TU WIEN Informatics

Finding inconsistency bugs in Solidity smart contracts

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Robin Knoll, BSc
Matrikelnummer 11828955

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Maria Christakis
Mitwirkung: Dr. Valentin Wüstholz
Dipl.-Ing. Christoph Hochrainer, BSc

Wien, 31. März 2025

Robin Knoll	Maria Christakis

TU WIEN Informatics

Finding inconsistency bugs in Solidity smart contracts

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Robin Knoll, BSc
Registration Number 11828955

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Maria Christakis
Assistance: Dr. Valentin Wüstholz
Dipl.-Ing. Christoph Hochrainer, BSc

Vienna, March 31, 2025

Robin Knoll	Maria Christakis

Erklärung zur Verfassung der Arbeit

Robin Knoll, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 31. März 2025

Robin Knoll

Acknowledgements

I would like to express my gratitude to my supervisors for their continued support and guidance during the work on this thesis. Thank you to Maria Christakis for the great advice, relentless positive energy and the swiftest feedback latency I have ever experienced from a professor. My gratitude also goes to Valentin Wüstholz for sharing his expert knowledge of all things surrounding Solidity and smart contracts and to Christoph Hochrainer for his advice on this topic.

Finally, I would like to thank my family, my friends and my girlfriend. Their collective support and encouragement throughout all my studies always pushed me forward, I could not have done it without you!

Kurzfassung

In Solidity entwickelte Smart Contracts werden dazu verwendet, große Mengen an Geldwert auf der Ethereum Blockchain zu verwalten. Das macht diese Programme zu einem häufigen Ziel für Angriffe. Statische Codeanalyse kann dabei helfen, Bugs in Smart Contracts zu finden und Schwachstellen zu verhindern. Im Besonderen sind Inkonsistenzen im Code eine Art von Fehler, der mit statischer Codeanalyse gefunden werden kann.

Diese Diplomarbeit untersucht die Verwendung von Inkonsistenzmustern, um Fehler in Solidity Smart Contracts zu finden. Acht solche Muster, die speziell für Solidity relevant sind, werden ausgewählt und zwei statischen Codeanalyseprogrammen implementiert. Das erste dieser Programme ist Semgrep, ein Framework zur Mustererkennung in Code. Wir zeigen, dass Semgrep stark darin ist, schnell neue Inkonsistenzmuster zu implementieren, und dass es für mehrere solche Muster effektive Codeanalyse bietet.

Wir demonstrieren jedoch auch, dass Semgrep gewisse Einschränkungen in seiner Unterstützung für Solidity aufweist. Um diese Einschränkungen zu überwinden, stellen wir SIA vor, den Solidity Inconsistency Analyzer. SIA übersetzt Solidity Quellcode in Datalog Fakten und verwendet eine leistungsstarke Datalog Engine, um den Code auf Inkonsistenzmuster zu überprüfen. Dies ermöglicht es dem Programm, Vorfälle aller ausgewählten Inkonsistenzmuster zu finden und den funktionsübergreifenden Kontrollfluss auf einem höheren Niveau als Semgrep zu verstehen.

Wir zeigen, dass beide statischen Analysewerkzeuge in der Lage sind, Inkonsistenzen in realen Smart Contracts zu finden, und heben die unterschiedlichen Eigenschaften der beiden Werkzeuge hervor.

Abstract

Solidity smart contracts are used to manage large amounts of monetary value on the Ethereum block chain, making them a frequent target for attacks. Static code analysis tools can help to discover bugs in these programs and prevent vulnerabilities. In particular, inconsistencies in code are a class of bugs that can be detected by static code analysis.

This paper explores the use of inconsistency patterns to find bugs in Solidity smart contracts. We select eight patterns that are specifically relevant for Solidity and implement the patterns in two static analysis tools. The first tool is Semgrep, a framework for pattern matching in code. We show that Semgrep is useful for swiftly developing new inconsistency patterns and, for a number of patterns, can provide effective code analysis. However, we also demonstrate that Semgrep has limitations in its support for the Solidity language. To address these limitations, we present SIA, the Solidity Inconsistency Analyzer. SIA translates Solidity code to Datalog facts and uses a high-performance Datalog engine to analyze the code for inconsistency patterns. This enables the tool to find instances of all the selected inconsistency patterns and comprehend interprocedural control flow to a higher level than Semgrep. We show that both static analyzers are able to find inconsistencies in real world Solidity programs and highlight the different characteristics of the two tools.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Solidity is a programming language that has its main use in creating smart contracts which are executed on the Ethereum block chain [sol]. Due to the financial nature of block chain technology, many monetary transactions are facilitated using Solidity contracts. Estimates by the consulting firm McKinsey concluded that block chain technologies had a market capitalization of around 150B$ in 2018 [BOP+19].

