested, as depicted in Listing 5.5. In the second setting, the conditional statements are nested like in Listing 5.6. Each conditional statement further restricts the variable. The last setting contains nested conditional statements, but the constraints in each statement are imposed on a different variable. An example can be seen in Listing 5.7. The results are shown in Figure 5.6.

```
#if X < 1
        ...
#endif
#if X < 1
        ...
#endif
```

Listing 5.5: A piece of code that is not nested, containing two conditional statements, with one variable per statement.

```
#if X < 2
        #if X < 1
                ...
        #endif
#endif
```

Listing 5.6: A piece of code that is nested, containing two conditional statements imposed on the same variable, with one variable per statement.

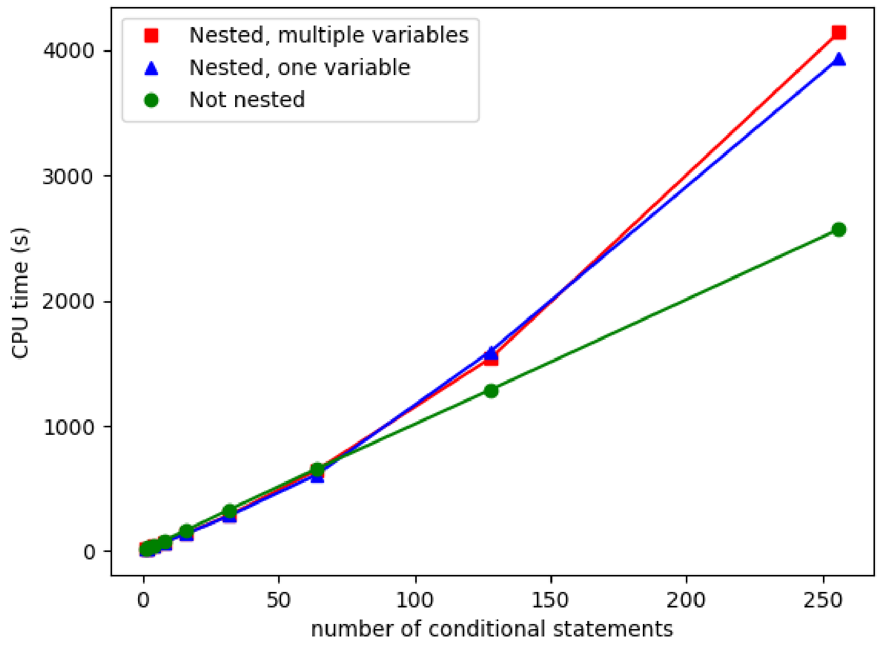


Figure 5.6: The runtime of the tool in different circumstances, measured over 100 iterations.

```
#if X < 1
    #if Y < 1
        ...
    #endif
#endif
```

Listing 5.7: A piece of code that is nested, containing two conditional statements imposed on the different variables, with one variable per statement.

The figure indicates close to linear time complexity for the 'not nested' setting. Both 'nested' settings show a similar runtime to each other. The time complexity for the 'nested' settings is worse than the 'not nested' setting.

5.2.2 Accuracy and precision

In the context of the CPP language, the tool can give false positives and false negatives. One reason is that the current implementation cannot handle logical operations (&& and ||). Additionally, in expressions that do not contain literals and operators, one does not know the type of the identifiers. If one chooses not to add a constraint in this situation, this will result in false negatives. If one instead chooses to guess the type of the identifiers, constraints could be imposed on the wrong type. This implementation assumes that every variable is an integer. If the identifiers turn out to be characters instead, it results in both false negatives and false positives.

It is currently assumed that all variables are independent of each other. If this is not the case, a constraint on one variable might trigger a constraint on a different variable. This second constraint would be missed, which results in false negatives.

False positives occur when an incorrect constraint is given. In theory, Z3 can give false positives in complex, undiscovered situations. However, due to the limited directives and types supported by the CPP language, this does not occur in practice.

Chapter 6

Discussion

6.1 Threats to validity

Claims made about adding languages other than the CPP language could be questioned, since just a single language has been implemented. To make a more accurate estimation of how difficult it would be to extract constraints from a tree in a language other than the CPP, one can compare syntax trees produced for different programming languages to the one produced on CPP source code.

