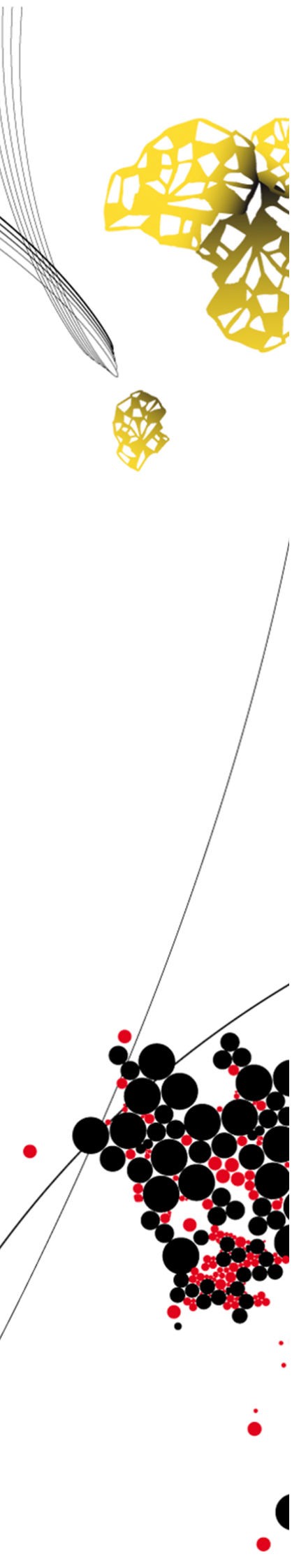


UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Incremental Symbolic Execution

Joran J. Honig

M.Sc. Thesis
June 2020

Supervisors:
prof. dr. M. Huisman
dr. M. H. Everts

Telecommunication Engineering Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Symbolic execution is a popular analysis technique used for finding bugs in Ethereum smart contracts. However, symbolic execution is computationally expensive. Furthermore, during the development of smart contracts, analysis is started from scratch for each new version of the software, recomputing many redundant results. Many approaches exist for the optimisation of symbolic execution, one of which is the use of symbolic summaries. In this thesis, we design a technique which efficiently permits the re-use of symbolic summaries between analyses, allowing for incremental symbolic execution for smart contracts. In particular, the technique aims to permit the re-use of summaries for code with syntactic changes.

First, we analyse the changes which occur in smart contracts for the design and evaluation of the summary checking approach. We formulate a set of three algorithms that use program normalisation and dataflow analysis to deal with the identified change types. We evaluate the performance of our summary checking approach through three benchmarks, focussing on particular change types, real-world scenarios, and compiler introduced changes.

The results show that this technique can be applied effectively in real-world scenarios, allowing for the re-use of, on average, 85% of symbolic summaries. Furthermore, the methods are particularly effective for program changes resulting from changes in the compiler, reaching a summary re-use rate of 100%. Finally, in our experiments, summary validation requires an order of magnitude less time than the re-generation of the summaries which remain valid between program versions.

In conclusion, the proposed normalisation based summary checking approach is an effective method for incremental symbolic execution by allowing the re-use of symbolic summaries.

Contents

Chapter 1

Introduction

Symbolic execution is a versatile program analysis technique that is computationally expensive. In this thesis, we propose a novel approach for the *must-summary checking problem* [1] that allows for incremental symbolic execution. The approach aims to efficiently enable the re-use of must-summaries between the analyses of two versions of a program. Enabling such incremental symbolic execution by allowing the re-use of must-summaries between the analysis of two versions of a program, has the potential to provide improvements to the scalability and real-world performance of symbolic execution based tools. In this chapter, we motivate the merits of such an approach by demonstrating that such an optimisation can be leveraged to assist the mitigation of security risks for smart contracts on the Ethereum blockchain [2].

Blockchain platforms like Ethereum [2], provide a platform that supports the execution of programs, called smart contracts. Unlike with regular programs, that a server or personal computer executes, it is the participants of the Ethereum network that execute smart contracts. Because they run on Ethereum blockchain, smart contracts gain properties like censorship resistance, immutability and verified execution [2].

These properties are attractive for applications that require a high level of security. However, the open Ethereum blockchain also makes for a high-risk environment. Firstly, smart contracts deployed on the Ethereum blockchain are visible and accessible to all the participants in the Ethereum network. Additionally, smart contracts are immutable; once deployed to the blockchain, they cannot be changed anymore. These aspects create a high stakes environment where smart contract developers have to be diligent in ensuring the correctness and security of their smart contracts. Unfortunately, there have been several cases where adversarial Ethereum users still managed to exploit a bug in a deployed smart contract; take, for example, The DAO hack [3], the Parity wallet hack [4] and the batchOverflow bug [5].

To help developers prevent such incidents from happening the Ethereum and academic communities are investing much effort into implementing and designing

different formal methods to reduce the risk of another security incident happening [6]–[9].

Mythril [6] is one of the tools implemented with this purpose. It is a tool that leverages symbolic execution [10], [11] to find bugs in smart contract systems. This tool allows developers to analyse smart contracts and find a wide range of potential vulnerabilities in their smart contracts. Examples of the bugs that can be detected using Mythril include integer overflows [12] and unprotected fund extractions [13]. Additionally, Mythril does not require any input from the user other than the contract that needs to be analysed, making the tool usable for a large part of the development community.

The primary technique used by Mythril is symbolic execution [10], [11], a versatile program analysis approach that finds uses in both autonomous analysis systems and user-aided verification. These uses cover program analysis problems like bug finding, property checking and automatic test case generation (see Section 2.1.2). At its base, symbolic execution is a program analysis technique that tries to explore all behaviours of a program, while determining what inputs lead to those specific program-behaviours. It does so by executing a program using so-called symbolic input variables, rather than concrete values. There are several benefits to this approach. Firstly, the exploration of the program-behaviours in symbolic execution does not require any input from the user. This trait is not shared by various other analysis approaches, that often require input in the shape of invariants or lemmas. As a result, symbolic executors are relatively easy to apply to software projects without requiring an understanding of formal methods. Secondly, while symbolic execution does not require aid from the user, it is still able to provide precise analysis results. This level of precision is not provided by various other autonomous analysis techniques that use abstraction to approximate all the behaviours of a program such as abstract interpretation [14]. The high precision of symbolic execution is crucial for a bug finding application, as it is essential to have a low false-positive rate when reporting bugs to developers [15]. False positives can both distract and delay developers in the triaging process; a high false-positive rate might even cause developers to ignore some of the analysis results.

Even though symbolic execution has clear benefits, there are some challenges to its development and use. One of the most prevalent problems is called “state explosion” [10], it results from the trait that many non-trivial programs have a near-infinite amount of possible program paths. Such situations can, for example, occur for programs that include loops over dynamically sized inputs. These programs will have a path for each possible size of the input variable. Moreover, the number of program paths grows exponentially for each of these loops. As a result, a program analysis approach that tries to enumerate all of those paths is not able to terminate

within a reasonable time frame.

Furthermore, symbolic execution relies heavily on SMT solvers to check the reachability for all the different explored paths. SMT solving is often computationally expensive and checking reachability for the different paths takes up a large part of the symbolic execution process [10].

Many approaches have been proposed to address these challenges, including several that aim to re-use partial results throughout the analysis [1], [16]–[18]. One such optimisation is called composite analysis [19], an approach that relieves both SMT solver costs and state explosion. Compositionally approaching the analysis of a program allows the analysis to explore each of the different functions in the target program just once. Each time the analysis reaches an unexplored function, it will analyse and explore that function and create a summary. On each subsequent call to the function, the analysis can use the summary rather than exploring the function again. Additionally, by applying the summary of the function, rather than determining all possible paths through it, the analysis can limit the effects of state explosion.

1.1 Symbolic Summary Re-use

Re-using partial analysis results within one analysis effort can be extended to the re-use of analysis results between different analysis campaigns. The prime observation behind incremental analysis and symbolic summary re-use is the following: “Between the analysis of two versions of a program; many of the computations are redundant.” Optimising the analysis process to leverage these redundancies, rather than exhausting computational resources on redundant computations, allows an analysis tool to both conserve effort and speed up the generation of analysis results.

Such incremental analysis approaches provide many benefits in a situation where different versions of a program continually need analysis. Two common use-cases we identify are:

1. A continuous integration (CI) pipeline, where a program is analysed for each newly added feature or bugfix
2. An IDE which continually provides the user with hints and feedback on their code

For the first use-case, developers might set some time bound on the analysis, re-using analysis results will allow the analysis to cover more of the program behaviours within the set bound. For the second use-case, the use of incremental analysis approaches makes it possible to provide the same results within a smaller timeframe, something which improves the usability of the analysis approaches [20].

In this thesis, We specifically consider symbolic summaries as partial analysis results that can be re-used to prevent redundant computations between analysis runs. Given two versions of a program, those functions that have not changed will generate equivalent summaries. A lightweight approach to show unchanged parts of the program between two program versions allows for the exploitation of this property. Godefroid et al. formalised this problem as the *must-summary checking problem* [1].

In order to enable incremental symbolic execution, Godefroid et al. [1] propose three algorithms to solve this *must-summary checking problem*. Furthermore, there exist a range of approaches aimed at the re-use of partial analysis results in symbolic execution [16]–[18], [21]–[23]. Many of these approaches leverage syntactic equivalence to discover which partial analysis results can be re-used. However, syntactic equivalence checks are limited in that they do not permit the re-use of analysis results for code where there are syntactic changes without an effect on the partial analysis results.

In this thesis, we provide a categorisation of these syntactic program changes that do not affect the semantics of a program. Moreover, we propose an approach that improves the current state-of-the-art by allowing the re-use of partial analysis results for code with such changes. We implement the approach to check must-summaries for Ethereum smart contracts, enabling efficient incremental bug finding. Lastly, we aggregate three benchmarks that evaluate the performance of must-summary algorithms for EVM smart contracts [2].

1.2 Method

In this thesis, we discuss the application of must-summary re-use for the analysis of smart contracts. Furthermore, we propose novel algorithms that improve upon the performance of current state-of-the-art in must-summary checking and incremental analysis.

For the implementation and evaluation of the proposed approaches, we leverage Mythril [6], a popular symbolic executor and bug finder for the Ethereum Blockchain that targets EVM bytecode [2]. Currently, Mythril does not support the generation and use of symbolic summaries. In the execution of this research, we have extended Mythril with the support for symbolic summaries, using a plugin for its symbolic execution engine. Furthermore, we leverage analysis-capabilities based on the abstract interpretation [14] of EVM bytecode [8]. This provides the required capabilities to perform the operations we propose in Chapter 5. We identify and categorise common change types introduced between smart contract versions in Chapter 3. Lastly, we propose the summary checking algorithms in Chapter 5 and evaluate their performance using the benchmarks proposed in Section 6. These benchmarks

evaluate the summary checking algorithms both on real-world performance and on the performance for arbitrary program changes.

1.2.1 Research Question

How can we efficiently check must-summaries for EVM bytecode [2] smart contracts[1]that have semantically preserving changes in the summarised code?

Subquestions:

1. Which different origins introduce program changes in smart contracts, and how do they affect the type of program change.
2. Which types of program changes are identifiable in smart contracts relating to the summary checking problem?
3. How can we efficiently check equivalence for the different types of program changes with respect to the summary checking problem?

Chapter 2

Background

An approach to incremental symbolic execution is likely to leverage a wide range of program analysis techniques and theories. This chapter contains a description of several program analysis techniques in order to introduce the reader to the topics discussed in this thesis.

2.1 Symbolic Execution

Symbolic execution [10], [11] is a program analysis technique central to this thesis. It is a technique that strikes a balance between dynamic and static analysis approaches. Like dynamic analysis techniques, it can create concrete counterexamples to disprove program properties. Like static analysis techniques, it provides semantic insight into the program.

The general approach of symbolic execution is to try to explore all possible paths through a program. A path through the program is a sequence of consecutive instructions starting from the entry of a program, continuing to an exit point of the program. For each path w, the analysis maintains a path constraint ϕ_w. This path constraint is the condition for the execution to take the path w through the program. Additionally, the analysis computes a symbolic state Σ for all the steps in the path. This state stores the expressions for each memory location and the path constraint until that point. During the exploration of the program, one can leverage this information for a variety of purposes.

One of the uses is property checking, where for each reachable program state, an analysis verifies that some properties hold. Automatic test case generation is another use of symbolic execution; The path constraints ϕ_w can be used to create concrete inputs that will cover each distinct program path covered by the symbolic execution. Lastly, symbolic execution is applicable in the area of bug finding. The use of symbolic execution for bug finding is similar to property checking, but with

generic pre-defined properties that imply the existence of bugs like buffer overflows.

One factor that has inhibited the mainstream adoption of symbolic execution is the scalability of the approach. A popular research topic for scalability has been the path explosion problem. Another factor that inhibits the scalability of symbolic execution is the computational cost associated with exploring a program path. It involves the computation of the program states, path constraint and satisfiability of the path constraint.

2.1.1 Key Concepts

In this subsection, we will iterate the core concepts of symbolic execution; these concepts will be demonstrated with a guiding example in Subsection 2.1.2.

Symbolic execution, as opposed to concrete execution, executes a program with symbolic values. A symbolic value is an algorithmic variable that can represent all values that a type can take. The analysis starts with an initial state Σ_{init} and path condition ϕ_w. Symbolic variables are used to represent all inputs in this initial state. The path condition for this initial state is $True$. Execution of the program is similar to concrete execution. In concrete execution, each program statement is represented by some function f that implements some behaviour in the concrete domain. For symbolic execution, it is possible to formulate a function f' which implements the same behaviour in the symbolic domain.

The state after the execution of this statement is defined as $\Sigma = f'(\Sigma_{init})$, where Σ is the result after the application of f on the initial state. The execution continues by continually computing successor states.

The analysis follows with this process until it reaches some branching statement. A branching statement has some conditional value, which determines which program branch to take. In concrete execution, this value is available, and the executor will follow the corresponding path. In symbolic execution, the branch condition can be symbolic, in which case, both the true and false case of the condition could be possible. The symbolic executor will, therefore, follow both branches, and store this branch condition as part of the path condition ϕ_w.

2.1.2 Guiding Example

This section will demonstrate the introduced concepts using an example. Figure 2.1 contains a simple function in the Solidity programming language, which we will use as a guiding example.

Figure 2.2a shows the control flow graph for the program. The numbers in the nodes correspond to the line numbers in the code, and the arrows indicate tran-

sitions. There are two possible paths (a sequence of execution steps through a program) in this function. One of the paths enters the if statement at line 4, the other continues execution at line 6. These paths are visualized in Figure 2.2b and Figure 2.2c.

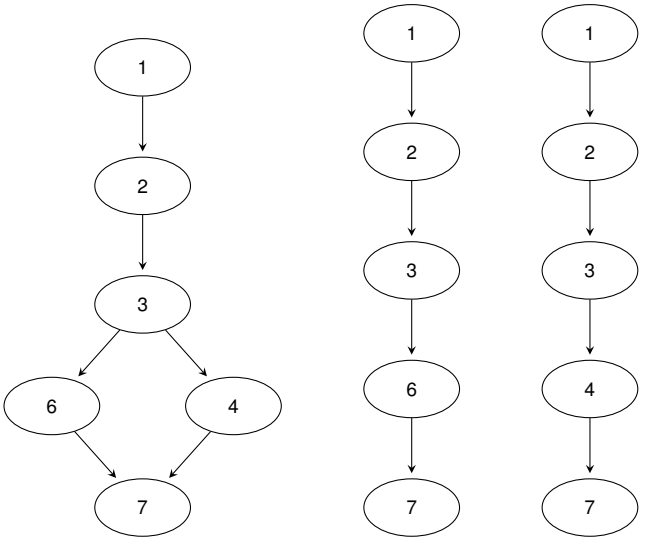
Figure 2.2d describes the symbolic state space of a program. This figure shows a graph of all the symbolic states discovered during symbolic execution. There are three types of elements in the graph: states, state transitions and branch conditions. Nodes and edges, respectively represent the states and state transitions. The branch conditions are shown as guards at the edges.