Because of the large amount of monetary value that is managed by Solidity contracts, they are often tested and audited extensively to prevent misbehavior. Permissionless block chains like Ethereum offer a large attack surface for contracts running on them as anyone can interact with them. Security misconfigurations and other bugs can lead to the loss of large amounts of funds. An example of such an incident occurred in 2017 in the smart contracts of a wallet provider called Parity [Bro17]. A bug allowed attackers to take over ownership of Ethereum wallets. The vulnerability led to assets that may have been worth up to 280M$ being frozen and unavailable to their owners.

Another example of an important incident due to bugs in Solidity contracts was a vulnerability in a contract managing "The DAO" [Pra22]. DAO stands for "Decentralized Autonomous Organization". DAOs are types of smart contracts that are designed to coordinate decisions within their stakeholders. "The DAO" was one of the first of these organizations and was supposed to act as an investment fund. Around 3 months after its creation, bad actors found a reentrancy vulnerability in the contract. The exploit led to the theft of ETH with a value of around 150M$ at the time. This event was so significant that it eventually led to a fork in the Ethereum block chain [Fou24].

Code analysis tools can help to discover bugs like these early, before they become a problem. The code can be analyzed from multiple perspectives. Static code analysis checks programs without running them. This usually involves some form of pattern

matching or parsing the abstract syntax tree to retrieve information about the program structure. Dynamic code analysis may instrument the code and collect data during runtime. Both methods have found success in identifying bugs in programs [Bin07]. In particular, inconsistencies in the code are likely candidates for finding bugs [ECC01]. An inconsistency is any contradiction in what the programmer believes to be true about the state of the program.

A classic example for this type of bug is the dereference of a variable that is then followed by a null check. In this situation, the programmer believes that the value of the variable is different from null at the point of dereference, otherwise they would not dereference it. However, at the later point when the variable is checked for null, it seems that it is possible the value may be null.

There are two possible causes for this issue. If the null check is unnecessary and the value can not be null, it should be removed. In case the check is necessary, it should be moved to before the dereference. Either way, something about the code has to be changed and the programmer should be notified about this.

Improvements to code and bug fixes usually happen incrementally. Changes are reviewed by a colleague or an analyzer finds an issue and the affected code is changed when the programmer is made aware. A shorter feedback loop tends to lead to higher quality code.

A popular approach is to run automatic code analysis in build pipelines. Using this system, issues are only found when code is pushed to the central repository. It can take anywhere from a few minutes to a few hours until the programmer receives feedback on their changes. The author may even ignore the results as their focus has already shifted to other tasks.

A better approach is to run analysis live while the code is being written. This is possible by integrating the analyzer into a developer's IDE. The programmer receives immediate feedback on any changes and can improve on issues much quicker.

For this to be possible, the analyzer needs to run continuously. This fact requires the analysis to be fast and efficient. Otherwise, continuous analysis would take up too many computational resources or delay feedback.

1.2 Problem statement and Contributions

Bugs in Solidity contracts can have catastrophic consequences. Preventing and finding them is a big priority in this space. In particular, inconsistency patterns are a class of issues that point to necessary changes by the authors of smart contracts. A noteworthy property of inconsistency patterns is that the ground truth does not necessarily have to be known to point out issues. The performant analysis of smart contracts to find such inconsistency bugs can help to decrease the feedback loop for programmers and auditors. This may result in an increase in code quality, and thereby lower the loss of funds through bugs in smart contracts.

Currently available tools for static analysis of Solidity smart contracts often do not include support for inconsistency patterns. Frameworks such as Semgrep[1] allow users to define custom patterns. An approach using Semgrep to define inconsistency patterns is evaluated. Additionally, this work offers a static analysis tool called SIA, the Solidity Inconsistency Analyzer, for finding these types of issues.