Figures 5.2 to 5.5 show the syntax trees produced by Tree-sitter for similar pieces of code, in different languages. The differences in the trees are not so much in the structure, but more so in the terms used for certain types of nodes. Additionally, the structural differences that can be seen, are due to the difference in syntax in the languages. These particular dissimilarities in structure are to be expected.

However, it is not the syntax that needs to be translated to constraints. The witnesses need to work semantically. This opens up a whole new world of complex translations. Consider the C++ programming language, where operators can be overloaded. How does one translate an operator to a constraint, if it has more than one definition? A similar problem is encountered for method calls in conditional statements. In the Java programming language, two methods could have the same name, but still differ due to the amount of arguments they take. How does one translate the appropriate method?

From the information that is currently collected, it is hard to estimate the amount of work it would take to add another language. It appears that the current design of the tool does not stand in the way of adding more languages, but additional research will have to determine its feasibility.

6.2 Complexity of adding a new language

The results show step by step instructions to add a new language to the existing implementation. With complex languages, the ‘visualise and explore’ step could take up a lot of time. If someone is already familiar with the language or the general structures Tree-sitter produces, time could be saved. This means adding a new language when one has already added one before could increase efficiency.

Complex steps are the ‘locate elements’ and ‘infer types’ ones. Because of their complexity, both steps take a longer time to implement than other ones, and the steps are more probable to produce errors.

The ‘locate elements’ step has already been implemented for binary expressions, but other languages could have more types of expressions. For example, the C programming language offers conditional expressions. Once implemented for one language, the ‘locate elements’ step will be simpler for other languages.

The ‘infer types’ step, while easy to implement for literals, will be tougher for identifiers. This is due to the fact that the type of the literals needs to be known prior. The literal is contained in a sibling node of an identifier, or a child of its sibling node. Again, once implemented for one language, this step will become less complex for other languages.

In conclusion, the complexity of adding a new language will decrease as more languages are added. Additionally, the complexity is dependent on the language to be implemented, and its similarity to languages that have already been implemented.

6.3 Analysis accuracy

The accuracy of our analysis is defined by the correctness of the binary classification of every witness discovered by Z3 being true or false. The results show the possibility of false positives and false negatives occurring. A false negative, where a witness is missed, may lead the security analyst to think the code is unreachable.

A false positive, where a false witness is given, could result in a witness that does not lead to the path of the desired line. The security analyst may be led to think that, when using the witness as input, a certain line is reached that may not be reached at all. Both false positives and false negatives could lead to security issues being overlooked.

6.4 Runtime

The runtime of conditional statements that are serial instead of nested appears to be linear, but the runtime for nested conditional statements is not as favourable. This is to be expected, since the more constraints are asserted in a solver, the more strenuous it is to find a witness. Moreover, in a nested statement, only the outermost lines have no constraints imposed upon them. In serial statements, more lines are free from constraints.

With regard to its importance, the runtime should improve on the time it takes for a security analyst to find a witness. For a single conditional statement containing one variable, the tool uses 0.13 seconds on average to compute a witness. For 256 nested conditional statements, the tool takes 40.35 seconds on average to compute a witness for every line. Not only the time difference between the computation of the tool and the security analyst is spared, but meanwhile, the security analyst is able to focus on other matters.

Chapter 7

Future work

7.1 Improving the tool

To be of use for Codean's RE, the tool needs to be able to produce a witness reliably for multiple popular imperative programming languages. The implementation of the CPP language needs to be completed.

The CPP language is a relatively simple language considering it only has 25 directives, as listed in Chapter 2.3. The expressions in the conditionals may contain...

- Integer constants
- Character constants
- Arithmetic operators
- Macros

...which means any other types can be disregarded [29].

The supported operations in this implementation are addition (+), subtraction (−), multiplication (∗) and division (/). The implemented comparisons are equal to (==), greater than (>), less than (<), greater than or equal to (>=), lesser than or equal to (<=) and not equal to (! =). Any bitwise operations, shifts, comparisons and logical operations have not been implemented to limit the scope of the paper. Additionally, there is the limit of the tool automatically assuming integers and thus not acknowledging characters. This limit is imposed not due to inability, but to reasonably limit the scope of this paper. Implementing the missing parts will eliminate some of the false positives and false negatives, thus improving the tool.