The first node and symbolic state represent the initial symbolic state Σ_{init}. At this state in the execution, there are no initialized variables yet. This happens with the execution of the next statement, which defines the variable result.

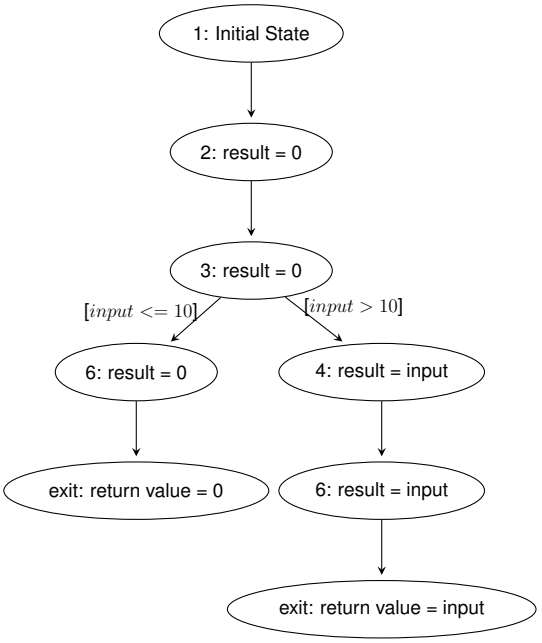
Branch conditions are signified using edge guards. They specify the condition for a specific branch to be taken. A path that follows a specific branch needs to satisfy the branch condition. This imposes constraints on the possible values that the variables in the symbolic states can take further along the path. The condition for a specific path to be taken is calculated by aggregating the branch conditions along that path. We formally say that ϕ_w is the path condition for path w and the conjunction of branch conditions of the branches on w.

```
1    function execute(uint256 input) public returns (uint256) {
2      uint result = 0;
3      if (input > 10) {
4        result = input;
5      }
6      return result;
7    }
```

Figure 2.1: Symbolic Execution Guiding Example



(a) Control Flow Graph *(b)* Path 1 *(c)* Path 2



(d) Symbolic state space

Figure 2.2: Models of the guiding example in Figure 2.1

Execution steps

In the next part of this subsection, we will go through all of the specific states, to describe how the program statements affect the symbolic states.

The first state is the initial state of the program at the entry of the function. At this point, there are not any initialized variables or path constraints.

The first statement after entry into the program is “uint memory result = 0;”. This

statement sets a variable in memory to the concrete value 0. The symbolic state in Figure 2.2d at number 2 shows the state after the execution of this statement.

The statement at line three is a branching statement; it compares $input > 10$ and then branches according to the result of this comparison. In this case, the $input > 10$ can be both true and false, as input can have values like 1 or 20. Therefore both branches are followed. The analysis also records the condition $input > 10$ for the branch that goes to line 4, and $input <= 10$ for the branch that immediately continues to line 7.

In the explanation of the example, we will first continue with the path that does not enter the if statement. The next state in this path is the return statement at line 6; this returns the value of the variable result, which is the concrete integer 0. This statement is also the exit point of the function and the end of this path.

Here we continue with the explanation of the path that does satisfy the branch condition. This path does enter the if statement, and executes “result = input;”. This statement sets the value for $result$ to the symbolic value of $input$. Note that the value of $result$ is not unconstrained. The path condition, which is the conjunction of the different branch conditions along that path, is $input > 10$. Therefore, the variable $result$ is also constrained to have a value higher than 10.

Since this is the only statement in the if body, execution continues to line 6, where the return statement is reached. Here the value of result is returned, which is “input”.

Property checking

This section demonstrates how symbolic execution can be used to check the validity of a property for the available example. Consider the property “the return value of the function *execute()* is always 0”, which we will check for the function *execute()* in Figure 2.1. Formalizing the example property “the return value of the function *execute()* is always 0” as a logical formula results in the following:

$$returnvalue == 0$$

Here $returnvalue$ represents the return value of the function *execute()*.

Proving that a property P always holds can be demonstrated by showing that there is no satisfying solution for $\phi_w \wedge \neg P$ for each of the relevant states. This logical formula represents the following intuition: “Given the conditions for reaching this state, it is not possible to violate the property”.
In this example, the property P is defined as $returnvalue == 0$; thus, we need to show that there is no state for which $\phi_w \wedge returnvalue! = 0$ has a satisfying solution.

Figure 2.2d shows that there are two possible symbolic states for the exit point of the function *execute()*. The first node has $returnvalue = 0$. In this case the condition

that needs to be checked is $input \leq 0 \wedge returnvalue = 0 \wedge returnvalue \neq 0$. The condition contains a trivial contradiction and is not satisfiable; thus, the property holds in this state.

The second node has $returnvalue = input$. For this case this formula looks like $input > 10 \wedge returnvalue = input \wedge returnvalue \neq 0$. An off-the-shelf SMT solver, like Z3 [24], can be used to show that this is in fact, satisfiable. One possible satisfying solution that could be generated by such an SMT solver is $input = 11$. Since we can show that the property does not hold for this exit state, we can conclude that the property does not hold for the function.

In conclusion, the symbolic execution allowed for the iteration of program states to check the satisfiability of property violations. Moreover, the semantic insight provided by symbolic execution allowed for the generation of a concrete input that demonstrates how the property is violated.

Test case generation

Symbolic execution can be used to generate concrete test cases for a program. A basic approach to test case generation is to generate one concrete input for each path discovered during the symbolic execution. By iterating the leaf nodes of the symbolic state space, and finding a satisfying solution to the path condition ϕ_w for each of those nodes, one can find concrete inputs that cover all paths in the state-space.

The symbolic state space in figure 2.2d, shows two leaf nodes. For these symbolic states, we find the path conditions $input <= 10$ and $input > 10$. Similar to the previous approach, we can use an off-the-shelf SMT-solver like Z3 [24], to find a satisfying solution for both of the path conditions.

In this case, such a solver might output $input == 5$ and $input == 11$ respectively. These two concrete inputs can now be used to extend a concrete test suite to cover the possible paths of *execute()*.

Bug finding

Another use of symbolic execution is bug finding. The process of using symbolic execution to find bugs in a program is similar to the approach of verifying properties.

In property checking, there is a property P. By showing that P is not violated we show that the function or program is correct. Showing violation of P for some state demonstrates the incorrectness of the function or program.

Consider a bug finding use case, where there is a condition Q. If Q does not hold at some point in the program, then this indicates the existence of a vulnerability.

Different from property checking, the absence of violations of Q does often not imply correctness of the program since there are likely bugs that Q does not identify.

The approach to finding violations to the condition Q is equal to the process for verifying a property P.

When symbolic execution is applied for property checking users commonly provide the property P that is to be checked. Bug finding tools, on the other hand, often come packaged with general conditions Q that find common bugs in software.

2.2 Symbolic Summaries

Chapter 1 provided a brief introduction into composite analysis techniques and symbolic summary re-use. The use of summaries was introduced by Godefroid [19], to improve dynamic test case generation [25]. This section provides an in-depth description of symbolic summaries and demonstrates the summarisation concept using the guiding example from Section 2.1.2.

2.2.1 Introduction

During symbolic execution, an analysis might cover some sections of code multiple times. Multiple calls to a single function throughout the code is a clear example of this event; similarly, program loops are another excellent example of this phenomenon. Within normal symbolic execution, such pieces of code are analysed multiple times for each different call or entry. A compositional approach to symbolic execution aims to decrease the redundancy of the analysis by re-using previous analysis results.

The analysis achieves this by storing symbolic summaries for each part of the already covered code. Each time the analysis encounters a previously encountered section of code, the analysis can re-use the respective previously computed summary of that section, instead of re-computing the required analysis results.

In addition to preventing redundant analysis, compositionality decreases the effect of the path explosion problem. In non-compositional approaches, each reachable path in the callee would result in a distinct path in the symbolic state space. Whereas in compositional approaches, the analysis applies a summary to the calling symbolic state similar to the application of a regular program statement; thus, the analysis creates just one successor state. Note, that while this approach reduces the number of resulting states, the complexity of the expressions in the successor state is relatively more complex.

2.2.2 Formalisation

In Section 2.1 we denoted the path condition for a given path w as ϕ_w. The path condition represents pre_w, the precondition that needs to hold for w to be executed. A summary for the path w can be formulated using the path condition and resulting symbolic state Σ. Specifically, a postcondition w_{post} holding over Σ can describe the effects of the execution of w. A conjunction of pre_w and $post_w$ describes the symbolic summary for the path w.

Formally, we describe a formula of the form $\phi_w = pre_w \wedge post_w$, where pre_w denotes the path condition and $post_w$ is a conjunction of constraints on the memory state after w has been executed [19]. Given the summaries for the paths w in a function f we can also formulate the function summary ϕ_f, as a disjunction of the path summaries ϕ_{w_f}, where ϕ_{w_f} describes the path summary for a path w in f [19].

For summary checking, we consider the must-summary notation $< lp, P, lq, Q >$ proposed by Godefroid et al. [1]. lp and lq are arbitrary locations in the program and represent the entry and exit point of the summary, respectively. P is the summary precondition holding in lp, and Q is the summary postcondition holding in lq; P and Q reflect to pre_w and $post_w$ respectively. A summary of this form specifies that if the program executes the statement at lp and the precondition P holds, then eventually lq is reached where the postcondition Q holds.

Note that this formal notation of must-summaries can represent the path summaries of the form $\phi_{w_f} = pre_{w_f} \wedge post_{w_f}$. Therefore, a set of must-summaries for different paths of the form $< lp, P, lq, Q >$ can be used to describe a function summary.

2.2.3 Guiding Example

Let us consider the example function in Figure 2.1, and the corresponding symbolic state space in Figure 2.2d. The symbolic state-space contains two program paths for the function execute. Since the paths are contained within and fully describe the function, they can also be seen as describing the partitions of the function *execute()*.

One might formulate symbolic summaries for this symbolic state space, and function in the following way. There are two program paths in the function that share the same entry and exit points. Therefore the symbolic summaries will be of the form: $<$execute entry, P, execute exit, $Q>$. Note that we can derive the postcondition Q from the symbolic state Σ.

Using this information, we formulate the following summaries:

- $<$execute entry, $input > 10$, execute exit, $result = 0 \wedge returnvalue = result >$

- <execute entry, $input <= 10$, execute exit, $result = input \wedge returnvalue = result$>

The combination of the two summaries constitutes the function summary for the function *execute()*. Each time that the function *execute()* is called during the analysis, instead of entering and executing the function, the analysis can apply function summary $\phi_{execute} = (input > 10 \wedge result = 0 \wedge returnvalue = result) \vee (input <= 10 \wedge result = input \wedge returnvalue = result)$.

2.2.4 Must-summary checking problem

Compositionally approaching symbolic execution has two significant benefits: a reduction of redundant computations, and decreased effects of path explosion. The former is an aspect that is also applicable to incremental analysis. Consider the execution of a composite symbolic executor on two versions of a program. Assuming that the changes affect just small parts of the program as a whole, then it is possible to re-use many of the summaries between those two program versions. A lightweight approach that would allow the analysis to re-use valid summaries from previous executions avoids unnecessary re-computation of symbolic summaries.

Godefroid et al. [1] provide a formalisation of this problem:

"Given a set S of symbolic summaries for a program Prog and a new version Prog' of Prog, which summaries in S are still valid must-summaries for Prog'?"

Previous work by Godefroid et al. [1] provides light-weight algorithms that allow re-use of symbolic summaries for a limited set of program changes (see Section 4.2).

In Chapter 5, we propose a set of algorithms that aim to enable the re-use of symbolic summaries for an broader spectrum of program changes.

Chapter 3

Program Changes

Between the versions of a program, one can identify many classes of program changes. Furthermore, a variety of actors and causes can affect changes between different program versions. In this section, we provide an overview of different change origins and a categorisation of changes that pose different challenges to the *must-summary checking problem* [1].

We leverage the identified change categories to inform the design of the proposed must-summary checking algorithms (see Chapter 5). Furthermore, to enable the evaluation of the approaches proposed in Chapter 5 we formulate a set of benchmarks (see Chapter 6) that aim to represent the program changes from the change categories identified in this chapter.

3.1 Change Origins

It is possible to identify multiple origins that can introduce changes between two versions of a program. First of all, programmers can introduce changes in a program. These are often changes that occur at the source code level. However, program analysis does not necessarily operate at this level. Some tools analyse programs written in some high-level language like Solidity or C, whereas others analyse lower-level languages like the EVM [2] and x86 instruction sets. The programs written in a higher-level language are often compiled to lower-level languages.

In this thesis, we propose algorithms for summary checking at the EVM level. Therefore, for the categorisation of changes and their origins, it is necessary to consider both the changes made by the developer at the source code level (see Section 3.1.3). In addition to the changes that can result from the compilation to EVM bytecode (see Section 3.1.1 and Section 3.1.2).

3.1.1 Compiler Passes

Many modern compilers enable the application of different optimisations. Depending on situational factors, a program will be compiled with different compiler passes enabled. For example, during the development phase, a developer might be inclined to disable thorough compiler optimisations which take more time to finish. This allows for a smoother incremental development process. It also means that between different analysis runs of increments of the program, there are changes introduced by enabled compiler passes.

Some examples of changes that different compiler passes can introduce are stack canaries [26], dead code removal [27], constant folding [27] and memory layout optimisation [28]. In general, we can divide these changes into two categories:

- Semantically preserving
- Semantically changing

Semantically preserving

The compilation passes that apply optimisations are generally semantically preserving under the assumption that the source language is typesafe. This property is imperative when considering the summary validation problem, as it implies that for a given program p mutated by some semantically preserving compiler pass, it is possible to re-use all previously found (partial) analysis results.

Semantically changing

Contrary to the previous category, semantically changing passes, as the name implies, do not necessarily preserve semantic equivalence.

Recall one of the previously mentioned compiler passes which introduce stack canaries or stack guards [26]. This is a compiler pass that introduces some additional checks throughout the code that check the integrity of the stack. This pass does not assume type safeness; rather, it is a pass solely introduced because type safety can be violated. Moreover, since the change introduces new behaviour in the program, it is not necessarily possible to re-use previously computed analysis results.

3.1.2 Compiler Versions

Similar to how enabling different compiler passes can cause changes in the final program, different compiler versions can also introduce changes. Different versions of

a compiler should produce semantically equivalent programs. There are two cases where this assumption does not hold. Firstly, a compiler can include a new semantically changing pass, changes caused by this will be equivalent to the changes discussed in Section 3.1.1. Secondly, some compiler versions may include unintended behaviour, or bugs, which result in semantically divergent compilation results.

We performed a study of changes that can be introduced between the recent versions of solc [29] (the compiler for the Solidity programming language), to identify the changes that a must-summary checking algorithm for EVM bytecode might encounter. Specifically, we looked at the versions of solc between 0.5.0 and 0.5.9. The scope of this analysis is limited to solc, and compilers might exhibit different behaviour.

The following subsections provide an overview and discussion of the most prevalent changes that we identified.

New operator

The Constantinople hardfork [30] (an update to the Ethereum blockchain) introduced changes in the Ethereum virtual machine. Among these changes is the addition of new instructions for the EVM. Between versions 0.5.4 and 0.5.5 solc introduced the application of these new EVM instructions.