Concretely, the key contributions of this work include:

- We identify the key inconsistency patterns affecting Solidity smart contracts. This is described in chapter 4.1.
- Semgrep, a framework implementing high-level pattern matching in programs, is instantiated for inconsistency patterns. Details can be found in section 5.1.
- We implement an extensible, performant static code analyzer called Solidity Inconsistency Analyzer to automatically look for inconsistency patterns, as detailed in section 5.2.
- A comparison of Semgrep with SIA is made. The evaluation can be found in chapter 6.

[1] Semgrep: https://semgrep.dev/

CHAPTER 2

Background

2.1 Inconsistency Patterns

Inconsistency patterns can be used as a way of finding program errors without knowing the exact rules of a program [ECC01]. Inconsistencies are contradictory beliefs about the state of a program. In this context, a belief is a fact that the code implies.

For example, dereferencing a pointer indicates a belief that that pointer is not null, as it would otherwise result in a null pointer dereference. Using a variable as the divisor in a division implies that the programmer believes that value to be something other than zero, as the alternative result would be an error due to the division by zero.

A conflicting belief in these examples might be a check for null or zero after the dereference or division. A check only makes sense if there is any uncertainty about a value.

Eugler et al. [ECC01] differentiate between "must" beliefs and "may" beliefs. "Must" beliefs are known implications from a certain behavior. A dereferenced pointer shows a belief that a pointer must have a value and must not be null. On the other hand, "may" beliefs are more unreliable and akin to heuristics.

A pairing of two function calls that is observed at the beginning and end of many function implementations may indicate a rule that the two functions should always be called together. For example, this heuristic would be valid on functions that open files and close them again at the end of the function. However, a more general definition of this rule, where functions that are *often* paired together *always* should be paired together, would most likely result in many false positives.

An invalid application of this rule might look like this: Let us think of functions that save data to a database, like in a simple CRUD application. Most of the functions that create entries might pair calls like *validate_data()* and *save_data()* together. We might infer a rule where if *validate_data()* is called, *save_data()* should also be called. However, the

rule might be invalid if the application offers functionality to validate some data and return the result of the validation to the user. The functions that only validate the data may call *validate_data()*, but then never execute a call to *save_data()*. These functions would be flagged by our previously defined rule, even though the validating functions may be implemented correctly. The rule would show a lot of false positives and prove unreliable.

Therefore, "may" beliefs require either manual approval or should be used in combination with a statistical analysis of the occurrences.

Inconsistencies in code can have many reasons. Copying code can lead to different ways of doing the same thing. Inexperience of developers is also a common reason, in case a developer is new to a language, they may not be aware of best practices or understand the functionality of language features. Changes to the language or libraries between different versions can also lead to inconsistencies, in case existing code is not updated and only extended with code using new features or the language version is simply upgraded without considerations for changes. For many developers, consistent style is also simply not an important factor of software development.

The inconsistencies that can be found specifically in Solidity code are described in detail in 4.1.

2.2 State of the art

A number of solutions already exist in the field of static analysis of Solidity smart contracts. The input to these static analyzers can differ. Some tools use EVM byte code as their input, while others analyze the Solidity source code. A popular approach is to use Datalog queries for the efficient analysis of code. There are also differences in the language used to describe the patterns of interest. Several tools define the patterns directly as Datalog queries, while others implement an easier to understand DSL that is translated to Datalog.

2.2.1 Semgrep

Semgrep[1] is a framework for defining and finding patterns in programs. The tool's main focus is on security issues and it is used for SAST. It offers support for multiple programming languages, including experimental support for Solidity. It comes with a couple of commonly used predefined rules for Solidity, but allows developers to add their own patterns using a syntax similar to the target code in a YAML format.

Semgrep's name comes from "Semantic grep", with "grep" being abbreviated from "global regular expression search and print"[Nag].

Semgrep does not compile the program, but it has some understanding of the AST. It parses the code and the input patterns and matches them, running quickly and efficiently.

[1]Semgrep: https://semgrep.dev/

While this is implemented for other languages, for Solidity, it can not follow data flow or understand some of the language constructs. This limits the power of the tool to a certain degree, but keeps pattern definition simple and easy to get started.

Semgrep is used in section 5.1 to define some of the inconsistency patterns and is run on a number of contracts.

As of December 2024 [O'M], the tool offers the free and publicly available version named "Community Edition" without the officially maintained rule set. It still allows the definition of custom rules and reuse of rules defined by other community members. Only the paid version offers support and updates on the core rule set.