Moreover, the way to find the type of an identifier can be improved. Currently, the type is always assumed to be an integer. One can often infer the type of an identifier from the remaining part of the expression, but if this is not the case, there should be a different way to be certain of the type of an identifier. The assumption that variables are independent is another problem, which can result in false negatives.

Perhaps a solution to the issues above could be to ask the user of the tool for more information. After all, the goal of Codean's Review Environment was never to fully automate tasks. Instead, it was created with symbiotic cooperation of man and machine in mind.

The tool, in its current state, is not more helpful to the security analyst than any other symbolic executor is. For it to become a useful addition to the RE, it needs to be expanded to other languages.

Another improvement could be made on the runtime of the tool. Chapter 3.1 mentions different adaptations of symbolic execution. One of these adaptations is differential symbolic execution [53]. It reduces the cost of symbolic execution by making use of the fact that different versions of source code are largely similar. This could be useful, because it is good practice to review source code again after it is updated.

One more potentially useful adaptation of symbolic execution is directed symbolic execution [44]. Instead of finding the constraints per line of code, this modification allows finding the constraints for one line of code only. The security analyst could have the option of computing a witness for all lines of code, or only the ones they desire. An advantage of directed symbolic execution is that if the security analyst wants to have a witness for a line of code soon after importing the source code, they would not have to wait for the tool to compute all the lines. A disadvantage, however, is that the security analyst will have to wait each time they want to learn a witness for a line, instead of it being precomputed.

7.2 Evaluating the tool

Accurately evaluating the tool is a project in and of itself. Not only does it involve assessing the current implementation for its speed and accuracy, but to be certain whether a multitude of languages can be implemented, a multitude of languages actually need to be implemented.

To properly evaluate whether the current implementation of the tool, without implementing other languages, can be of use, it needs to be assessed on real use cases. The presence of false positives and false negatives is known, but also important is the frequency of them. Furthermore, when such an error does occur, how does this affect the work of the security analyst? Can intervention from the security analyst eliminate the errors? And if so, does this labour weigh up to the automation the tool provides?

Chapter 8

Conclusion

With software becoming more complex, the number of digital hackers increasing and the world becoming progressively more reliant on software, the need for secure software is at an all-time high. Security analysts need to become more effective in their task.

Codean has developed the RE, which is aimed to assist the security analyst in reviewing more code in less time. A technique that can be utilised to aid security analysts is symbolic execution. The goal in this implementation is to generate a witness for a given line, which is a set of values that satisfies all constraints imposed on the execution of that line. It is essential that this tool supports multiple languages in order to be used in the RE. It was expected that such a tool developed with Tree-sitter and Z3 results in a complete implementation of symbolic execution, that can be expanded to programming languages that are supported by Tree-sitter.

This resulted in a generic architecture for symbolic execution of conditional execution paths in programming languages, using the CPP language as a test subject. The implementation of this particular language is not yet 100% accurate, as it generates both false positives and false negatives. The difficulty of adding on another language depends on the similarity of the language to the previously implemented languages as well as the amount of previously implemented languages.

The next step forward is proper evaluation of the tool, followed by the implementation of characters and logical operators to complete the CPP language. Adding more languages to the tool will increase its reach. False positives and false negatives need to be diminished. One possible solution is allowing interaction between the tool and the security analyst. The runtime of the tool could be improved by using adaptations of symbolic execution.

Bibliography

[1] Roberto Amadini, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. Abstract interpretation, symbolic execution and constraints. In *Recent Developments in the Design and Implementation of Programming Languages*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020. 5

[2] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381. Springer, 2008. 4

[3] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. In *IEEE Transactions on Dependable and Secure Computing*, volume 1(1), pages 11–33. IEEE, 2004. 3

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018. 10, 11

[5] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: a versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022. 10

[6] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011. 10

[7] Ira D Baxter. Dms: Program transformations for practical scalable software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 48–51, 2002. 7

[8] A. Bhargav. Universities Aren't Doing Enough About Developer Security. Here's Why You Should Care, 2022. Last accessed 5 October 2022. 1

[9] Thomas Bouton, Caminha B de Oliveira, David Déharbe, Pascal Fontaine, et al. verit: an open, trustable and efficient smt-solver. In *International Conference on Automated Deduction*, pages 151–156. Springer, 2009. 10

[10] Robert S Boyer, Bernard Elspas, and Karl N Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245, 1975. 9