Dispatch function restructuring

The first four bytes of the calldata in a transaction to a smart contract are used to identify the function that the sender wants to execute. The Solidity compiler implements dispatching logic that directs control flow to the function entry point; we discovered changes in this dispatch logic introduced by different versions of the solc compiler.

Improved optimisation passes

The solc compiler implements dead code analysis and stack layout optimisation. We observe that newer versions of the solc compiler can designate more sections as dead code, allowing them to minimise the smart contracts more.

3.1.3 Developer introduced changes

Developers can introduce a range of changes that influence the verification of summaries to a different extent. We identify three main aims that a developer wants to achieve when introducing changes in their software:

- Feature Addition or Removal
- Bug Fixing
- Software Refactoring

Feature addition, removal and bug fixes are cases where the developer introduce meaningful changes in the program. Such changes inhibit the ability of incremental analysis techniques to re-use partial analysis results for the changed code.

Software refactoring [31] is different, as the purpose of a refactor is to change the code while preserving the existing functionality and semantics. As a result, changes in this last category should permit the re-use of partial analysis results.

3.2 Change categories

In this section, we look at the different situations that program changes can create and how they affect summary re-validation. Note that this is not an exhaustive categorisation of program changes. Future work can extend upon this categorisation with the addition of specific categories. Such refinements can permit optimisation of the must-summary checking algorithms for these specific cases.

For each summary S, defined as a quadruple $< lp, P, lq, Q >$ (see Chapter 2.2), we define the set of all possible traces between lp and lq as T. Additionally, we specify a single trace $T' \epsilon T$ as the specific path taken through the program given the summary conditions. As shown in Section 4.2, demonstrating conditional equivalence is sufficient to verify a previously valid summary S. Therefore a summary can be proven valid for a new version of a program if one can prove that there are not semantically relevant changes in T'.

3.2.1 No change to dependent basic blocks

We say that the set of basic blocks that are executed in T' are B'. Similarly, the set of basic blocks that can be executed by the traces T are B. This category describes all changes that do not affect the basic blocks in B'. For these cases, syntactic equivalence of the basic blocks in B' demonstrates conditional equivalence. Showing syntactic equivalence of all basic blocks in B is sufficient, as $B' \subseteq B$.

3.2.2 Syntactic change to basic block

For changes in this category, we say that at least one basic block BB that gets executed in T' has some syntactic change. We also confine the change to preserve partial equivalence of the basic block BB.

We identify the following examples of changes in this category:

- stack reordering
- arithmetic operation change
- changes to dead code [1]

3.2.3 Semantically equivalent change to basic blocks

Similar to the previous category, we say that at least one basic block BB in the trace T' has some syntactic change. Different from the previous category, the change does not result in partial equivalence of BB. There is a subset of consecutive blocks in T' BT that includes BB, for which we can show partial equivalence.

We identify the following concrete cases:

- stack reordering
- parameter order change
- changes to dead code

3.2.4 Effectless semantic changes

For this category, we consider changes in the basic blocks of T, that introduce a new behaviour in the program, but not in the result of T'.

Take the following example

```
1 fn example(input) {
2     i = input * 2;
3     if ( input < 0 ) {
4         return 0;
5     }
6     return i;
7 }
```

Removing the first line of the function will not have an effect on a trace that originally executed lines 1, 2, 3 and 4. Note that this category is in actuality a special case of the previous category. We identify this as a separate category because an algorithm can potentially optimise to treat such semantic changes efficiently.

[1] Statements are dead whenever they do not affect the execution of the program

3.2.5 Basic block structure changes

The previous categories identify program changes that do not modify the basic block structure of a program. This category describes the range of changes that do not introduce semantic changes in a program, but that do change this basic block structure. An example of a change in this category is partial-loop unrolling [32], a compiler optimisation technique.

3.2.6 Semantic changes

In this category, we consider all changes that disallow the re-use of symbolic summaries. However, we identify a series of special cases where it is still possible to leverage previously computed symbolic summaries.

These cases allow the executor to optimise its interaction with the SMT solver by leveraging the assumption that the precondition P of S is satisfiable. Such an approach could extend existing constraint caching approaches [33], [34].

Stronger constraints

The first sub-category is that of changes for which there is a valid summary S' for the new program, where S' has a stronger precondition but is otherwise equivalent to S. In this case, the symbolic executor can be allowed to assume that the original precondition P is satisfiable.

Weaker constraints

Similar to the previous sub-category, this category defines changes that induce a change in summary preconditions. In particular, this category describes those changes for which there is a valid summary S' for the new program, where S' has a weaker precondition but is otherwise equivalent to S.

Changed effects

For this category, we consider the changes for which there is a valid summary S' for the new program, where S' has a changed post-condition compared to S. Changes to the effects of a summary can coincide with the two change types mentioned above.

Chapter 4

Related Work

In this section, we provide an overview of current state-of-the-art in incremental and differential analysis techniques. Furthermore, we provide an extensive discussion on symbolic summary re-use as one of the specific approaches taken in incremental program analysis.

4.1 Incremental and Differential Analysis Techniques

During the software development life cycle, a program undergoes many changes. For each addition to, or refactor of, the program, the developer ponders two questions:
(1) Did the change introduce any unwanted behaviour or remove desired behaviour (regression testing)?
(2) Did the change introduce the desired behaviour?
Incremental and differential analysis techniques enable optimisations or provide an answer to these questions.

This section provides an overview of incremental and differential analysis techniques. The first category that we discuss are differential program analysis techniques [35]–[38] (see Section 4.1.1), which focus on the discovery and characterisation of changes between two versions of a program.

The second category is that of incremental analysis techniques, which leverage information on program changes to direct and speed up future analyses (see Section 4.1.2). In this research area we identify two main approaches.

Firstly, instead of focusing the analysis on the entire program, one can focus the analysis only on those parts of the program that might be influenced by the program changes, leaving old and already analysed program behaviours alone [21]–[23].

The second approach is to re-use parts of the previously computed analysis results to reduce redundant computation [1], [16], [18], [20], [39], [40].

4.1.1 Differential program analysis

This subsection provides an overview of the work in differential analysis. Additionally we reflect on the possible application of these techniques and approaches to the *must-summary checking problem*.

The first two techniques [36], [37] that we discuss, introduce methods for change characterisation. Change characterisation intends to increase developer understanding of changes, giving more information than the binary program equivalence property; thus, they help the developer with both question (1) and (2).

Proposed by Jackson and Ladd [37], the first of these two methods describes an early approach for providing a semantic diff for programs. They leverage dependence relations of variables to report changes in the program to the user. The goal of this approach is to increase developer understanding of the effects of program changes. This approach does not directly solve either of the problems highlighted at the start of this subsection; rather they help the developer understand the changes in order to answer the questions.

Jackson and Ladd provide an approach for visualising the difference between program versions. By showing changes in control- and dataflow, the authors increase developer understanding of the changes that occur. However, such change information is not sufficient to soundly determine equivalence of two programs; therefore, it does not apply to the *must-summary checking problem*.

The second method, which was proposed by Person et al. [36], is an approach to differential analysis based on symbolic execution. The researches apply a combination of abstract and symbolic summaries to verify that two versions of a program are semantically equivalent. The researchers exploit the similarity between program versions to improve and refine analysis results.

This aspect allows for the application of the method to the *must-summary checking problem*, as the technique does not spend computational resources on showing equivalence for sections of the program that are equivalent. The approach is complementary to the methods proposed in Chapter 5, as their approach to determine equivalent parts of the code that can be extended with the algorithms proposed in this thesis.

The previous two techniques provided tools to characterise program changes, another set of approaches [35], [38], [41] tries to solve the first question “Did the change introduce any unwanted behaviour or remove desired behaviour?” by showing program equivalence.

Godlin and Strichman [38] implemented an technique for regression verification using equivalence checking. This helps developers to verify that refactors do not introduce unwanted changes, which solves question (1) that was posed at the beginning of this section. In their approach, they transform two versions of a program

into loop-free and recursion free versions of that program through substitution with uninterpreted functions. Consecutively, the programs are transformed into static single assignment (SSA) form. This form is leveraged to dispatch an equivalence query to an SMT solver.

Unlike the technique proposed in this thesis, Godlin and Strichman do not leverage the fact that two versions of a program are very close. However, their approach can deal with generic program changes. Therefore, their approach is complementary to ours with regards to the *must-summary checking problem* and can be used to prove equivalence for those changes where the algorithms in this thesis are not able to determine validity.

Lahiri and Hawblitzel implemented a tool called SYMDIFF [35], a semantic difference tool for imperative programs that uses verification conditions rather than a technique based on symbolic execution. In their approach, they ask the user to provide two versions of a loop-free program and a mapping between the functions of the two program versions. They then formulate a procedure for each function that checks for partial equivalence. This procedure calls the two versions of a function with the same inputs. Additionally, it includes an assertion that the outputs of the two versions of the function must be the same. The generated procedures are thereon checked for faults using the Boogie modular verifier [42].

Similar to the research by Godlin and Strichman [38], and Person et al. [36], this approach to equivalence checking can deal with general program changes. This allows it to be used in unison with the algorithms proposed in this research, to try and check for partial equivalence for those summaries where the proposed algorithms are unable to show equivalence.

Backes et al. [41] propose an additional approach for equivalence checking and regression verification. This technique leverages DiSE [21] to create impact summaries, which summarise the behaviour of modified parts of the code. By showing that the impact summaries for two versions of a program are equal, they are able to demonstrate semantic equivalence.

Similar to the earlier work by Person [36], the approach proposed by Backes et al. leverages symbolic execution to determine the equivalence of a program. The approach itself is based on computing the symbolic summaries for the different versions of the program. This technique can be extended with algorithms proposed in this thesis, allowing the equivalence checker to re-use summaries for parts of the code that can efficiently be shown equivalent.

Reverse engineering

In addition to the application to formal methods, and the improvement of development processes, differential program analysis has also seen a successful application in reverse engineering applications. Here program differencing is used in two ways. Firstly, difference data can be used to port reverse-engineered information efficiently between program versions. Secondly, differences allow for the identification of software patches, allowing for targeted manual analysis. These goals overlap precisely with those of incremental formal verification. In this section, we provide an overview of the work in this field and compare it to the contributions of this thesis.

BinDiff is a well-known binary differencing tool that leverages graph-theoretical approaches to compare binaries [43]. Dullien and Rolles [44] introduce these approaches and implement a binary differencing analysis using graph comparison. In their paper, they identify three change types that occur between two variants of the same executable:

1. Different Register Allocation
2. Instruction Reordering
3. Branch Inversion

These change types map to some of change types described in Section 3.2.2. For the purposes identified in their paper (namely porting reverse engineering results), it is not strictly necessary to soundly approximate differences between two versions of a program. Instead, to compare basic blocks, they use the small primes product, an efficient, but unsound method of comparing two basic blocks.

The algorithms proposed in this thesis find and use a mapping between the basic blocks of two smart contracts. Similar to Dullien and Rolles, we leverage a range of heuristics to map basic blocks between the versions of a program incrementally. Our approach can leverage the heuristics identified by Dullien and Rolles to improve the speed and efficiency of the algorithms; as such, their research is complementary to ours. Additionally, our work provides an extension to that of Dullien and Rolles; Algorithm 2 introduces a heuristic that leverages dataflow relations to map different basic blocks.

Bourquin et al. [45] extend BinDiff with Hungarian algorithm [46] for bipartite graph matching. The main contribution is the addition of a heuristic that considers graph edit distance for the potential mapping of basic blocks. Similar to the heuristics proposed by Dullien and Rolles [44], our algorithm potentially benefits from the extension with this heuristic.

Another technique based on the comparison of the control flow of two programs is that of Ming et al. [47]. In their paper, the authors propose a binary diffing algorithm

that uses interprocedural control flow to match basic blocks between two versions of a program. Furthermore, the authors show that their approach is more resistant to obfuscation techniques such as function inlining. While improvements with regards to obfuscations are not relevant for this thesis, the proposed matching algorithm could be leveraged by Algorithm 2.

Gao et al. [48] introduce a tool called BinHunt. They identify changes to register allocation or instruction selection as potential issues for program differencing. They leverage symbolic execution to compare the semantic effects of basic blocks; as such, their approach becomes agnostic of the changes introduced within a basic block (see Section 3.2.2). In this thesis, we propose a technique based on normalisation, rather than strict equivalence checking using symbolic execution. Further research is required to show which is more efficient on the scope of basic blocks, or whether a hybrid approach is warranted. Furthermore, their approach only considers the comparison of semantic effects between two basic blocks. Normalisation potentially considers a broader scope, without incurring the cost of symbolically executing parts of the target program.

Fluri et al. [49] implement an analysis technique that leverages tree differencing to identify changes in source code. Their approach efficiently finds and characterises program changes between program versions. However, this technique is focused on the identification of syntactic program changes and does not reason about the possible preservation of some program behaviours. Furthermore, the technique allows for finding changes in source codes, rather than bytecodes, such as the approach proposed in this thesis. That said, in their paper, Fluri et al. introduce a taxonomy of different program changes. The fine-grained taxonomy of change types provides a valuable overview of different change types that a must summary checking algorithm might consider. Their taxonomy provides an alternative perspective on program changes when compared to the change categorisation in Section 3.2 as the taxonomy identifies different syntactical changes, while the categorisation particularly considers semantics preserving program changes.

Egele et al. [50] take an alternative approach to similarity testing. They leverage dynamic analysis runs to compare the semantic behaviour of two versions of a program. This approach bases itself on the intuition that similar code must behave similarly, using a dynamic analysis approach they approximate the semantics of a function which can then be used to compare the similarity of two programs. Note that similarity is not sufficient to permit the re-use of analysis results, as similar code might still have semantic differences. Therefore, this research is orthogonal to ours.

Baker et al. [51] design an approach to express syntactic differences between program versions efficiently. This technique allows for the compression of software patches to smaller sizes. Their research is orthogonal to ours, as it revolves around

syntactic differences, rather than the presence or absence of semantic differences.

In addition to efforts from the academic community, we find some open-source tools implementing novel differential analysis techniques. Firstly, a popular binary diffing implementation is called Diaphora [52]. This tool leverages several heuristics to find mappings between the functions of two programs. Similarly, Turbodiff [53] is a tool that allows for function matching. These tools implement functionality to match the functions of two programs, which is not considered by the algorithms in this research.

4.1.2 Incremental program analysis

In this section, we will first look at some incremental analysis approaches that have been used in techniques other than symbolic execution [20], [54]–[57]. Next in Section 4.1.2, we will look at incremental analysis approaches that have been proposed for symbolic execution.

Binkley [58] studied the application of semantic differencing for improving the efficiency of regression testing. He uses this technique to reduce cost in two ways: Firstly, using the difference information, it is possible to distinguish affected test cases from unaffected cases. Unaffected test cases do not have to be re-run, as their results will remain unchanged. Secondly, Binkley can compute a simplified version of the program under test, that only exhibits those changed behaviours, that improve the runtime of those remaining tests. Their approach is a precursor to the one proposed in this thesis. Similarly, they soundly approximate the affected locations in the code, which lets them re-use previous analysis results. However, our work improves upon the prior research of Binkly by introducing normalisation,, intending to remove common semantics preserving change categories. Additionally, in his paper, Binkley uses program slicing techniques to determine whether different changes affect other program changes. In algorithm 3 (see section 5.3), we similarly use dataflow information to improve on the precision of our analysis. As is discussed in section 5.3, this is more effective than program slicing.

An approach for incremental program analysis has been proposed by Leino and Wustholz [20]. They use a flow insensitive approach to detect whether a statement depends on a change in the program. For those assertions where they can show that the assertion is not dependent on a changed statement, they inject assume statements before the assertion. This makes the approach agnostic of the verification tool that is being used to check the validity of the assertions. Their approach to checking for changed statements, and semantic divergence between two versions of a program, while efficient, is limited to syntactic equivalence and does not consider different semantically preserving change categories (see Chapter 3). The

algorithms we propose in this thesis can be used to extend the approach by Leino and Wustholz to be able to re-use more intermediate verification results.