2.2.2 Securify

Securify [TDD+18] is a static code analyzer for Ethereum smart contracts, developed by the SRI Lab[2] at ETH Zürich. It does not use Solidity source code as input, instead it transforms EVM byte code into a static-single assignment form. From this representation, it can infer semantic facts like data flow and control flow. Those semantic facts are translated into Datalog facts. Using byte code as an input allows it to analyze smart contracts written in other languages than Solidity as well, however some more language specific patterns and Solidity language semantics can not be checked for. Securify defines its patterns using a domain-specific language that is translated to Datalog queries. This method of implementing patterns allows its users to more easily extend the rule set, as the DSL is easier to understand than Datalog. Securify requires manual installation of solc, the Solidity compiler, and utilizes Java as its implementation language. Soufflé is used as the Datalog engine.

The original tool has been deprecated as of January 2020. The researchers have released Securify v2, with notable changes including a change to operation on Solidity source code instead of EVM byte code. The implementation language was also switched to Python.

The change of using Solidity source code as an input now allows Securify 2 to include patterns that execute on the source code, abstract syntax tree, the intermediate representation or the control flow graph. In total, the tool comes with 37 patterns. None of the patterns are inconsistency patterns. Securify v2 still uses Datalog queries for its analysis and facilitates the Soufflé engine[3].

2.2.3 Ethainter

Ethainter [BGL+20] presents a static code analyzer with a focus on detecting composite information flow violations, mainly for authentication issues. These problems involve the escalation of tainted data and could allow attackers to execute actions like becoming the owner of a contract or initiating its destruction.

[2]SRI Lab ETH Zürich: https://www.sri.inf.ethz.ch/
[3]Soufflé: https://souffle-lang.github.io/

The tool is based on the Gigahorse [GBSS19] tool chain. Gigahorse decompiles EVM byte code into an intermediate representation. It also has a feature to pull deployed contracts from the chain an decompile them. Ethainter is therefore able to perform static code analysis on any real-world contract that is deployed to the Ethereum block chain. The intermediate representation that is received as an output from Gigahorse has a static-single assignment form, it is translated further into Datalog facts and queried for the patterns of interest using the Soufflé Datalog engine[4]. The patterns are also implemented directly in Datalog. Ethainter does not implement any inconsistency patterns.

2.2.4 SmartCheck

SmartCheck [TVI+18] is another static code analyzer that analyzes Solidity source code. However, instead of translating to Datalog as the two previously described tools, it uses ANTLR[5] to parse the Solidity code and then uses XML as an intermediate representation. Problematic patterns are then found using an XPath[6] checker. The main reason for using XML as the representation for querying the code structure is the easy extensibility and the expressiveness of of XPath patterns.

SmartCheck supports a total of 21 patterns, 8 of which are security issues. Due to the more simple translation method, data flow and control flow are harder to check for, but the tool is also able to find some reentrancy issues. Most of the checked issues are general best practices, none of the patterns are inconsistency patterns.

2.2.5 Soufflé

Datalog is declarative programming language [CGT89]. It is syntactically based on Prolog and used for logic programming, also finding application as a data query language. Soufflé is a high-performance Datalog engine [JSS16]. It uses a dialect of Datalog as its language and enables the transformation of Datalog programs into optimized C++ code that can be further compiled to native binaries. This approach is a significant performance advantage over other Datalog implementations that interpret the code. Soufflé's execution algorithm is also highly optimized, offering numerous performance enhancing features like parallel execution or automatic indexing of data.

Due to its performance benefits, Soufflè in particular is a popular choice for code analyzers and other applications that rely on the fast processing of large data sets. Its integration with databases and support for modularity through components make it developer-friendly, making relatively quick development of programs using the framework possible. The project is also open-source and freely licensed, allowing developers and researchers to use it without restrictions.

[4]Soufflé: https://souffle-lang.github.io/
[5]ANTLR: https://github.com/antlr/antlr4/
[6]XPath: https://developer.mozilla.org/en-US/docs/Web/XPath/

CHAPTER 3

Overview

3.1 Reasons for bugs in Solidity

Solidity smart contracts are not easy to develop. The language has some pitfalls that may lead unfamiliar developers to make mistakes. Solidity is also very different in how the code is executed to other languages. Smart contracts are publicly visible to anyone and any miner may execute the contracts. This shift in perspective leads to confusion, for example by misconceptions about the "private" keyword [Roz23]. Private variables in Solidity are still readable and visible to anyone executing the contract, the keyword just protects the data from being accessible to other contracts.