[11] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of computer programming*, 72(1-2):52–70, 2008. 7

[12] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009. 10

[13] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The mathsat 4 smt solver. In *International Conference on Computer Aided Verification*, pages 299–303. Springer, 2008. 10

[14] D. N. Burrell. An exploration of the cybersecurity workforce shortage. *Cyber Warfare and Terrorism: Concepts, Methodologies, Tools, and Applications*, pages 1072–1081, 2020. 1

[15] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012. 10

[16] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, number CONF, 2009. 9

[17] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol: An interpolating smt solver. In *International SPIN Workshop on Model Checking of Software*, pages 248–254. Springer, 2012. 10

[18] Lori A Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, pages 488–491, 1976. 9

[19] Timothy Clem and Patrick Thomson. Static analysis at github: An experience report. *Queue*, 19(4):42–67, 2021. 10

[20] Codean. Codean, 2022. Last accessed 5 October 2022. 1

[21] James R Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006. 7

[22] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. Smt-rat: an open source c++ toolbox for strategic and parallel smt solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 360–368. Springer, 2015. 10

[23] P. D. Coward. Symbolic execution and testing. *Information and Software Technology*, 33(1):53–64, 1991. 4, 10

[24] Simon Cruanes. *First-order reasoning in Yices2*. PhD thesis, master thesis in Computer Science, 2012. 10

[25] Giorgios Robert Economopoulos. *Generalised LR parsing algorithms*. PhD thesis, Citeseer, 2006. 10

[26] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001. 7

[27] Ericpony. Z3 Python tutorial. Last accessed 19 October 2022. 13, 14

[28] M.D. Ernst, G.J. Badros, and D. Notkin. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002. 6

[29] Inc Free Software Foundation. The C Preprocessor. Last accessed 26 October 2022. 24

[30] Inc Free Software Foundation. The C Preprocessor - Index of Directives. Last accessed 31 October 2022. 6

[31] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *International conference on computer aided verification*, pages 519–531. Springer, 2007. 10

[32] Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 166–176, 2012. 9

[33] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005. 4

[34] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012. 10

[35] William E Howden. Experiments with a symbolic evaluation system. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 899–908, 1976. 9

[36] Ying Hu, Merlo, Dagenais, and Lague. C/c++ conditional compilation analysis using symbolic execution. In *Proceedings 2000 International Conference on Software Maintenance*, pages 196–206, 2000. 9

[37] Antti EJ Hyvärinen, Matteo Marescotti, Leonardo Alt, and Natasha Sharygina. Opensmt2: An smt solver for multi-core and cloud computing. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 547–553. Springer, 2016. 10

[38] Raghudeep Kannavara, Christopher J Havlicek, Bo Chen, Mark R Tuttle, Kai Cong, Sandip Ray, and Fei Xie. Challenges and opportunities with concolic testing. In *2015 National Aerospace and Electronics Conference (NAECON)*, pages 374–378. IEEE, 2015. 9

[39] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. 1, 9, 10

[40] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177. IEEE, 2009. 7

[41] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 47(6):193–204, 2012. 4

[42] Mario Latendresse. Fast symbolic evaluation of c/c++ preprocessing using conditional values. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pages 170–179. IEEE, 2003. 10

[43] Mario Latendresse. Rewrite systems for symbolic evaluation of c-like preprocessing. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 165–173. IEEE, 2004. 10

[44] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, pages 95–111. Springer, 2011. 9, 25

[45] M. Metheny. Continuous monitoring through security automation. *Fed. Cloud Comput*, pages 453–472, 2017. 4

[46] Microsoft. Online Z3 Guide. Last accessed 24 October 2022. 18

[47] S. Morgan. Cybercrime To Cost The World $10.5 Trillion Annually By 2025, 2020. Last accessed 5 October 2022. 1

[48] L. D. Moura and N. Bjørner. Z3: An efficient SMT solver. In C.R. Ramakrishnan and J. Rehof, editors, *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, Berlin, Heidelberg, 2008. 2, 8

[49] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012. 3

[50] Aina Niemetz and Mathias Preiner. Bitwuzla at the smt-comp 2020. *arXiv preprint arXiv:2006.01621*, 2020. 10

[51] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995. 7

[52] Thomas J Pennello. Error recovery for lr parsers. Technical report, CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES, 1977. 10