Unlike Leino and Wustholz, Szabo et al. [54] leverage an approach that is orthogonal to ours. In their work, Szabo et al. leverage an incremental solver for rete networks to allow them to formulate a DSL (domain-specific language) that allows the specification of several analyses.

Similarly orthogonal, Rothenberg et al. [57] propose an incremental checking approach based on trace abstraction.

Lastly, an approach to incremental analysis based on summary re-use was proposed by Ondrej et al. [55], [56]. They implement an approach to check the validity function summaries derived using Craig's interpolation.

In their research, Ondrej et al. focus on re-using function summaries. Ondrej et al. check whether previous summaries are still valid over-approximations of new functions. The approaches proposed in this thesis aim to allow the re-use of must-summaries; thus, the approach proposed by Ondrej et al. is orthogonal to ours.

Incremental symbolic execution

Specifically, for symbolic execution, there has been ongoing research interest in incremental analysis techniques as a way to improve and scale the analysis technique. As mentioned in the initial section of this chapter, we identify two main approaches to incremental computation. The first is optimising or directing the coverage of symbolic execution to changed program behaviours, instead of trying to cover the entire program. The second is re-using analysis results from previous executions to reduce redundant computations between analyses.

Person et al. [21] proposed an approach to symbolic execution that directs the symbolic execution to cover changed program behaviours. They do this by first using a static data- and control-flow analysis to compute which program statements are affected by changes in the program. The regular symbolic execution process, as described in Section 2.1, is then used with a depth-first exploration approach. During the symbolic execution the executor will keep track of the affected program statements that have been covered. At each point where the execution reaches a state that is not able to cover a changed program statement that has not yet been covered, then the execution will prune that state. At each point where the symbolic execution covers a changed statement, it will re-try to cover the program statements that are dependent on the just covered statement. In doing so, they guarantee that if the symbolic execution terminates, that they have covered each possible sequence of influenced program statements.

Taneja et al. [22] implemented an approach called eXpress, using a dataflow

analysis they prune paths from the search space that do not meet one of three requirements:

1. The path covers a changed statement
2. The change introduced by the changed statement in a path propagates to the output
3. The changed statement introduces a change in the state.

The purpose of this selection is to do efficient regression testing

Marinescu and Cadar [23] proposed an approach for testing of program patches. Similarly to directed incremental symbolic execution, their approach optimises the analysis' coverage of the changed code. In their approach, they use an existing suite of test cases to seed the analysis of the patch. From the test cases, the input that covers the path with the shortest branch distance of the patch is selected. It then uses a combination of greedy exploration, informed path generation and definition switching to flip branches in this original test case to get an input that covers the desired statements in the patch.

These three approaches [21]–[23] are all directed at steering symbolic executors towards changed parts of the code, rather than the re-use of previously computed analysis results. This aspect makes the approaches complementary to the re-use of summaries, as they optimise different aspects of incremental analysis. Note that these works can leverage the algorithms proposed in this research to direct effort to parts of the program that have semantic changes.

Yang et al. [18] implement a technique called *memoized symbolic execution* in a tool called Memoise. In their approach, they maintain a trie that represents the symbolic search space. In successive iterations of the analysis, they query the trie discovered by previous iterations, this allows for several optimisations.

1. It allows them to refrain from checking the constraints on paths that have previously been covered.
2. It allows them to select the states for which the suffix can not include a changed instruction, and pruning it
3. It allows them to perform a heuristic search

The approach relies on syntactic equivalence with the original program to be able to re-use analysis results. The technique can be extended to filter for many program changes that do not impact the semantics of the program, and thus would permit the re-use of intermediate analysis results such as stored in the trie by Yang et al.

Yang et al. leverage this work to efficiently do incremental property checks in a tool called iProperty [17]. In their approach, they first apply property differencing to find property clauses that are not implied by previous analysis results. Consecutively they leverage DiSE [21], to only symbolically explore the program behaviours that have changed. Yang uses property differences to speed up symbolic execution; as such, their approach is complementary to the re-use of symbolic summaries.

Lauterberg et al. [16] proposed an approach to state-space exploration analysis techniques. In their approach, Lauterberg et al. store a representation of the state space after the analysis of the program. During the analysis of newer versions of the program, they query this representation to see whether some transitions and states have been checked before. Using this information, they defer checking properties over the state if the property has been proven to hold previously. Laterberg et al. leverage the syntactic equivalence of unchanged program parts to speed up symbolic execution. This technique can be extended with the approach in this thesis to check for semantic equivalence rather than pure syntactic equivalence.

4.2 Symbolic summary re-use

In Chapter 1 we introduced symbolic summary re-use and the *must-summary checking problem*. The latter is more formally described in Section 2.2.4. Symbolic summary checking and the possibility of re-use was introduced by Godefroid et al. [1]; in their paper, they introduce a formalisation of the *must-summary checking problem* and three algorithms to solve the problem.

In their first algorithm, Godefroid et al. [1] use a light-weight control flow analysis and syntactic equivalence check to show that all paths between the entry and exit point of a summary are unchanged.

Their second algorithm uses a more precise analysis, using Boogie [42], to determine which basic blocks are potentially part of the summarised code.

These two techniques use syntactic equivalence checks to determine whether code has changed. As mentioned before this is unnecessarily restrictive; this thesis extends upon this work by permitting summary re-use for code with syntactic changes.

Furthermore, we observe that the approach used to determine which basic blocks are summarised by a symbolic summary is both computationally expensive and imprecise. Rather than approximating the path summarised, we record the basic blocks covered by a summary, during the summary generation. By recording the basic blocks covered by a symbolic summary, we remove the need for expensive approximation.

The final algorithm proposed by Godefroid et al. does not consider possible similarities or equalities between two versions of a program. Instead, they aim to prove the validity of a must-summary directly using Boogie [42]. This aspect makes this third algorithm complementary to light-weight must-summary checking algorithms, as this algorithm can be used whenever it is not possible to check the validity of a must-summary using a more light-weight approach.

Chapter 5

Approach

In Chapter 4, we discussed the current state-of-the-art in incremental and differential analysis techniques. Our primary observation is that the program differencing approaches which use the similarity between two versions of a program usually leverage trivial syntactic equivalence checks to determine which parts of the code have changed.

However, using purely syntactic equivalence is unnecessarily restrictive. In this chapter, we propose an approach (see Section 5.2) which allows incremental analysis techniques to re-use more intermediate analysis results. We achieve this by increasing the ability for the tools to show the equivalence of parts of the code that have syntactic changes without any semantic effect.

Additionally, we introduce a novel approach which leverages the dataflow relations in a program to determine whether a semantic change affects a symbolic summary.

We designed these algorithms with the *must-summary checking problem* in mind, but they are extensible to other incremental analysis techniques (see Section 4).

We propose a set of algorithms that can be used in successive phases. Algorithms 1 to 3 each provides increasingly extensive analyses, increasing both the computational cost and the algorithm's ability to show equivalence.

5.1 Algorithm 1

The first algorithm we propose in this thesis constitutes a syntactic bytecode equivalence check.

While such an algorithm does not provide a significant contribution, it does offer a practical performance bonus. The observation driving the inclusion of this algorithm is that there are frequent occurrences of fully equivalent bytecodes within the benchmark sets to validate the approaches in this thesis. While the ensuing two

algorithms also allow summary checking for unchanged contracts, this algorithm provides a more efficient approach. This algorithm is more efficient since a string comparison of the original, and new bytecodes are sufficient to proof validity of all previous summaries, given that those two bytecodes are equivalent.

5.1.1 Algorithm

function CHECKSUMMARY($originalProgram, newProgram, summary$)
    return originalProgram.bytecode == newProgram.bytecode
end function

Figure 5.1: Pseudo code algorithm 1

Figure 5.1 demonstrates the pseudocode for this algorithm. The correctness of this approach is intuitive, as naturally, it seems apparent that intermediate results can be re-used when a program has not changed.

More formally, we can demonstrate the correctness of this approach using negation. Assume there is one summary valid for the original program ”Program A” and not valid for the new program ”Program B”. Furthermore, assume that Program A and Program B are syntactically equivalent.

These two programs can not be semantically equivalent since there is some behaviour in Program A that is not in Program B. Semantic inequivalence of two programs implies that there is a path through Program A, which is not present in Program B. Thus; Program A and Program B cannot be syntactically equivalent. This conclusion contradicts the syntactic equivalence assumption. Therefore, the syntactic equivalence of two programs implies the validity of symbolic summaries between the two versions.

5.1.2 Conclusion

We identify a range of cases in real-world scenarios where there are no changes in analysis targets between two versions of a project. While more refined analysis techniques can correctly validate these summaries, this algorithm is more efficient when faced with exactly equivalent bytecodes. Here we provide a trivial algorithm that allows for the efficient evaluation of the *must-summary checking problem* for programs without changes.

5.2 Algorithm 2

The second algorithm we propose poses a novel approach to summary checking that extends the current-state-of-the-art enabling the re-use of symbolic summaries for code with syntactic changes. Program changes can be purely syntactic, meaning that the changes only affect the appearance of the program. Such changes do not affect the semantics of a program, which is crucial for program analysis and the re-use of program analysis results. Summary checking algorithms can be sensitive to these syntactic changes, and unable to re-use analysis results for code which exhibits semantics-preserving changes. We introduce an approach that transforms the program into a representation where many syntactic changes are normalised, which makes it less sensitive to syntactic changes.

There are two core intuitions to this algorithm. Firstly, the language of a program can be sensitive to program changes. For example, a stack-based language such as EVM is sensitive to changes to the stack layout. Secondly, both the user (see Section 3.1.3) and compiler (see Section 3.1.1 and Section 3.1.2) can introduce purely syntactic changes that can efficiently be normalised.

This algorithm leverages a translation, and several normalisation passes to handle a range of program changes. In particular, the design of the algorithm enables summary checking agnostic of the following change types (see Section 3.2):

1. statement and parameter re-ordering
2. equivalent statements (arithmetic operation change)
3. effectless changes

The section below demonstrates how the algorithm handles these change types.

5.2.1 Algorithm

The pseudocode functions in Figure 5.2 and Figure 5.3 represent the core logic of this algorithm. These figures introduce the normalisation and summary checking logic, respectively. In this section, we will first discuss the normalisation procedure, followed by the summary checking logic.

5.2.2 Normalisation

Above, we formulated the two primary insights that drive the design of this algorithm. The normalisation procedure performs the transformations necessary for the desired outcome.

```
function NORMALISE(originalProgram, newProgram)
    CONVERTTOSSA(originalProgram)
    CONVERTTOSSA(newProgram)
    NORMALIZEARITHMETICEXPRESSIONS(originalProgram)
    NORMALIZEARITHMETICEXPRESSIONS(newProgram)
    PROPAGATECONSTANTS(originalProgram)
    PROPAGATECONSTANTS(newProgram)
    REMOVEDEADCODE(originalProgram)
    REMOVEDEADCODE(newProgram)
    MAPVARIABLENAMES(originalProgram, newProgram)
    ORDERSTATEMENTS(originalProgram, newProgram)
end function
```

Figure 5.2: Pseudocode for normalisation procedure

```
function CHECKSUMMARY(originalProgram, newProgram, summary)
    NORMALISE(originalProgram, newProgram)
    mapping = GETMAPPING(originalProgram, newProgram)
    originalSubgraph = GETSUBGRAPH(originalProgram, summary.trace)
    newSubgraph = GETMAPPEDSUBGRAPH(newProgram, originalSubgraph, mapping)
    return originalSubgraph.equals(newSubgraph)
end function
```

Figure 5.3: Pseudocode algorithm 2

First, we translate the program into an SSA representation, which is less sensitive to program changes. Second, we perform a range of normalisation transformations that aim to remove a range of purely syntactic changes.

SSA conversion

As mentioned above, the first transformation in the normalisation procedure comprises a conversion from the original language (EVM bytecode) to an SSA form representation. SSA (static single assignment) [59] form is a property of intermediate representations. This property ensures that each variable is defined and written to only once. As a result, programs in SSA form encode explicit dataflow relations. The application of SSA representations is a well-known practice in compiler engineering [59], which enables efficient transformation procedures like dead code removal and constant propagation.

In this thesis, we consider the *must-summary checking* problem in the context of EVM smart contracts. The language of those smart contracts, EVM bytecode, is

stack-based and therefore is sensitive to changes in the stack layout. In the translation to an SSA based representation, the stack-based language is transformed into a register-based representation.

Using an SSA form, register-based internal representation (IR for short) provides a benefit for summary checking, as it is agnostic of stack layout. This property enables the instruction order normalisation discussed below. Furthermore, similar to the application in compiler systems, we leverage the SSA form in the normalising transformations which apply constant propagation and dead code removal.

Normalisation procedures

Following a transformation to an SSA form representation of the programs, we apply a series of normalisation procedures. We implement these normalisation procedures based on the kinds of changes discovered in Chapter 3.

Note that there is no restriction on the implemented normalisation passes. Depending on the target architecture or language, different normalisation procedures can be beneficial. Even more so, we identify finding additional effective normalisation transformations for EVM as a topic for future research.

Figure 5.2 introduces the applied normalisation transformations, which we will re-iterate:

- Constant propagation
- Dead code removal
- Arithmetic normalisation
- Variable name mapping
- Statement order normalisation

Constant propagation

Constant propagation is a compiler optimisation technique [59]. It leverages abstract interpretation in the domain of constants to determine which variables in the programs have a constant value. The compiler uses this information to replace uses of constant variables with the constant value.

This procedure targets changes of the three different kinds discussed previously as constant propagation normalises semantics preserving changes to constant expressions.

Original	Replacement
a: variable + b: constant	b + a
a: variable * b: constant	b * a
a: constant + b: constant	*evaluate
a: constant * b: constant	*evaluate
a: constant + (b: constant + c:any)	(a + b) + c
a: constant * (b: constant * c:any)	(a * b) * c
a: variable + (b: constant + c:any)	b + (a + c)
a: variable * (b: constant * c:any)	b * (a * c)
a: const mod b: const	* evaluate
(a mod b) * (c mod b)	(a * c) mod b

Table 5.1: Arithmetic rewrite rules

Dead code removal

The second normalisation approach, dead code removal, is another compiler optimisation technique [59]. This technique leverages dataflow relations within a program to find the statements that do not affect the result of the program. Since they do not affect the result of the program, they can be removed without changing the meaning of the program. For a compiler this means that it will not have to emit bytecode for dead code, which allows for improved performance. For the summary checking algorithm it has the effect that the changes in dead code are removed from the program, allowing the algorithm to deal with *effectless changes*.

Arithmetic normalisation

Additionally, we leverage a simple normalisation of the arithmetic expressions in each basic block, allowing the algorithm to handle semantics preserving changes to arithmetic expressions. This normalisation contributes to the algorithms ability to verify summaries in the presence of the change type *equivalent statements*.

This normalisation is similar to the expression simplification that occurs within compilers. However, compilers aim to optimise code, whereas the purpose of normalisation is to remove syntactic divergences. Instead of trying to reach a normal form, the goal of this procedure is to normalise the syntactic changes commonly introduced by developers and compilers. To normalise the arithmetic expressions, we leverage a fixpoint algorithm that applies a set of rewrite rules until it reaches a fixed point. The rewrite rules can be found in Table 5.1.