In addition, the language has changed a lot over the last few years. Best practices for one version of Solidity may become obsolete, new features are added or old features deprecated. It can be hard to keep an overview of the behavior of the language and unexpected consequences tend to occur.

For example, prior to Solidity 0.8.0, arithmetic operations that would result in values outside of the defined range would overflow or underflow without throwing an error[1]. As this behavior was unwanted by many developers, libraries such as OpenZeppelin's SafeMath[2] became popular. These libraries would implement logic around the arithmetic to revert operations in case of an over- or underflow. Since it feels less intuitive to most programmers to call a function called *add()* instead of using the + operator, there is a certain risk that the incorrect operation would be used and the danger of over- and underflows was still present. With the introduction of Solidity 0.8.0, this issue was largely resolved. The language now features built-in checks for overflows and underflows on arithmetic operations. As a result, the need for external libraries like the previously mentioned SafeMath was reduced.

[1] https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic
[2] OpenZeppelin SafeMath: https://docs.openzeppelin.com/contracts/2.x/api/math/

However, the update also introduced the *unchecked* keyword. When a block is marked with this keyword, arithmetic operations behave just like before the update and can over- and underflow. This behavior is desired in some situations as the operations require lower gas fees, but the feature still allows developers to inadvertently introduce bugs.

Continuing this notion, the gas fee system that is employed by EVM smart contract execution incentivizes developers to take risks for the purpose of reducing fees. This trade-off between security and cost efficiency can lead to the introduction of vulnerabilities. While lower gas fees make smart contract interactions more affordable for users, cutting corners in security can have severe consequences, including exploitable vulnerabilities and unintended behavior.

Risks also occur when interacting with external smart contracts. Solidity contracts are allowed to call other contracts. However, when inadequate protections are used, this behavior can lead to reentrancy vulnerabilities. A well-known example for this attack is the "DAO" hack of 2016 [Pra22]. Reentrancy is a vulnerability in Solidity contracts where an attacker is able to repeatedly execute a section of the contract by calling an external contract[3]. The attack works by calling the exploitable method of the vulnerable contract in the attacker's method accepting transactions. This method is called when the contract receives funds from another contract. If state changes in the vulnerable function are only saved after the call to the external contract, this can lead to the exploitable section being called over and over, depositing funds to the attacker contract without ever updating the attacked contract's state.

A minimal example for a reentrancy vulnerability, taken from *Cyfrin.io* [Cyf], can be found in fig. 3.1.

In this example, the *EtherStore* contract's *withdraw* function has the reentrancy vulnerability. The method correctly checks the caller's balance in lines 9 and 10. It is not possible to call the method if the calling contract has not previously deposited any funds because of the *require* statement. In case the expression in the *require* statement evaluates to *false*, the function is reverted.

However, if the attacking contract has a positive balance, the execution flow will reach line 12, where *EtherStore* attempts to send the balance the attacker's contract is called. Since the attacking contract has no function with the name given by *EtherStore*, the *fallback* function is called instead. The *fallback* function again calls the vulnerable *withdraw* and since the *balances* contract variable in *EtherStore* has not been updated, we again reach line 12 and call *Attack*'s *fallback* function. These recursive calls continue until *EtherStore*'s balance has been drained.

Protection against reentrancy is possible by completing all state changes before making the call to an external contract. A popular approach is also to use reentrancy guards, such as the one by OpenZeppelin[4]. In our example from fig. 3.1, moving the state

[3]Reentrancy: https://solidity-by-example.org/hacks/re-entrancy/

[4]OpenZeppelin Reentrancy Guard: https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard

```
1  contract EtherStore {
2      mapping(address => uint256) public balances;
3
4      function deposit() public payable {
5          balances[msg.sender] += msg.value;
6      }
7
8      function withdraw() public {
9          uint256 bal = balances[msg.sender];
10         require(bal > 0);
11
12         (bool sent,) = msg.sender.call{value: bal}("");
13         require(sent, "Failed to send Ether");
14
15         balances[msg.sender] = 0;
16     }
17 }
18
19 contract Attack {
20     EtherStore public etherStore;
21
22     constructor(address _etherStoreAddress) {
23         etherStore = EtherStore(_etherStoreAddress);
24     }
25
26     fallback() external payable {
27         if (address(etherStore).balance >= 1) {
28             etherStore.withdraw();
29         }
30     }
31
32     function attack() external payable {
33         require(msg.value >= AMOUNT);
34         etherStore.deposit{value: AMOUNT}();
35         etherStore.withdraw();
36     }
37 }
```

Figure 3.1: Simple example of a reentrancy attack

change in line 15 to before line 12 would prevent the vulnerability, as would adding the *nonReentrant* modifier from OpenZeppelin's library.