[53] Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237, 2008. 9, 24

[54] K. Scarfone, M. Souppaya, A. Cody, and A. Orebaugh. Technical guide to information security testing and assessment. *NIST Special Publication, 800(115)*, 2008. 1

[55] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007. 9

[56] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016. 10

[57] SMT-COMP. 2022 results QF_Equality+LinearArith. Last accessed 22 October 2022. 20

[58] R. M. Stallman and Z. Weinberg. The c preprocessor. *Free Software Foundation*, 8, 1987. 5

[59] Bjarne Stroustrup. *The design and evolution of C++*. Pearson Education India, 1994. 6

[60] The-Crankshaft Publishing. Error Detection and Recovery in LR Parsers. Last accessed 13 October 2022. 10

[61] TIOBE. TIOBE Index for September 2022, 2022. Last accessed 5 October 2022. 1

[62] Tree-sitter. Tree-sitter, 2022. Last accessed 5 October 2022. 2, 7, 10

[63] Tree-sitter. Tree-sitter Documentation, 2022. Last accessed 17 October 2022. 13

[64] Eric R Van Wyk and August C Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 63–72, 2007. 10

[65] Verizon. Data Breach Investigations Report, 2022. Last accessed 5 October 2022. 1

[66] Tim A Wagner and Susan L Graham. Incremental analysis of real programming languages. *ACM SIGPLAN Notices*, 32(5):31–43, 1997. 10

[67] Tim A Wagner and Susan L Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(5):980–1013, 1998. 10

[68] Tim Allen Wagner. *Practical algorithms for incremental software development environments.* University of California, Berkeley, 1997. 10