Variable name mapping

function MAPVARIABLENAMES($originalProgram, newProgram$)
  $mapping$ = FINDMAPPING($originalProgram, newProgram$)
  $worklist$ = $mapping$
  while $worklist$ **do**
    for $originalBasicBlock, newBasicBlock$ in $worklist$ **do**
      MAPVARIABLES($originalBasicBlock, newBasicBlock$)
    end for
    $newMapping$ = FINDMAPPING($originalProgram, newProgram$)
    $worklist$ = $newMapping \setminus mapping$
    $mapping$ = $newMapping$
  end while
end function

Figure 5.4: Pseudo code for variable name mapping

The fourth transformation applies a mapping procedure that renames the variables that are recurrent between the two versions of the program. Unlike the other procedures, this pass is the unique transformation that correlates between the original program and the newer version.

Figure 5.4 shows a pseudocode representation of the implemented variable renaming approach. In essence, the algorithm iteratively performs two actions until it reaches a fixed point. Firstly, the algorithm computes a partial mapping between the basic blocks of the two versions of the program. Subsequently, the variable renaming procedure finds a mapping between the variables in the mapped basic blocks and renames the variables in the new version to match their mapped equivalents in the original program. It is possible to re-use information between the iterations of the algorithm; we refrain from discussing these for simplicity. The result of the variable renaming then provides additional information to the basic block mapping procedure.

Basic block mapping It is necessary to find a mapping between similar basic blocks to find the basic blocks for which we can apply the variable name mapping. This algorithm (see Figure 5.6) applies a set of heuristics to determine which blocks are sufficiently related to apply the variable name mapping procedure.

The following is a list of the applied heuristics:

1. Two basic blocks do not have conflicting usage of previously mapped variables
2. Two basic blocks share uniquely mapped parents
3. Two basic blocks share uniquely mapped children

```
function FINDMAPPING(originalProgram, newProgram)
    worklist = originalProgram.basicBlocks
    mapping = new Map()
    while worklist do
        originalBasicBlock = worklist.pop()
        for newBasicBlock in newProgram.basicBlocks do
            score = SCORE(originalBasicBlock, newBasicBlock)
            if score > threshold then
                mapping[originalBasicBlock] ∪ [newBasicBlock]
            end if
        end for
    end while
    REMOVENONUNIQUE(mapping)
     return mapping
end function
```

Figure 5.5: Pseudo code to find a mapping between programs

4. Two basic blocks are both entry points to the contract

5. Two basic blocks have similar instructions

This procedure is run incrementally along the renaming process, as more variable names get mapped more basic blocks will be uniquely mappable.

Variable mapping Above we describe the basic block mapping algorithm, the other main component of the variable renaming normalisation procedure is the variable name mapping procedure. Whereas the previous algorithm discovered a mapping on the scope of basic blocks, this algorithm will discover a mapping between the variables in the basic blocks. The variables in the representation of the changed program will be changed to reflect their mapped equivalent in the original basic blocks using the mapping discovered using the process above.

The process can be described as follows. Initially, the algorithm builds a forest-like graph representation of the instructions in both basic blocks, similar to the instruction order normalisation procedure. Such a representation neatly encodes the dependencies between the instructions in the basic blocks. Then, the algorithm iterates over each layer of the forest graph correlating variables and building up a mapping between the variables. Finally, the variable names in the entire control flow graph are renamed based on the discovered mapping.

More variables in the control flow graph are mapped after the successful completion of the variable renaming pass. The renamed variables then provide improved

```
function SCORE(originalBasicBlock, newBasicBlock)
    if MAPPEDVARIABLECOLLISION(originalBasicBlock, newBasicBlock) then
        return 0
    end if
    score = 0
    if SHAREUNIQUELYMAPPEDPARENT(originalBasicBlock, newBasicBlock)
then
        score += 1
    end if
    if SHAREUNIQUELYMAPPEDCHILD(originalBasicBlock, newBasicBlock) then
        score += 1
    end if
    if AREENTRYNODES(originalBasicBlock, newBasicBlock) then
        score += 1
    end if
    if SIMILARINSTRUCTIONS(originalBasicBlock, newBasicBlock) then
        score += 1
    end ifreturn score
end function
```

Figure 5.6: Pseudo code for scoring potential basic block matches

input to the basic block mapping procedure, which can leverage the discovered variable relations.

Instruction order normalisation

The last transformation normalises the order of statements in a basic block, which enables handling of the change type *statement and parameter re-ordering*.

First, the procedure constructs a directed graph of the instructions in the basic block. In the graph, the nodes represent the instructions, and the edges the dependency relations between the instructions. We say that an instruction is dependent on another when changing their order will change the semantics of the program.

For the construction of the graph, we leverage the fact that the internal representation is in SSA form, which explicitly encodes dataflow relations between the statements in the basic block. Each of the dataflow dependencies is added to the graph, except those caused by loops in the program. We do not include these relations, as they do not constrain the order of the statements within the basic block. In addition to these dataflow relations, we add dependency edges to jump instructions when they are present, as changing their position in the basic block likely has

a semantic effect.

Followed by the construction of this graph, we assign a layer value to each of the nodes using the formula described in Figure 5.7.

Within each layer, the order of instructions may change since there are never dependencies between the nodes of one layer. Computation of the layers requires linear time since the graph does not have any cycles.

After having computed the layers, the algorithm sorts the statements within each layer based on their operation name and arguments.

$$layer(statement) = \left\{ \begin{array}{ll} 0, & \text{isPHINode(statement)} \\ 1, & \text{parents(statement)} = \emptyset \\ max(map(layer, parents(statement)) & \end{array} \right\}$$

Figure 5.7: Node layer formula

5.2.3 Correctness

To reason about the correctness of the algorithm we have to identify the high-level structure of the technique. Algorithm 2 takes two versions of a program and a set of symbolic summaries as its input. The algorithm then performs a normalising transformation on both programs. Consecutively, the algorithm will leverage the normalised program representations to check the validity of every summary.

This section first provides an intuitive explanation for the correctness of this approach, followed by a formalisation of the intuition.

Intuition

There are two intuitive properties that indicate the correctness of the algorithm:

(property 1) Normalising transformations do not introduce semantically divergent behaviour. Therefore, any must-summary that would be valid for an original program is also valid for the normalised version of that program, and vice versa.

(property 2) If there are two syntactically equivalent paths through the two programs, then a summary for either path must be valid for both.

The following is a concrete example to demonstrate these intuitions in the summary checking technique.

Suppose we have two programs, O (for original) and N (for new). Additionally, let S be a valid symbolic summary for program O.

The algorithm will start by normalising the original and new programs; the normalised programs will be called O' and N' respectively. From property 1, we have that S must be valid for O' since it was valid for O.

The algorithm will now try to find if the path through O', summarised by S, is also present in N'. If it cannot find such a path, then the algorithm will not be able to determine the validity of the summary.

If a syntactically equal path through N' does exist, then the summary S must also be valid for N' according to *property 2*.

Lastly, when S is a valid summary for N', then it must also be valid for N following the *property 1*.

Theorems

This section formalises the two properties introduced above as Theorems 5.2.1 and 5.2.2.

Theorem 5.2.1 elaborates the first property mentioned above: "The normalising transformations do not introduce semantically divergent behaviour. Therefore, any must-summary that would be valid for an original program is also valid for the normalised version of that program, and vice versa."

Theorem 5.2.1. *Let $N(Prog)$ be a normalising transformation that takes a program and produces a normalised version $Prog_n$ of that program. Furthermore, assume that N is semantically preserving and does not alter the control flow of the program. Then, for any path t in $Prog$ there exists a path t' in $Prog_n$ which is semantically equivalent to t. Furthermore, any valid summary that summarises t or t' is a valid summary for either.*

Proof. For each path t through $Prog$ there must be a path t' through $Prog_n$ which executed under the same precondition, as the normalisation procedure does not introduce changes that alter the semantics or control flow of the program. Additionally, the effect of t' must be semantically equivalent to that of t since $Prog$ and $Prog_n$ are semantically equivalent. Thus t and t' must be semantically equivalent. Since t and t' are semantically equivalent, any summary valid for either is valid for both. □

Theorem 5.2.2 elaborates the second property mentioned above: "If there are two syntactically equivalent paths through the two programs, then a summary for either path must be valid for both."

Theorem 5.2.2. *Let $Prog$ and $Prog'$ be two syntactically inequal programs. Furthermore, let there be two paths $t \in Prog$ and $t' \in Prog'$. Lastly, let S be a symbolic summary. If the paths t and t' are syntactically equivalent, and S is a valid summary that summarises either, then S must be valid for both $Prog$ and $Prog'$.*

Proof. The correctness follows from the fact that syntactically equivalent code must also be semantically equivalent. Recall that Algorithm 1 leverages the same property to show validity when the entire program is syntactically equivalent. Different from the application of this property in Algorithm 1, here reasoning centres around semantic equivalence of individual paths rather than entire programs.

Formally the correctness of the theorem can be shown through negation. Assume two programs $Prog$ and $Prog'$ which respectively have a path t and t'. Let t and t' be syntactically equivalent and let t be summarised by a summary S. Assume that S is not a valid summary for t'. Since S is not a valid summary for t', t and t' can not be semantically equivalent. Since t and t' can not be semantically equivalent, they must have some syntactic divergence. This last statement leads to a contradiction with the prior assumption that t and t' are syntactically equivalent. Therefore, the theorem must hold. □

Conclusion

When combined, the beforementioned theorems demonstrate the correctness of Algorithm 2.

The first phase of the algorithm, namely the normalisation transformation, satisfies the requirements of the normalisation procedure defined in Theorem 5.2.1. In this phase, two versions of a program O and N are normalised into O' and N'.

The subsequent phase takes each summary S for O and finds the path t through O' summarised by S. When it finds that there is a path t' through N' which is syntactically equivalent to t, it will be able to determine that S is valid for N' (following Theorem 5.2.2) and thus also for N (following Theorem 5.2.1).

5.2.4 Conclusion

In conclusion, we designed an algorithm that leverages a conversion into SSA form and normalising procedures to gain a representation that is resistant to syntactic changes. The algorithm includes specific normalisation passes that target changes such as those discussed in the introduction of this section. However, the approach is not limited to the formulated normalisation procedures. It is possible to extend the normalisation passes used by the algorithm to cover more change categories. Furthermore, use of an alternative intermediate representation is also possible. Similar to the benefits of SSA, an alternative language might have additional beneficial properties for summary validation.

5.3 Algorithm 3

In the previous section, we described Algorithm 2. This algorithm performs several transformations on a program to make it less sensitive to syntactic changes. One of the change categories that the algorithm can deal with is changes in program statements that have no semantic effect (also called *dead code*). A similar program change category is that of changes to *partially dead code*. A statement is partially dead when there are executions of the program where the statement has an observable effect and executions where the statement does not. (see Figure 6.5 for an example of a change to partially dead code). Algorithm 3 provides an improvement on Algorithm 2 by allowing the analysis to ignore partially dead statements if they do not affect a summary.

5.3.1 Algorithm

function CHECKSUMMARY($originalProgram, newProgram, summary$)
    NORMALISE($originalProgram, newProgram$)
    mapping = GETMAPPING($originalProgram, newProgram$)
    originalSubgraph = GETSUBGRAPH($originalProgram, summary.trace$)
    newSubgraph = GETMAPPEDSUBGRAPH($newProgram, originalSubgraph, mapping$)
    REMOVEDEADCODE(originalSubgraph)
    REMOVEDEADCODE(newSubgraph)
    return originalSubgraph.equals(newSubgraph)
end function

Figure 5.8: Pseudo code algorithm 3

Figure 5.8 demonstrates the pseudocode for this algorithm.

A comparison of the pseudocode for Algorithm 2 and Algorithm 3 demonstrates significant overlap. To simplify the pseudocodes and explanations, we have duplicated these statements between the two algorithms. However, an implementation of the proposed algorithms might run the three proposed algorithms in succession and re-use the normalisation results between Algorithm 2 and 3.

As described above, the intuition behind this algorithm is that we can ignore program changes in statements that do not affect the result of a summary under test.

To achieve this, we require a method of determining which statements affect the result of a summary. A commonly used analysis technique used for this purpose is program slicing. The program slicing technique starts with a *slicing criterion*, which describes a set of program statements and variables. Consecutively, an algorithm

will compute all statements that have a direct or indirect effect on the slicing criterion. However, such an approach is unnecessarily general for our purpose, since a program slice can still include *partially dead* program statements. In the evaluation (see Figure 6.5) we introduce an example of exactly such a change to a *partially dead* program statement. Therefore we designed an alternative approach that would achieve the desired benefits. The key insight behind this algorithm is that dead code analysis achieves something very close to the desired effect. Namely, it removes all statements that do not have a semantic effect. In this particular case, we want to remove all statements that do not have a semantic effect within a specific path. Therefore we take the subgraph of a program with just those basic blocks summarised by the summary. Given this subgraph, we perform a dead code analysis, which will remove any statements that do not affect the semantics of this subgraph and summary.

5.3.2 Correctness

To demonstrate the correctness of this third algorithm, we leverage the theorems introduced in Section 5.2.3. Additionally, we formulate Theorem 5.3.1, which reasons about the additional analysis performed in Algorithm 3.

Theorem 5.3.1. *Assume a path t through a program $Prog$. Furthermore, let $M(t)$ be a transformation, which removes all statements that do not affect the semantics of t, and produces t'. Then a summary valid for t is valid for t', and vice versa.*

Proof. M preserves the semantics of t as it only removes statements without a semantic effect. Thus, since t and t' are semantically equivalent, a summary valid for either must be valid for both paths. □

Conclusion

When combined with the theorems discussed in the previous section; Theorem 5.2.1 and Theorem 5.2.2, Theorem 5.3.1 demonstrates the correctness of Algorithm 3.

Similar to Algorithm 2, the first phase of this algorithm applies a normalisation transformation, which satisfies the requirements of the normalisation procedure defined in Theorem 5.2.1. The result of this phase is the programs O' and N', the normalised versions of O and N respectively.

The subsequent phase takes each summary S for O and finds the path t through O' summarised by S. Furthermore, the algorithm determines a corresponding path t' through N' based on the discovered mapping between the basic blocks of the two normalised programs (see Section 5.2.2). The algorithm will then remove any statements in t and t' that do not affect their semantics, producing t_d and t'_d respectively.

When it finds that t_d and t'_d are syntactically equivalent, it will be able to determine that S is a valid summary for N using the following reasoning:

1. We know that S is a valid summary for $Prog$, summarising the path t.
2. S must be a valid summary for O' summarising the path t' (following Theorem 5.2.1).
3. Thus S also summarises t_d (following Theorem 5.3.1).
4. Since S summarises t_d, S must also summarise t'_d (following Theorem 5.2.2).
5. As S is valid for t'_d, it also summarises t' in N' (following Theorem 5.3.1).
6. Finally, S must be a valid summary for N (following Theorem 5.2.1).

5.3.3 Conclusion

In conclusion, Algorithm 3 provides an extension to Algorithm 2, allowing for summary re-use in the presence of semantic changes that do not affect a particular summary. It does so by leveraging previously computed analysis results, and a compiler optimisation approach called dead code analysis.

Chapter 6

Evaluation

This chapter provides an evaluation of the techniques proposed in Chapter 5. First, we describe the implementation of the algorithms and symbolic summary generation plugin. Then we discuss three benchmarks, which evaluate the efficiency of the approach for particular program changes, real-world projects, and changes introduced by compiler version changes respectively. All experiments were executed on an Ubuntu 19.10 machine with a Threadripper 1950X CPU and 32GB ram.

6.1 Implementation

In the previous chapter, we described three summary checking algorithms which enable the re-use of symbolic summaries and incremental symbolic execution. We provide an implementation of these algorithms in a tool called Eternity.