Access control is another issue that causes confusion and can lead to unexpected behavior [LSS20]. The default visibility of functions in Solidity is public, meaning that any external contract can call them. Restricting access to a function to only one person or a group of people, a very common need, requires developers to either manually check the caller's address in the function or utilize a library. Manual access control negatively affects the readability of the code, as all protected functions need to begin with a *require* statement that checks the address. Additionally, the contract needs some way of initially receiving the configuration of the owner's address. Often, initialization functions that can only be called once are used for this. The idea is that after the contract is deployed, the owner will be able to call this function before anyone else, the contract will set the owner address to the correct value and after this, the initialization function will be locked and can not be called again. However, it is by no way guaranteed that the actual owner is able to call this function before anyone else, leading to unreliability and security issues.

OpenZeppelin provides the *Access Control* library, including a *onlyOwner* modifier that only allows the owner of the contract access to the protected functions[5]. It also supports a system with RBAC for more advanced scenarios. Developers have to include modifiers on all functions they want to protect, any functions not equipped with the modifier will still be publicly accessible. This introduces the possibility of forgetting the modifier on a function. Other languages use a "secure by default" approach, where the default configuration is the most secure option and any exceptions to it have to be manually made by the developer.

The mentioned challenges of writing Solidity smart contracts illustrate why bugs and vulnerabilities are common. Developers have to invest significant time and effort to prevent these issues. Inexperienced programmers may assume behavior that the language does not support.

In particular, inconsistency bugs are prone to happen and often easy to overlook. Copying code, which is a common practice, may easily lead to situations where the added sections show inconsistencies with the rest of the code. For instance, the message sender, message data and send/transfer inconsistencies described in section 4.1 can be the results of reused code without the necessary adaptations. The breaking changes between different versions of Solidity can also lead to inconsistencies, as newer code sections may use different language features if the old sections are not updated. Particularly the unchecked block inconsistency and SafeMath inconsistency, also described in section 4.1, may be caused by changes in language version. Oversights on the developer's part may also be the cause of inconsistencies. Access control or reentrancy protections that have to be explicitly added to public functions fall danger to being left out, as evidenced by access control inconsistency and reentrancy guard inconsistency.

[5] https://docs.openzeppelin.com/contracts/2.x/access-control/

If developers are informed about inconsistencies quickly and provided with actionable feedback, they can fix them and prevent them from ever becoming problems. Static code analysis tools can help to do exactly that.

3.2 Important qualities of static analysis tools

An important question when developing a code analyzer is what information is the most helpful for developers and what aspects to focus on. An analysis tool with perfect soundness and correctness may be a significant academical achievement, but if the tool is not used in practice, it will not prevent any bugs.

Christakis et al. [CB16] conducted an empirical study to find the important functional and nonfunctional properties of a widely adopted analyzer. They also identified the pain points of static analysis, that is, the properties that make developers stop using an analyzer.

According to the study, one important quality for a static analyzer is a meaningful set of default rules or, at the least, easy configuration of the used rules. The default activation of all available rules should most likely be avoided. Rules that do not apply to the users have the opposite effect to being useful, they annoy the developer and distract them from real issues. Examples given for this are the enforcement of different naming conventions than the ones being used or spell checks that do not apply.

The study also finds that a low count of false positives is more important than a low number of false negatives. This means that, at least in the eyes of developers using static analyzers, it is more important to avoid wrong warnings than to find every occurrence of an issue. Badly written or unhelpful warning messages are also identified as an important pain point.

In terms of the types of issues that the questioned developers find to be most important, security issues together with general best practices were given as the main interests. The extensibility of an analyzer with custom rules does not seem to be a high priority for a majority of developers. The suppression of warnings is of major importance, for example to deal with false positives. Suppression inside the code files is found to be most popular.

As for the locations where the users would like to see the generated warnings, most developers prefer them to be shown in their code editor and in build output. Showing issues in the browser or during code review, as it would happen when running in CI pipelines, is seen as less helpful.