Appendix A

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

```python
from z3 import *
from .parser import C_LANGUAGE, parse

source_code = "examples/stdio.c"


def solve(node):
    """ Finds the type of a node and transforms its content to a Z3 variable.

    :param node: the node of which we want the type.
    :return: a Z3 variable containing the value the node provided.
    """
    if node.type == "binary_expression":
        return solve_bin_exp(node)
    elif node.type == "identifier":
        return Int("%s" % node.text.decode('utf-8'))
    elif node.type == "number_literal":
        return IntVal(node.text.decode('utf-8'))
    else:
        raise TypeError(f"node type not supported: {node.type}")


def solve_preproc_defined(children):
    """ Handles the '#if defined' statements.
    A negation of a defined ('!defined') results in a unary expression.

    :param children: the children of the binary expression,
                     in this case one of them indicates an
                     '#if defined' statement occurred.
    :return: the constraints in the form of Z3 variables.
    """
    operator = children[1]
    identifier1 = children[0].children[1]
    identifier2  = children[2].children[1]
    if operator.type == "||":
        if children[0].type == "unary_expression" and children[2].type == "unary_expression":  # Both 'defined' statements are negated.
            lhs = Bool('defined('+identifier1.children[1].text.decode('utf-8')+')')
            rhs = Bool('defined('+identifier2.children[1].text.decode('utf-8')+')')
            return Or(Not(lhs), Not(rhs))
        elif children[0].type == "unary_expression":  # Only the first 'defined' statement is negated.
            lhs = Bool('defined('+identifier1.children[1].text.decode('utf-8')+')')
            rhs = Bool('defined('+identifier2.text.decode('utf-8')+')')
            return Or(Not(lhs), rhs)
        elif children[2].type == "unary_expression":  # Only the second 'defined' statement is negated.
            lhs = Bool('defined('+identifier1.text.decode('utf-8')+')')
            rhs = Bool('defined('+identifier2.children[1].text.decode('utf-8')+')')
            return Or(lhs, Not(rhs))
        else:  # Neither 'defined' statements are negated.
            lhs = Bool('defined('+identifier1.text.decode('utf-8')+')')
            rhs = Bool('defined('+identifier2.text.decode('utf-8')+')')
            return Or(lhs, rhs)
    elif operator.type == "&&":
        if children[0].type == "unary_expression" and children[2].type == "unary_expression":  # Both 'defined' statements are negated.
            lhs = Bool('defined('+identifier1.children[1].text.decode('utf-8')+')')
            rhs = Bool('defined('+identifier2.children[1].text.decode('utf-8')+')')
            return And(Not(lhs), Not(rhs))
        elif children[0].type == "unary_expression":  # Only the first 'defined' statement is negated.
            lhs = Bool('defined('+identifier1.children[1].text.decode('utf-8')+')')
            rhs = Bool('defined('+identifier2.text.decode('utf-8')+')')
            return And(Not(lhs), rhs)
        elif children[2].type == "unary_expression":  # Only the second 'defined' statement is negated.
            lhs = Bool('defined('+identifier1.text.decode('utf-8')+')')
            rhs = Bool('defined('+identifier2.children[1].text.decode('utf-8')+')')
            return And(lhs, Not(rhs))
        else:  # Neither 'defined' statements are negated.
            lhs = Bool('defined('+identifier1.text.decode('utf-8')+')')
            rhs = Bool('defined('+identifier2.text.decode('utf-8')+')')
            return And(lhs, rhs)
    else:
        raise TypeError(f"operator type not understood: {children[1].type}")


def solve_bin_exp(binary_expression):
    """ Extracts the constraints from the binary expression.

    :param binary_expression: the expression of which we want the constraints.
    :return: the constraints in the form of Z3 variables.
    """
    children = binary_expression.children
    assert len(children) == 3, "not 3 children"
    if children[0].type == "preproc_defined" or children[0].type == "unary_expression":  # An '#if define' statement needs to be handled differently.
        return solve_preproc_defined(children)
    lhs = solve(children[0])  # Solve the left hand side of the expression
    rhs = solve(children[2])  # Solve the right hand side of the expression
    operator = children[1].text.decode('utf-8')
    if operator == '*':
        return lhs.__mul__(rhs)
    if operator == '+':
        return lhs.__add__(rhs)
    if operator == '-':
        return lhs.__sub__(rhs)
    if operator == '/':
        return lhs.__truediv__(rhs)
    if operator == '==':
        return lhs.__eq__(rhs)
    if operator == '>':
        return lhs.__gt__(rhs)
    if operator == '<':
        return lhs.__lt__(rhs)
    if operator == '>=':
        return lhs.__ge__(rhs)
    if operator == '<=':
        return lhs.__le__(rhs)
    if operator == '!=':
        return lhs.__ne__(rhs)


def find_identifier(binary_expression):
    """ All identifiers contained in this (nested)
    binary expression are extracted and collected
    in a list.

    :param binary_expression: the expression of which the identifiers need to be found.
    :return: a list of identifiers that belong to the binary expression.
    """
    temp_identifiers = []
    children = binary_expression.children
    for child in children:
        if child.type == "identifier":
            temp_identifiers.append(child)
        elif child.type == "binary_expression" or child.type == "preproc_defined":
            temp_identifiers.extend(find_identifier(child))
    return temp_identifiers


def copy_solver(s):
    """ This function creates a new solver and
    copies the constraints from the old solver
    to the new one.

    :param s: is the solver to be copied.
    """
    s_copy = Solver()
    for assertion in s.assertions():
        s_copy.add(assertion)
    return s_copy


def check_nodes(node, s_copy, line_number):
    """ This function performs checks on the current node.
    If the current node matches a type that imposes a constraint,
    additional actions need to be taken to extract this constraint
    and add it to the solver.

    :param node: is the current node that is evaluated.
    :param s_copy: is the solver containing the current constraints.