We identify three main areas in Eternity:

1. Core
2. Static analysis
3. Summary checking

Core To the core component belongs the logic related to data models and the logical operations on it. For example, the control-flow graph (CFG) and data-flow graph (DFG representations are implemented here. Examples of the logical operations include operations that allow for querying control- or data-flow and functions which perform mutations on the control-flow graph.

Static Analysis Eternity leverages the Vandal static analysis framework [8] to perform abstract interpretation on smart contracts. This framework allows for analysis in the constant domain and recovers the control- and data-flow of the smart contract.

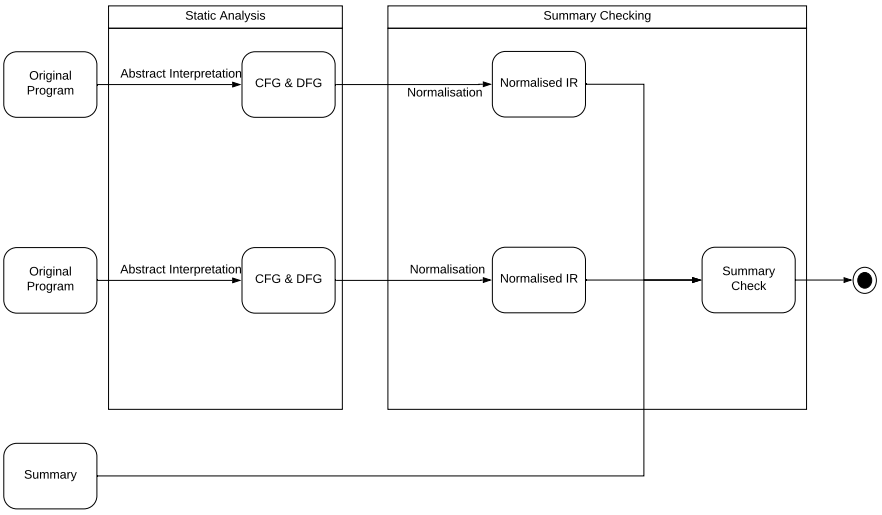


Figure 6.1: High level flow of Eternity

Summary Checking The summary checking component is the primary area of Eternity which regards the implementation of the proposed algorithms. This component leverages leverages the **Core** and **Static Analysis** components to perform the normalisations, data-flow analysis and summary validation.

Figure 6.1 demonstrates the flow of Eternity and highlights which elements of the process are handled **Summary Checking** and **Static Analysis** components . The diagram does not explicitly indicate where the **Core** component is used, as it is used uniformly throughout the summary checking process.

The remaining part of this section describes the extension to Mythril, enabling the generation of summaries for this evaluation. Additionally, at the end of this section, we reflect on the implementation and its limitations.

6.1.1 Mythril

This thesis introduces an approach to the must-summary checking problem that enables the re-use of symbolic summaries for EVM bytecode. Mythril is a security analysis tool that targets EVM bytecode, and can benefit from the proposed algorithms to enable incremental analyses. In this chapter, we leverage the symbolic execution engine in Mythril to generate summaries for the benchmarks in the next sections.

Internally, Mythril applies symbolic execution (see Section 2.1) to explore the different behaviours of a smart contract. The primary application of Mythril is bug

finding (see Section 2.1.2). Here, Mythril enables the detection of bugs such as integer overflows and re-entrancy vulnerabilities [6]. Mythril also has more generic detectors that can identify code structures that allow an attacker to extract funds or destroy a contract. In addition to bug finding, Mythril has also been used for the property checking problem [60].

We implement two extensions to the Mythril analysis tool. First, we implement support for plugins in Mythril, allowing for modification of the symbolic execution process. This improves the extensibility of Mythril with regards to existing and future work on symbolic execution. Second, using the plugin architecture in Mythril, we implemented support for symbolic summarisation.

Plugin

To explain the architecture of the symbolic summary plugin, we first provide some background on the EVM and how Mythril models the relevant components of the EVM.

The EVM has two principal state components. First is the global state, which is the persistent state of the blockchain. The global state includes the current state of the persistent storage, as well as which accounts exist and how many Ether (the native currency of the Ethereum blockchain) they have. Second is the machine state, which only persists during the execution of a transaction on the Ethereum network. The current program counter, stack and memory regions are examples of elements in the machine state.

The summary generation plugin will generate symbolic summaries that describe an entire transaction. As a result, the generated summaries do not need to define mutations that occur on the machine state, because the machine state does not persist after evaluation of a transaction.

In Figure 6.2, we provide a pseudocode representation of the algorithm and how it alters the behaviour of the symbolic executor. We have excluded logic related to the application of symbolic summaries for simplicity. Furthermore, the execution of a symbolic transaction might result in zero or more symbolic summaries. For simplicity the pseudocode will assume that the symbolic transaction results in a single *resultingState*.

The process has three phases. First is the setup phase, where the algorithm creates a state with plain symbolic variables and constrains them to be equal to the values of the input *globalState*. It is easier to compute the effects of the ensuing symbolic execution on the state because the new state is symbolic. In the second phase, the function applies regular symbolic execution on the created symbolic state. Finally, in the third phase, the result of the execution is recorded and reported

as a symbolic summary. After which, the function uses the summary to compute the successor state to the input *globalState*.

function EXECUTESYMBOLICTRANSACTION($globalState, transaction$)
    symbolicState = generateSymbolicState() ▷ Phase 1
    constrainSymbolicState(symbolicState, globalState)
    resultingState = execute(symbolicState, transaction) ▷ Phase 2
    summary = computeSummary(symbolicState, resultingState) ▷ Phase 3
    return summary (globalState)
end function

Figure 6.2: Summary recording

6.1.2 Discussion

Here we reflect on three aspects of the implementation, relevant to the evaluation of the algorithms.

Gas modelling

The use of gas, or gas-metering, is a method to price the computation of a transaction [2], which is a part of the EVM semantics that the algorithms intentionally do not model. In this section, we describe the mechanics of gas, followed by a discussion on its significance for summary re-use.

Background Putting a price to execution serves several purposes in the Ethereum blockchain.

Firstly, it prevents potential abuse, where users send transactions that consume many computational resources, inhibiting the throughput of the network for real traffic. By associating a cost to the expended computational resources, it becomes unprofitable to perform this type of attack.

Additionally, gas-metering puts a monetary incentive on the optimisation of smart contracts. As a result, developers optimise their code and put only critical business logic in their smart contracts, potentially saving on storage and improving the throughput of the Ethereum network.

Lastly, a finite amount of gas (called the gasLimit) is available for each block, which guarantees termination of the smart contracts.

Each transaction has an amount of available gas for the execution of a smart contract. The evaluation of each instruction costs some gas which is deducted from the available gas. Once the execution has finished the remaining gas is refunded to

the transaction sender. The virtual machine will halt in an exceptional state should the gas ever run out.

Gas and summary re-use As described above, gas is not accounted for by the algorithms in Chapter 5.

Firstly, it is our observation that many smart contract analysis approaches do not model exact gas cost [6], [8].

Secondly, the smart contract source language Solidity is agnostic of gas cost. Therefore, it is not necessary to accurately model gas cost for functional properties over the Solidity smart contracts.

Lastly, program analyses that currently consider potential vulnerabilities that relate to gas usage [61] also do not require accurate gas estimation. Instead, they recognise constructs in the code that are vulnerable, regardless of the precise gas cost that would be associated with specific instructions.

Thus, there would be little merit to the accurate modelling of gas for the two primary purposes: bug finding and property checking. On the other hand, accurate modelling of gas usage would significantly reduce the capability to re-use summaries for changed code since many, otherwise semantics preserving, changes will affect the gas usage of parts of the program.

Computational cost of SSA conversion

The first step of the proposed algorithms is a conversion from the stack-based instruction language EVM, to a single static assignment form intermediate language. The translated program is more amenable for the later normalisations and transformations, as SSA form programs explicitly encode the data-flow within the program. This translation is non-trivial, and as seen in Chapter 7, can require relatively many computational resources. For this reason, one might consider improving our algorithms by removing the dependency on this translation. Such a change could indeed improve the performance of the algorithm when only the *must-summary checking problem* is considered. However, the SSA translation of the original program is not a worthless byproduct. Consider the symbolic executor Mythril. While available on its own as an open-source project, Mythril is just one component in a commercial bug finding application called MythX. In this platform, MythX engineers strive to leverage and combine multiple analysis techniques to provide better and more accurate analysis techniques. In such an application, SSA conversion of the bytecode is likely to occur, regardless of the possibility for symbolic summary re-use. In this scenario, the summary checking algorithms could use the previously computed SSA form representation, rather than compute the SSA form of a program, saving much of the otherwise required checking time.

Abstract interpretation failure

The implementation of the proposed approaches occasionally encounters a program that it is not able to recover control- and data-flow relations for within a reasonable timeout.

For the control- and data-flow analysis, we leverage the implementation provided by the authors of Vandal [8]. In their paper, the authors evaluated the performance of Vandal and found that it would timeout for roughly 5 % of the smart contracts they used to assess it.

For these cases, it is not possible to evaluate the efficiency of Algorithm 2 and 3 as proposed in Chapter B, as they are dependent on the successful completion of this analysis step. Furthermore, possible issues and optimisations for the supportive analysis techniques such as the static analysis performed by Vandal are out of scope for this research. Therefore, we omitted the results from our experiments where Vandal was unable to provide analysis results within a timeout of 30 seconds.

6.2 Benchmarks

To evaluate the proposed algorithms, we design three benchmarks that each aim to cover a different aspect of *must-summary checking* performance. Specifically we measure the speed of the algorithms, as well as the percentage of summaries that they are able to re-validate (summary validation rate).

We identify the following three main factors for the evaluation of the proposed algorithms:

1. Which change categories can the algorithm handle?
2. How does the algorithm perform in real-world scenario's?
3. How does the algorithm perform when faced with compiler-introduced changes?

We formulate a benchmark to assess each of the topics mentioned above. Each of these benchmarks comprises a set of contracts for which there are one or more versions. For each unique contract, we will use the Mythril symbolic execution tool to generate symbolic summaries. For each contract in the benchmark, we evaluate which of the discovered summaries are valid between the present program versions.

6.3 Benchmark 1: Arbitrary changes

Chapter 3 describes a range of change types and possible instances thereof. In addition to demonstrating the general performance of must-summary checking algo-

rithms, it is beneficial to provide a detailed evaluation of the must-summary checking algorithms, and the change types they support. This benchmark achieves that by evaluating the summary checking results for a set of minimal smart contracts, that exhibit an instance of a single change type. With this approach, the benchmark takes a *testing* approach to the evaluation of must-summary checking algorithms. Thus, this benchmark reflects on the first factor discussed in the introduction of this section: ”Which change categories can the algorithm handle?”

Note that Algorithm 2 and Algorithm 3 enable re-use of summaries for syntactically changed code. At the same time, Algorithm 1 cannot deal with any program changes and is primarily beneficial for real-world projects where some contracts remain unchanged between versions. Therefore, this benchmark primarily provides an insight into the performance of the techniques applied in Algorithm 2 and Algorithm 3.

6.3.1 Formulation

The benchmark consists of a set of contracts that each have two versions, a base version and a changed version. This changed version is precisely equal to the base version, except for a single syntactic change. We demonstrate whether a summary checking algorithm covers the introduced type of program change by seeing whether all generated summaries for the base version are confirmed to be valid for the changed program.

The benchmark we constructed contains examples which introduce the following kinds of program changes (see Chapter 3 for extended definitions):

- Arithmetic change
- Effectless change
- Statement order change

Arithmetic changes

The category of arithmetic changes involves syntactic changes to arithmetic statements that are demonstrably equivalent using arithmetic normalisation.

Specifically we introduce a change as described in Fig. 6.3. A summary checking algorithm might use the commutativity of multiplication to show equivalence of the functions base and changed.

```
function BASE(x)
    a := 2 * x
    return a * 2
end function
function CHANGED(x)
    a := 4 * x              ▷ The second multiplication with 2 has been moved here
    return a
end function
```

Figure 6.3: Benchmark 1: Arithmetic change

Effectless change

To represent this category we add two cases that each represent one type of effectless change.

The first entry is described by Figure 6.4, which includes an effectless statement that has no effect in any code path.

Figure 6.5 describes the second entry, a change to a statement that only affects one code path. Specifically, we can say that if $x \leq 10$ the behaviour of the program remains unchanged, only the code path through the modified if statement is affected. Therefore, a summary checking algorithm should only report one invalid summary for this code.

```
function BASE(x)
    a := 0
    b := x
    a = b*2
    return a * 2
end function
function CHANGED(x)
    a := 0
    b := x
    a = b*2
    b = 0                                   ▷ This statement was added
    return a * 2
end function
```

Figure 6.4: Benchmark 1: Effectless change

```
function BASE(x)
    a := 0
    b := x
    if x > 10 then
        b += a
    end if
    return b
end function
function CHANGED(x)
    a := 10                          ▷ a now has a different constant value
    b := x
    if x > 10 then
        b += a
    end if
    return b
end function
```

Figure 6.5: Benchmark 1: Partially dead code

Statement order change

Figures 6.6 contains an example of the change type tested in this part of the benchmark. Changes to the order of statements do not necessarily change the behaviour of the code.

```
function BASE(x, y)
    a := y
    b := x
    return a * b
end function
function BASE(x, y)
    b := x
    a := y                           ▷ The definition of a and b have been switched
    return a * b
end function
```

Figure 6.6: Benchmark 1: Statement order change

Change Type	Valid Summaries	Total Summaries	Algorithm 1	Algorithm 2	Algorithm 3	Validation Rate
arithmetic change	5	5	0	5	-	100%
commutative change	5	5	0	5	-	100%
effectless change	5	5	0	5	-	100%
partially dead change	5	6	0	4	1	100%
statement order change	5	5	0	5	-	100%

Table 6.1: Results benchmark 1

6.3.2 Results

Table 6.1 shows the the results of the benchmark. Each entry in the table corresponds to a pair of smart contracts in the benchmark test set, a base version and a changed version.

The second column describes how many summaries are valid between the versions of the smart contract, whereas the third shows the total number of summaries generated using Mythril. Note that while the smart contracts have just a single code path on the Solidity level, they usually have more paths in their bytecode representations. Similar divergences in the number of code paths might exist for other source languages and their compilation targets.

The ensuing three columns describe how many summaries each algorithm was able to validate.

Since the algorithms provide increasing capabilities at the expense of increased computational cost, we run the algorithms in succession passing on summaries that an algorithm was not able to validate to the next. For example, each summary that Algorithm 2 was unable to validate, will be passed on and rechecked by Algorithm 3.

Note that for many of the cases in this benchmark, Algorithm 2 can successfully validate all summaries. For these cases, the “-” sign is used to denote when an algorithm is not applied.

The final column highlights the percentage of valid summaries that the algorithm was able to validate.

6.3.3 Discussion

We define the research question in Section 1.2.1 as follows: “How can we efficiently check must-summaries for smart contracts that have semantically preserving changes in the summarised code?”

This benchmark provides a partial answer concerning the ability of a summary checking algorithm to validate summaries for changed code. Specifically, it shows instances of the specific change types that the algorithms can effectively handle.

As is visible in the results, the algorithms cover all the changes included in this initial benchmark. The benchmark demonstrates both that the algorithms have the

capability of validating summaries for syntactically changed code, and a range of examples of the changes that the algorithm can effectively handle.

In our discussion on related work (see Chapter 4), we discuss a range of techniques aimed at program differencing and incremental analysis techniques.

While other program differencing approaches, such as those based on symbolic execution or verification condition generation, can determine the equivalence of these examples. Our approach, to our knowledge, is the only one able to achieve this by leveraging similarities between the original and new code of a program.