When asked about the time that developers are willing to wait for program analysis, the results indicate that there are two acceptable kinds of analyzers in this regard. Analyzers for more complex issues are allowed to take more time for running. These would then most likely be used when running from a central repository overnight. However, the standard for quality, especially in regard to low number of false positives and false negatives, is higher for these kinds of analyzers. Fast analyzers running in a code editor on the other hand have a higher acceptable rate of false positives and negatives.

```
1  contract $C {
2    $T private $V = ...;
3
4    function $G(...) public onlyOwner {
5        ...
6        $V = ...;
7        ...
8    }
9
10   function $F(...) public {
11       ...
12       $V = ...;
13       ...
14   }
15 }
```

Figure 3.2: A Semgrep pattern attempting to find access control inconsistencies

Nachtigall et al. [NSB22] evaluated a number of static code analysis tools, partly based on the properties found to be important described previously. The tools are compared on the quality of the warning message, how they deal with false positives and how well they integrate with existing workflows among other properties. The findings suggest that over half of the reviewed tools show poor warning messages, with almost no explanation of the problem or the underlying cause. Additionally, only a small number of tools allow the user to flag issues as false positives. Around half the tools looked at are available for the CLI only and do not support integration into code editors or offer a graphical user interface. Overall, the authors find that many static analysis tools have usability issues, suggesting that their utilization may be held back by these problems.

3.3 High level functionality of the analyzers

In this paper, we apply inconsistency patterns to Solidity smart contracts. The patterns are implemented in two different variants. Semgrep, an extensible framework for static code analysis that uses pattern matching without requiring compilation, is utilized to define and execute the rules. Rules in Semgrep are written in a way based on providing examples of the patterns of interest. The use of wildcards in the code allows rule authors to define which parts of the code snippets should be replaceable in the search. This makes getting started with Semgrep patterns simple and fast. Covering edge cases and non-standard occurrences of a pattern can be tedious though. An example of a Semgrep rule is found in fig. 3.2. The wildcards ($C, $F, $V,...) are replaceable by names when searching for occurrences of the pattern. This specific pattern looks for any instances where two functions write to the same contract variable, but only one of the two functions is protected by the *onlyOwner* modifier.

To counter some of the disadvantages of Semgrep, SIA, the Solidity Inconsistency Analyzer, is implemented. SIA is a static analysis tool translating Solidity contracts to Datalog

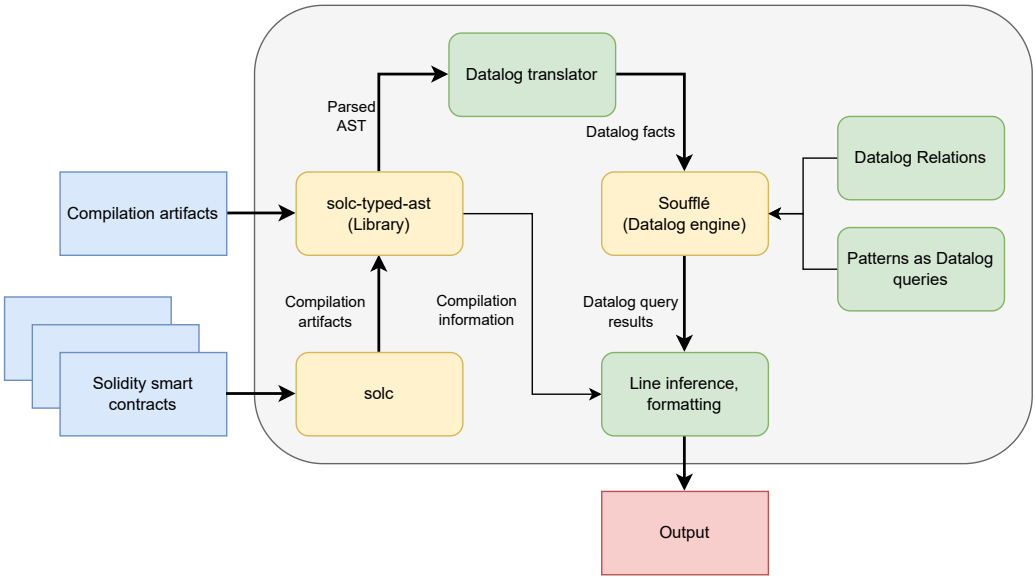

Figure 3.3: High level view on the tool's architecture

facts and applying the inconsistency patterns using Datalog queries. On a high level, the tool is made up of several components. An overview of the architecture is shown in fig. 3.3. Inputs are shaded in blue and outputs in red. The components colored yellow are externally developed tools or libraries where the tool's main job is to feed the correct inputs and outputs in and out of the components. Anything marked green is a proprietary implementation specifically by SIA.