    :param line_number: is the number of the line that is currently evaluated.
    """
    if node.type == "preproc_ifdef":  # An '#ifdef' statement is encountered.
        children = node.children
        for child in children:
            if child.type == 'identifier':
                identifier = child
        for child in children:
            if child.type == '#ifdef':
                s_copy.add(Bool('defined('+identifier.text.decode('utf-8')+')') == True)
            if child.type == '#ifndef':
                s_copy.add(Bool('defined('+identifier.text.decode('utf-8')+')') == False)

    if node.type == "preproc_if":  # An '#if' statement is encountered.
        skip = False  # The 'skip' variable indicates whether the if constraint applies.
        # Its 'endpoint' makes it so that '#else' and '#elif' statements are affected by the '#if'.
        for child in node.children:
            if child.type in ["preproc_elif", "preproc_else"] and child.start_point[0] <=
            ↪  line_number:
                skip = True
        if not skip:
            for child in node.children:
                if child.type == 'binary_expression':
                    binary_expression = child
                    constraints = solve_bin_exp(binary_expression)
                    s_copy.add(constraints)
                    identifiers = find_identifier(binary_expression)  # If a variable is used in an
                    ↪  '#if' statement, it means it cannot be undefined.
                    for identifier in identifiers:
                        s_copy.add(Bool('defined('+identifier.text.decode('utf-8')+')') == True)

    if node.type == "preproc_elif": # An '#elif' statement is encountered.
        skip = False  # The 'skip' variable indicates whether the if constraint applies.
        # Its 'endpoint' makes it so that '#else' statements are affected by the '#elif'.
        for child in node.children:
            if child.type in ["preproc_elif", "preproc_else"] and child.start_point[0] <=
            ↪  line_number:
                skip = True
        if not skip:
            for child in node.children:
                if child.type == 'binary_expression':
                    binary_expression = child
            constraints = solve_bin_exp(binary_expression)
            s_copy.add(constraints)
            parent = node.parent  # The negation of the parent is added to the constraints.
            if parent.type == "preproc_if":
                for child in parent.children:
                    if child.type == 'binary_expression':
                        binary_expression_if = child
                constraints_if = solve_bin_exp(binary_expression_if)
                s_copy.add(Not(constraints_if))
            elif parent.type == "preproc_ifdef":
                children = parent.children
                for child in children:
                    if child.type == 'identifier':
                        identifier = child
                for child in children:
                    if child.type == '#ifdef':
                        s_copy.add(Bool('defined('+identifier.text.decode('utf-8')+')') == False)
                    if child.type == '#ifndef':
                        s_copy.add(Bool('defined('+identifier.text.decode('utf-8')+')') == True)
            identifiers = find_identifier(binary_expression)  # If a variable is used in an '#elif'
            ↪  statement, it means it cannot be undefined.
            for identifier in identifiers:
                s_copy.add(Bool('defined('+identifier.text.decode('utf-8')+')') == True)

    if node.type == "preproc_else": # An '#else' statement is encountered.
        parent = node.parent  # The negation of the parent is added to the constraints.
        if parent.type == "preproc_if":
            binary_expression_if = parent.children[1]
            sat_if = solve_bin_exp(binary_expression_if)
            s_copy.add(Not(sat_if))
        elif parent.type == "preproc_ifdef":
            identifier = None
            children = parent.children
            for child in children:
                if child.type == 'identifier':
                    identifier = child
            for child in children:
                if child.type == '#ifdef':
                    s_copy.add(Bool('defined('+identifier.text.decode('utf-8')+')') == False)
                if child.type == '#ifndef':
                    s_copy.add(Bool('defined('+identifier.text.decode('utf-8')+')') == True)


def dfs(node, s, line_number):
    """ Perform depth-first search on the generated tree.
    It starts with printing the lines for which all constraints have been found.
    Next it checks whether the current node imposes a constraint.
    Finally it moves to the next node in line.

    :param node: represents the current node.
    :param s: is the solver that contains the current constraints.
    :param line_number: is the number of the line that is currently evaluated.
    """
    s_copy = copy_solver(s)  # This copy makes sure that the solver of the parent node is not
    ↪  overwritten.

    while node.start_point[0] > line_number: # The current node is further than the line we last
    ↪  printed, so we need to print more lines.
        line_number += 1
        if s_copy.check() == z3.sat:
            print(line_number+1, s_copy.model())
        else:
            print(line_number+1, "line can never be reached")

    check_nodes(node, s_copy, line_number)

    for child in node.children:  # Move to the next node.
        if child.type == "preproc_else" or child.type == "#endif" or child.type == "preproc_elif":
        ↪  # Do not use the constraints that their parent got.
            line_number = dfs(child, s, line_number)
        else:
            line_number = dfs(child, s_copy, line_number)
    return line_number


def solve_per_line(node):
    """ Creates a solver and initiates the depth-first search.

    :param node: represents the root node of the tree.
    """
    s = Solver()
    STARTING_LINE_NUMBER = -1
    dfs(node, s, STARTING_LINE_NUMBER)


def print_source_code():
    """ Prints all the lines of the source code. """
    file = open(source_code, "r")
    file_lines = file.readlines()
    for i in range(len(file_lines)):
        print(f"line {i+1} ",file_lines[i].strip("\n"))


def main():
    tree = parse(source_code, C_LANGUAGE)
    print_source_code()
    solve_per_line(tree.root_node)


if __name__ == "__main__":
    main()
```