Furthermore, the results for the test case illustrated by Figure 6.5 confirm that the data-flow analysis used by Algorithm 3 is indeed effective for changes to partially dead code. Here, existing techniques [58] using program slicing would have proven insufficient.

6.3.4 Limitations

As mentioned above, this benchmark resembles a testing-based approach to evaluation, which introduces some limitations to the evaluation.

Firstly, the cases in the benchmark do not represent the full range of program changes that a summary checking algorithm could encounter. Instead, these are cases aimed at demonstrating the capabilities of the summary checking algorithms for particular examples. Furthermore, the ensuing two benchmarks provide a complimentary assessment of the algorithms by evaluating the test suite with test cases representative of the changes introduced in real-world projects.

Secondly, the results of the benchmark are insufficient to conclude that the algorithms can appropriately deal with all possible instances of the tested change types. Nevertheless, they show that the applied normalisation strategies are indeed capable of handling the included cases as expected. Furthermore, its results are beneficial for the comparison of summary checking algorithms, as the results highlight how our approach improves over alternative approaches. Additionally, future publications might use an extended version of this benchmark to demonstrate their contribution.

6.4 Benchmark 2: Real-world version increments

Where the previous benchmark demonstrates the capabilities of a summary checking algorithm on minimal examples, this benchmark will show the efficiency of summary checking problems when faced with changes that occur in non-arbitrary programs.

This benchmark evaluates the performance of the algorithms on real-world projects. Therefore, this algorithm allows evaluates the second factor discussed above: “How does the algorithm perform in real-world scenarios?”

6.4.1 Formulation

For the construction of this benchmark, we selected two well maintained smart contract projects: openzeppelin-solidity [62] and aragonOS [63]. There are three arguments why we have selected these projects.

Firstly, these two projects have an available long history of releases, which is not the case for most smart contract projects. Ethereum is relatively new compared to other execution platforms. Additionally, development teams regularly use a waterfall process to design a smart contract system. As a result, the published version history for most projects is sparse. AragonOS and openzeppelin-solidity are different because they provide a host of smart contract implementations, which have seen recurrent releases over time.

Secondly, both projects provide reference implementations of many of the standards in Ethereum [64]. Furthermore, they serve as libraries used by many other smart contract systems. For this reason, the code in these projects will feature the most common patterns used in smart contract development.

Finally, the selected projects are of non-trivial size. Some deployed smart contract systems are relatively small when compared to applications on traditional platforms. AragonOS and openzeppelin-solidity, however, feature various smart contract implementations, with non-trivial business logic.

To construct the benchmark, we take advantage of the version control systems used by aragonOS and openzeppelin-solidity to collect the history of the respective projects. Then, we compile the contracts in each available version using the toolchain configuration as provided in its respective project. Over the versions of the project, this toolchain configuration will also change, potentially introducing new compiler versions. These steps construe a database which encompasses the version history of compiled smart contracts from both aragonOS and openzeppelin-solidity, including both changes introduced by developers, and changes introduced by compilers.

6.4.2 Results

We first generated symbolic summaries for the bytecodes present in this benchmark, which took, on average, 8.5 seconds per summary. The benchmark implementation includes a timeout which limits the computation time per version-comparison to 16

minutes. In the subsequent execution of the benchmarks, the algorithms were able to run without error or timeout for 89% of the smart contracts. Table 6.2 presents the results of this benchmark.

The first column called “Target” provides an identification of the project for which a row is describing the results. As described above, these two projects are aragonOS and openzeppelin-solidity. The bottom row shows the accumulated results for both projects.

Following the initial column, is the column called “Total” which describes the total amount of summaries that comprise this benchmark.

The three subsequent columns describe the number of summaries that were determined valid using each algorithm. It is important to note that the algorithms run in order. More specifically, any summary which is determined valid will not be checked using any of the ensuing algorithms.

The next column describes the summary re-validation rate, which denotes the percentage of summaries that the algorithm could re-use between the different versions of the projects.

Lastly, the final two columns denote the average time taken to compute the equivalence of each summary and the average percentage of basic blocks that the normalisation algorithm was able to map using the basic block mapping algorithm (see Section 5.2.2).

Target	Total	Algorithm 1	Algorithm 2	Algorithm 3	Re-use Percentage	Average Time per Summary	Average percentage of mapped basic blocks
openzeppelin-solidity	4,215	576	2,519	1	73%	0.29 s	85%
aragonOS	14,987	10,698	2,600	5	88%	0.55 s	81%
Total	19,202	11,274	5,119	6	85%	0.50 s	83%

Table 6.2: Results benchmark 2

6.4.3 Discussion

Recall the research question established in Chapter 1: “How can we efficiently check must-summaries for smart contracts that have semantically preserving changes in the summarised code?”

In this benchmark, we evaluate the performance of the summary checking techniques on the versions released for real-world projects. The benchmark demonstrates the efficacy of summary re-use for Ethereum smart contracts.

The results show that the algorithms are effective at validating symbolic summaries. The algorithms were able to re-validate a significant portion of the symbolic summaries (73% and 88% for the two projects, respectively), spending on average 0.50 seconds on validation where summary generation took 8.5 seconds. Therefore,

using summary validation, rather than re-generating the summaries for unchanged code provides an order of magnitude speedup.

The results also show that the added capability of Algorithm 3 provides a limited improvement over Algorithm 2; this is an indication that within the domain of smart contracts, changes that only sometimes affect the execution are rare.

Finally, the results show an interesting difference between the summary re-validation rate of aragonOS and openzeppelin-solidity. Specifically, the number of summaries that Algorithm 1 determines valid is divergent. Recall that Algorithm 1 used a syntactic equivalence check to compute the possible validity of symbolic summaries. These results indicate that between the versions of aragonOS and openzeppelin-solidity, the latter had comparatively less contracts that were unaffected by any syntactic change.

6.4.4 Limitations

The following subsections will address the potential limitations of this benchmark.

Summary Scope

Symbolic executors may compute symbolic summaries at different scopes. For example, the benchmarks in this thesis handle summaries at the scope of ethereum transactions. Other summary checking implementations might operate at a different scope, such as that of function summaries. Decreasing the scope of a summary could decrease the difficulty of showing the correctness of that summary because it will less likely cover some changed piece of code. More fine-grained symbolic summaries might also increase the required computational resources, however. Evaluating the effect of symbolic summary scope on the *must-summary checking problem*, and finding an optimal scope is a topic for future work.

Domain

The implementation of the proposed algorithms and the design of the normalisation procedures aims to allow the re-use of symbolic summaries for ethereum smart contracts, which has a potential effect on the reproducibility of these results for alternative platforms.

Firstly, ethereum smart contracts potentially follow other architectural patterns when compared to alternative platforms.

For example, a best practice in smart contract development is to limit the business logic implemented in a smart contract. An off-chain (i.e. a non-blockchain)

application should implement non-critical business logic, and only critical components should be committed to the blockchain.

As a result, smart contracts might generally be less complex than alternative programs. However, this benchmark comprises two relatively large and non-trivial projects, aragonOS and openzeppelin-solidity.

Furthermore, differences in idiomatic programmatic structures are likely to be present for any platform. Therefore, it might be necessary to design normalisation procedures specific to other platforms to be able to replicate the demonstrated performance.

Language Level

A factor that influences the performance of a summary checking algorithm is the language level of the program for which the summaries are generated. In this thesis, we consider summaries generated for EVM bytecode, whereas other approaches might reason about languages such as Boogie [42] or a source-level language such as C.

Different challenges arise depending on this level. Take changes introduced during compilation (see Section 3.1.2 and Section 3.1.1), such changes are irrelevant when the analysis operates on the source code level.

Furthermore, the language might have beneficial properties for summary checking algorithms. For example, Algorithm 2 and Algorithm 3 (see Chapter 5) translate the EVM representation to an SSA form intermediate representation as its properties are beneficial and enable subsequent normalisation. Similarly, changes introduced in different languages potentially shape up to different challenges for summary checking and normalisation transformations.

Therefore, the demonstrated results do not necessarily reproduce for alternative platforms. Despite that, the results do indicate that summary re-use using the proposed algorithms would be beneficial for the analysis of EVM smart contracts. Furthermore, the results indicate that the evaluation of a program normalisation based summary checking approach is a promising avenue for future work (see Section 7.1).

6.5 Benchmark 3: Compiler Versions

The final benchmark evaluates the performance of summary checking algorithms when faced with changes caused by the use of different compiler versions. Such changes occur, for example, whenever a different or newer compiler is used for the compilation of the smart contracts. Newer versions might include support for new

language features, improved optimisation passes, or changed code synthesis procedures. These differences in the compiler potentially affect the generated bytecode for a software project.

This benchmark evaluates the must-summary checking algorithms when faced with changes introduced by differences in the compiler used, which matches the third question (see Section 6.2): “How does the algorithm perform when faced with compiler-introduced changes?”

6.5.1 Formulation

For the construction of this benchmark, we sample a set of thirty smart contract applications and compile them using different versions of the Solidity compiler. Thereupon, Mythril is used to generate symbolic summaries for each of the generated bytecodes. Subsequently, we evaluate the algorithms using the generated bytecodes and summaries for each of the smart contracts.

For the compilation, we selected eighteen versions of the Solidity compiler; specifically, the versions between 0.5.0 and 0.5.17. These capture all the changes that occurred within a single non-breaking release cycle of the Solidity compiler. The benchmark does not incorporate breaking releases of the Solidity since breaking changes might introduce the need for changes to the source code. However, the goal of this benchmark is the evaluation of the algorithms solely for the changes introduced by compilers. Furthermore, the new features introduced in breaking releases encourage rewrites and refactorings of source code. The effects of such changes are not covered when the only changing parameter is the compiler versions. Benchmark 2, on the other hand, does cover such changes as it tracks the changes of a real-world project where both the compiler and source can change.

The set of smart contracts have been selected randomly from the verified smart contracts available on Etherscan [65], specifically the contracts originally compiled with a 0.5.x version of the solidity compiler. The source of the smart contracts, Etherscan, is a block explorer which provides a user interface to inspect the state of the Ethereum blockchain. One of its features allows developers to upload their source code and link it to the code on the blockchain. Etherscan will use the Solidity compiler [29] to verify that compiling the provided source code will result in that bytecode. The set of verified contracts provide a sample of deployed Ethereum smart contract applications and their source code. Furthermore, the contracts in this dataset are the recipients of 72% of all Ethereum transactions [66]. As a result, the test suite samples from those contracts that get deployed, have available source code and see active use.

6.5.2 Results

After having generated symbolic summaries for this benchmark, taking on average 25 seconds per summary, we run the algorithms on the benchmark. In the benchmark, we introduce a timeout, limiting summary checking time for each version increment to ten minutes per contract. We saw that the algorithms were able to terminate without error or timeout for 76% of the smart contracts in the benchmark. The two tables, Table 6.3 and Table 6.4 describe the results of this benchmark.

Table 6.3 shows the global results for each of the different algorithms. The first column shows the total amount of summaries that were subject to the summary checking algorithms. The following three columns demonstrate how many of the summaries each algorithm was able to re-use. Similar to the previous benchmark these algorithms run in succession, passing on any unvalidated summaries to the next algorithm.

Table 6.4 provides a detailed overview of the summary re-use rate for the different Solidity compiler version changes. The first column of the table describes the change in compiler version, and the second column shows how many summaries the algorithm was able to compute as valid.

# Summaries	Algorithm 1	Algorithm 2	Algorithm 3	Re-use Percentage	Average Time per Summary
5099	0	4180	0	82%	4.29 s

Table 6.3: Results benchmark 3

6.5.3 Discussion

The results demonstrate that summary re-use would be advantageous when analysing a version of a program that has changed due to changes in a compiler. Specifically, the normalisation based technique proposed in Chapter 5 enables the re-use of 82% of summaries on average. Furthermore, the percentage of re-usable summaries reaches up to 100% for some of the version changes of the compiler (see Table 6.4). We see that the techniques can validate a summary in 4.24 seconds, while Mythril spends 25 seconds on average to generate a summary. Summary validation, therefore, provides a 4.6 times speedup over the regeneration of summaries for the contracts in this benchmark.

Interestingly, Table 6.3 demonstrates that Algorithm 2 is the algorithm that enables the re-use of all the summaries. The results follow from the nature of the changes in this benchmark.

We see that changes in the compiler introduce syntactic changes in the bytecode, which explains the lack of summaries validated using Algorithm 1. This algorithm

Solidity Compiler Version Change	Re-use Rate
0.5.0 → 0.5.1	0%
0.5.1 → 0.5.2	0%
0.5.2 → 0.5.3	98%
0.5.3 → 0.5.4	93%
0.5.4 → 0.5.5	22%
0.5.5 → 0.5.6	74%
0.5.6 → 0.5.7	100%
0.5.7 → 0.5.8	96%
0.5.8 → 0.5.9	100%
0.5.9 → 0.5.10	0%
0.5.10 → 0.5.11	99%
0.5.11 → 0.5.12	100%
0.5.12 → 0.5.13	100%
0.5.13 → 0.5.14	99%
0.5.14 → 0.5.15	100%
0.5.15 → 0.5.16	100%
0.5.16 → 0.5.17	100%
0.5.17 → 0.5.18	31%

Table 6.4: Results benchmark 3: Summary re-use with respect to compiler version

applies a trivial syntactic check, which is not efficacious when a compiler change has transpired.

Additionally, the results show that Algorithm 3 does not provide improved results over Algorithm 2. The additional analysis in Algorithm 3 allows for summary re-use in the presence of semantic changes that do not affect all executions. However, the changes present in this benchmark are all purely syntactic, originating from a change in the compiler used. As expected, the results in Table 6.3 show that the added capability of Algorithm 3 does not add improvements for those syntactic changes introduced by compilers.

6.5.4 Limitations

There are three principal limitations on the conclusions derived from these results.

Firstly, the benchmark considers just the Solidity compiler. Other compilers might introduce different types of changes, which the current techniques would not adequately handle. Notwithstanding, the results do show the efficiency of the proposed algorithms for changes introduced by the Solidity compiler.

Secondly, the results show that there is a small selection of compiler versions between which the algorithm was not able to re-use a significant amount of summaries. For these specific version increments, the compiler introduces change types that are currently not covered by the normalisation strategies. As mentioned above, designing effective normalising transformations that enable the handling of an increased amount of change types is a topic for future work (see Section 7.1). Even though extensions to the covered change types are still possible, the current algorithm proves effective for a majority of the tested Solidity compiler version increments.

Thirdly, the algorithms only successfully terminate for 76% of the smart contracts. This success rate is remarkably lower than that of Benchmark 2. Nevertheless, the algorithm shows promising results for a significant amount of the evaluated smart contracts.

Chapter 7

Conclusion

In this research, we have designed an approach which can determine the validity of symbolic summaries between the versions of a program. Determining which summaries are still valid for new versions of a program allows a symbolic executor to re-use those summaries and thus incremental symbolic execution.

During our analysis of the related work, we found that program differencing algorithms rarely considered syntactic program changes without a semantic effect.

We performed a study of the program changes and their origins for Ethereum smart contracts. Based on this analysis, we propose a novel normalisation based summary checking algorithm which permits the re-use of symbolic summaries in the presence of syntactic changes.

Furthermore, we also introduce a new approach leveraging data-flow analysis to permit re-use of symbolic summaries in the presence of semantic changes which do not affect a particular summary. This method extends upon existing program differencing techniques that use program slicing [58], which is more restrictive (see 5.3).

In our evaluation, we found that the techniques permit on average 85% of symbolic summaries to be re-used between the versions of real-world projects (see Section 6.4). Furthermore, on contracts with changes introduced by a compiler, we see that on average, 82% of summaries were able to be re-used (see Section 6.5). Lastly, we saw that the algorithm was able to normalise a range of program changes effectively (see Section 6.3).

In the benchmarks, we saw that summary re-use would provide an order of magnitude speed improvement over the re-generation of symbolic summaries, demonstrating the efficacy of summary re-use and incremental symbolic execution.

In summary, the main contributions of this thesis are as follows:

- The formulation of a novel program normalisation based summary checking algorithm, that enables incremental symbolic execution for smart contracts with syntactic changes.

- The formulation of a data-flow analysis based summary checking algorithm, allowing summary re-use in the presence of semantic changes.
- A study of the origins and types of changes that occur between the versions of smart contracts.
- The introduction of a novel data-flow based heuristic used for discovering mapping between basic blocks.
- An extension of the smart contract analysis tool Mythril with support for symbolic summaries.

7.1 Future Work

This section provides an overview of our recommendations for future work, and is structured as follows. First, we discuss the future work related to the currently applied normalisation techniques (see Section 5.1). Then, we discuss the different research topics related to change categorisation and the extension of the normalisation to deal with more change categories. Finally, we discuss the need for comprehensive evaluation of incremental analysis techniques for different platforms.

7.1.1 Program Normalisation

In Section 5.2 we introduce several normalisation techniques. Here, we discuss potential extensions to these normalisation procedures and techniques.

Arithmetic Normalisation

One of the normalising transformations applied in Algorithm 2 (see Section 5.2) rewrites arithmetic expressions to make them more amenable for syntactic equivalence checking. The arithmetic normalisation is implemented with an abstract rewrite system, using some trivial rewrite rules. However, future work is needed to formulate and evaluate an extensive approach for arithmetic normalisation within program normalisation and incremental analysis. Moreover, the evaluation of existing arithmetic normalisation and simplification approaches within this context would enable the future design of arithmetic normalisation algorithms for use within program normalisation settings. Techniques, such as the arithmetic simplification using off-the-shelf SMT solvers [24] could improve the percentage of summaries that can be computed as valid.

Mapping Between Basic Blocks

In Section 4.1.1, we discussed prior work related to reverse engineering, a research area with much overlap with incremental program analysis, where the core challenge is as follows: ”How can information be re-used between versions of a program?”.

While different requirements arise from the application area, be it formal methods or reverse engineering, some techniques apply to both areas. For example, one of the steps in the normalisation phase of Algorithm 2 requires the discovery of a map between the basic blocks of two versions of a program. We leverage a heuristic-based approach, similar to that introduced Dullien et al. [44], that incrementally discovers such a mapping.

Further evaluation of the application of methods in reverse engineering to incremental software analysis is promising. Improvements to the accuracy, completeness and efficiency of algorithms establishing a mapping between basic blocks apply to the approach formulated in this thesis and are attractive topics for future work.

Language Design

Section 5.2 discusses the possible effects that an intermediate representation has on program differencing. Exploring the benefits of different intermediate representations for program differencing might allow more efficient implementations of a normalisation based summary checking algorithm.

In particular, we identify the E-PEG representation as proposed by Tate et al. [67]. While initially designed as a representation for use within a compiler, this representation provides several benefits for a summary checking algorithm. In their paper Tate et al. already demonstrate the capabilities of the algorithm for equivalence checking of compiler-generated machine code. An extension of their approach with normalising transformations is promising for the *must-summary checking problem*.

7.1.2 Change Categories

The algorithms, as proposed in Chapter 5, currently apply a range of normalisation strategies to enable the normalisation of a selection of change types. In addition to improving the presently applied normalisation strategies, one might also extend the normalisation strategies. Such an extension broadens the range of change types that a normalisation based summary checking algorithm could deal with.

In the following subsections, we identify three topics for future work related to extending the range of handled change types.

Change Categorisation

An extensive categorisation of the changes that might occur between the versions of a program is essential for the design of algorithms to deal with them. Chapter 3 describes an initial study and categorisation of the program changes that occur in Solidity smart contracts. While existing approaches exist for the categorisation of program changes [49], these approaches handle general categorisation. Future work could explore the formulation of an automated approach for identifying program change types which preserve the semantics of a program.

Such a categorisation provides two principal benefits in relation to the technique formulated in this thesis.

Firstly, an extensive overview of different change types and the frequency at which they occur enables the targeted design and selection of normalisation procedures.

Secondly, such a study might inform the formulation of a benchmark providing detailed information on the capabilities of a summary checking algorithm, similar to the insight provided by Benchmark 1 (see Section 6.3). The changes in such a benchmark would be representative of the changes introduced in real-world projects; therefore, it would provide an indication as to the real-world performance of summary checking algorithms.

A technique to compile such a dataset might use symbolic execution for the identification of purely syntactic program changes, similar to the techniques applied by Person et al. and Backes [21], [41] (see Section 4.1.1).

Extended Change Categories

The normalisation strategies used in Algorithm 2 and Algorithm 3 cover some of the changes identified in Chapter 3, while others remain unsupported. In future work, one can explore the development of normalisation procedures to handle more change types.

The following are some of the change types which future normalisation strategies could handle.

First, the current normalisation manages changes that do not modify the control-flow of a program (see Section 3.2). Future work can explore strategies which allow summary re-use when control flow is changed (see Section 3.2.5).

Another extension on the normalisation techniques could follow from the further identification of change types that preserve semantics.

Architectures other than the EVM, potentially pose different challenges to program normalisation and summary checking (see Section 6.4.4). In future work, one might explore alternative normalisations to manage these challenges.

In addition to the development of normalisation procedures, an appealing topic for future work is the formulation of an optimised selection of normalisation strategies. An improved selection could improve computational performance and the amount of re-usable summaries.

Changes Introduced by Compilers

In Section 3.1.2, we introduced the compiler as an actor that can introduce changes between analysis runs of a program. Later, in the evaluation of the summary checking algorithms (see Section 6.5), we showed summary checking performance in the face of changes introduced by different compiler versions. However, some of the compiler versions introduced yet unsupported changes. Similarly, compilers for other languages can introduce unsupported change types. In future work, one could design normalisation transformations targeting these particular change types.

Note, that it is possible to enable normalisations based on the observed compiler version change. Therefore, it is possible to develop and optimise normalisations to a specific version change without potentially impacting general performance.

7.1.3 Improved Evaluation

During the evaluation of the proposed techniques (see Chapter 6), and the comparison with related work, we noted that different languages pose different challenges for summary checking algorithms. Some techniques might be universally applicable, while others are uniquely efficient for a single platform.

The formulation of an extensive benchmark suite that demonstrates the capabilities and efficiency of summary checking and incremental analysis techniques for different languages and platforms would be beneficial. Such a benchmark allows for the comparison of the existing and future techniques designed and applied to different platforms. Furthermore, extensive evaluation of this kind improves the future development of normalisation procedures and incremental analysis approaches.

Bibliography

[1] P. Godefroid, S. K. Lahiri, and C. Rubio-González, "Statically validating must summaries for incremental compositional dynamic test generation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6887 LNCS, pp. 112–128, 2011.

[2] "Ethereum: A secure decentralised generalized transaction ledger." [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[3] "The DAO Attacked: Code Issue Leads to $60 Million Ether Theft - CoinDesk." [Online]. Available: https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft

[4] "Parity Bug Security Alert." [Online]. Available: https://www.parity.io/security-alert-2/

[5] "Batch overlflow vulnerability - CVE-2018-10299." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10299

[6] "Mythril." [Online]. Available: https://github.com/consensys/mythril

[7] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018.

[8] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *CoRR*, 2018.

[9] J. J. Honig, M. H. Everts, and M. Huisman, "Practical mutation testing for smart contracts," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer International Publishing, 2019, pp. 289–303.

[10] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, 2018.

[11] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. IEEE Computer Society, 2010, pp. 317–331.

[12] "Swc-101: Integer overflow and underflow." [Online]. Available: https://swcregistry.io/docs/SWC-101

[13] "Swc-105: Unprotected ether withdrawal." [Online]. Available: https://swcregistry.io/docs/SWC-105

[14] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. ACM, 1977, pp. 238–252.

[15] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[16] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan, "Incremental state-space exploration for programs with dynamically allocated data," p. 291, 2008.

[17] G. Yang, S. Khurshid, S. Person, and N. Rungta, "Property differencing for incremental checking," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014, pp. 1059–1070.

[18] G. Yang, S. Khurshid, and C. S. Păsăreanu, "Memoise: A tool for memoized symbolic execution," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 1343–1346.

[19] P. Godefroid, "Compositional dynamic test generation," *ACM SIGPLAN Notices*, vol. 42, no. 1, p. 47, 2007.

[20] K. R. M. Leino and V. Wüstholz, "Fine-grained caching of verification results," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9206, pp. 380–397, 2015.

[21] S. Person, G. Yang, N. Rungta, and S. Khurshid, “Directed incremental symbolic execution,” *SIGPLAN Not.*, vol. 46, no. 6, pp. 504–515, 2011.

[22] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, “eXpress: Guided Path Exploration for Efficient Regression Test Generation,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA ’11. ACM, 2011, pp. 1–11.

[23] P. D. Marinescu and C. Cadar, “KATCH: High-Coverage Testing of Software Patches,” in *European Software Engineering Conference / ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013)*, 2013, pp. 235–245.

[24] L. de Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Springer Berlin Heidelberg, 2008, pp. 337–340.

[25] P. Godefroid, N. Klarlund, and K. Sen, “Dart: Directed automated random testing,” *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, 2005.

[26] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks,” in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM’98. USENIX Association, 1998, pp. 5–5.

[27] P. B. Schneck, “A survey of compiler optimization techniques,” in *Proceedings of the ACM Annual Conference*, ser. ACM ’73. ACM, 1973, pp. 106–113.

[28] P. Clauss and B. Meister, “Automatic memory layout transformations to optimize spatial locality in parameterized loop nests,” *SIGARCH Comput. Archit. News*, vol. 28, no. 1, pp. 11–19, 2000.

[29] “Solidity, the Contract-Oriented Programming Language.” [Online]. Available: https://github.com/ethereum/solidity

[30] N. Savers, “EIP 1013: Hardfork Meta: Constantinople.” [Online]. Available: https://eips.ethereum.org/EIPS/eip-1013

[31] T. Mens and T. Tourwé, “A survey of software refactoring,” *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, 2004.

[32] A. V. Aho and J. D. Ullman, *Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing)*. Addison-Wesley Longman Publishing Co., Inc., 1977.

[33] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, “Exe: Automatically generating inputs of death,” in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS ’06. ACM, 2006, pp. 322–335.

[34] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. USENIX Association, 2008, pp. 209–224.

[35] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, “SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs,” in *Computer Aided Verification*. Springer Berlin Heidelberg, 2012, pp. 712–717.

[36] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, “Differential symbolic execution,” in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT ’08/FSE-16. ACM, 2008, pp. 226–237.

[37] D. Jackson and D. Ladd, “Semantic diff: A tool for summarizing the effects of modifications,” in *Proc. Int’l Conf. Software Maintenance (ICSM)*. IEEE Computer Society Press, 1994.

[38] B. Godlin and O. Strichman, “Regression verification,” in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC ’09. ACM, 2009, pp. 466–471.

[39] O. Sery, G. Fedyukovich, and N. Sharygina, “Incremental upgrade checking by means of interpolation-based function summaries,” in *Twelfth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2012.

[40] G. Fedyukovich, O. Sery, and N. Sharygina, “evolcheck: Incremental upgrade checker for c,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2013, pp. 292–307.

[41] J. Backes, S. Person, N. Rungta, and O. Tkachuk, “Regression verification using impact summaries,” in *Model Checking Software*, E. Bartocci and C. R. Ramakrishnan, Eds. Springer Berlin Heidelberg, 2013, pp. 99–116.

[42] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, “Boogie: A modular reusable verifier for object-oriented programs,” in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Springer Berlin Heidelberg, 2006, pp. 364–387.

[43] “Bindiff.” [Online]. Available: https://www.zynamics.com/bindiff.html

[44] T. Dullien and R. Rolles, “Graph-based comparison of executable objects (english version),” *SSTIC*, vol. 5, no. 1, p. 3, 2005.

[45] M. Bourquin, A. King, and E. Robbins, “Binslayer: accurate comparison of binary executables,” in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013, pp. 1–10.

[46] J. Munkres, “Algorithms for the assignment and transportation problems,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[47] J. Ming, M. Pan, and D. Gao, “ibinhunt: Binary hunting with inter-procedural control flow,” in *International Conference on Information Security and Cryptology*. Springer, 2012, pp. 92–109.

[48] D. Gao, M. K. Reiter, and D. Song, “Binhunt: Automatically finding semantic differences in binary programs,” in *International Conference on Information and Communications Security*. Springer, 2008, pp. 238–255.

[49] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, “Change distilling: Tree differencing for fine-grained source code change extraction,” *IEEE Transactions on software engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[50] M. Egele, M. Woo, P. Chapman, and D. Brumley, “Blanket execution: Dynamic similarity testing for program binaries and components,” in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 303–317.

[51] B. S. Baker, U. Manber, and R. Muth, “Compressing differences of executable code,” in *ACMSIGPLAN Workshop on Compiler Support for System Software (WCSS)*. Citeseer, 1999, pp. 1–10.

[52] “Diaphora.” [Online]. Available: https://github.com/joxeankoret/diaphora

[53] “Turbodiff.” [Online]. Available: https://www.coresecurity.com/corelabs-research/open-source-tools/turbodiff

[54] T. Szabó, S. Erdweg, and M. Voelter, “IncA: A DSL for the Definition of Incremental Program Analyses,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. ACM, 2016, pp. 320–331.

[55] O. Sery, G. Fedyukovich, and N. Sharygina, “Interpolation-based function summaries in bounded model checking,” in *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing*. Springer-Verlag, 2012, pp. 160–175.

[56] G. Fedyukovich, O. Sery, and N. Sharygina, “Flexible SAT-based framework for incremental bounded upgrade checking,” *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 5, pp. 517–534, 2017.

[57] B.-C. Rothenberg, D. Dietsch, and M. Heizmann, “Incremental verification using trace abstraction,” in *Static Analysis*, A. Podelski, Ed. Springer International Publishing, 2018, pp. 364–382.

[58] D. Binkley, “Using semantic differencing to reduce the cost of regression testing,” in *Proceedings of the Conference on Software Maintenance*, vol. 92. Citeseer, 1992, pp. 41–50.

[59] K. D. Cooper and L. Torczon, “Chapter 9 - data-flow analysis,” in *Engineering a Compiler (Second Edition)*, second edition ed., K. D. Cooper and L. Torczon, Eds. Morgan Kaufmann, 2012, pp. 475 – 538.

[60] K. Weiss and J. Schütte, “Annotary: A concolic execution system for developing secure smart contracts,” in *Computer Security – ESORICS 2019*, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds. Springer International Publishing, 2019, pp. 747–766.

[61] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, “Madmax: surviving out-of-gas conditions in ethereum smart contracts,” *PACMPL*, vol. 2, pp. 116:1–116:27, 2018.

[62] “openzeppelin-solidity.” [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-solidity

[63] “aragonOS.” [Online]. Available: https://hack.aragon.org/docs/aragonos-intro.html

[64] “Ethereum Improvement Proposals.” [Online]. Available: https://eips.ethereum.org/

[65] “Etherscan.” [Online]. Available: https://etherscan.io/

[66] G. A. Oliva, A. E. Hassan, and Z. M. J. Jiang, “An exploratory study of smart contracts in the ethereum blockchain platform,” *Empirical Software Engineering*, pp. 1–41, 2020.

[67] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, “Equality saturation: A new approach to optimization,” *SIGPLAN Not.*, vol. 44, no. 1, pp. 264–276, 2009.