Firstly, as an input, Solidity code or compilation artifacts can be used. In case the tool is called on source code, the Solidity compiler *solc* is used to produce the necessary artifacts containing the program's AST. The AST is extracted from the build data and is then parsed using the *solc-typed-ast* library[6]. The representation received from this library allows for the targeted translation of all necessary information about the program into Datalog facts. These facts, together with the relations describing Solidity's language concepts, are processed using the Soufflé[7] Datalog engine. Using this data and the relations between it allows reasoning about control flow and other important aspects of programs. Our picked inconsistency patterns are then implemented as Datalog queries and can be executed in a performant way.

The query results are then read by SIA, any instances of the defined patterns are recognized and their locations in the code are identified using the information about the source code provided by the *solc-typed-ast* library. Finally, all details about the potential finding are aggregated and output to the user, either on the CLI or to a custom JSON

[6]Consenys solc-typed-ast: https://github.com/Consensys/solc-typed-ast/
[7]Soufflé: https://souffle-lang.github.io/

```
function balanceOfClaimable(address account) returns (uint256) {
    ...
    uint32 today = uint32(blockTimestamp().div(SECONDS_PER_DAY));
    ...
}

function balanceOfUnstakable(address account) returns (uint256) {
    ...
    uint32 today = uint32(blockTimestamp() / SECONDS_PER_DAY);
    ...
}
```

Figure 3.4: An instance of SafeMath inconsistency discovered by Semgrep

```
function transfer(address recipient, uint256 amount) public returns (bool) {
    _transfer(_msgSender(), recipient, amount);
}

function _transfer(address from, address to, uint256 amount) private {
    ...
    _feeAddr1 = 2;
    _feeAddr2 = 10;
    ...
}

function callAirdrop() public onlyOwner {
    _feeAddr1 = 0;
    _feeAddr2 = 0;
    ...
    _feeAddr1 = 2;
    _feeAddr2 = 10;
}
```

Figure 3.5: An instance of access control inconsistency discovered by SIA

format.

Some examples of bugs that the two tools are able to find are explained in the following. Fig. 3.4 shows a Solidity contract containing a SafeMath inconsistency. This type of inconsistency involves the same arithmetic operation occurring in two places, one of them using operators and the other using checked arithmetic through a library. The pattern is explained in more detail in section 4.1.8.

In the example, the functions *balanceOfUnstakable* and *balanceOfClaimable* both calculate the date from a timestamp. However, one of the functions uses the *div* function and the other uses the / operator for the calculation. Semgrep is able to detect this inconsistency thanks to its powerful deep expression matching. This technique can find even deeply nested subexpressions matching a given pattern.

An inconsistency discovered by SIA is illustrated in fig. 3.5. In the program, the contract variables *_feeAddr1* and *_feeAddr2* are modified in the functions *_transfer()* and *callAirdrop()*. The public *transfer* function calls *_transfer()*. The inconsistency present here is an access control inconsistency, the same data is modified through a public and a privileged function. In this instance, the access control problem does not present a vulnerability, however, it still shows some problematic coding practices. SIA was able to discover this issue because of its ability to follow interprocedural control flow, Semgrep does not possess this feature and does not find the inconsistency.

The inconsistency patterns implemented in the two tools are described in section 4.1. Details on the implementation process of the Semgrep patterns and SIA can be found in section 5. An evaluation of the capabilities of both static analyzers is shown in section 6.

CHAPTER 4

Approach

4.1 Inconsistency Patterns

For this work, a number of inconsistency patterns were chosen as the most relevant for Solidity smart contracts. In the following section, these inconsistency patterns are explained and examples for them are shown. The patterns originate from the review of several real-world Solidity smart contracts or are adaptations from preexisting literature. The first three patterns point to security issues, the remaining five can lead to general unexpected behavior.

4.1.1 Contract call inconsistency

An inconsistent contract call means that the address of an external contract is dereferenced and then later checked for the default address of zero. This basic pattern indicates that the programmer had a belief that the address is set to something other than zero when it is dereferenced, however this clashes with the conflicting belief of the check for zero afterwards. In this pattern, the check has to follow the dereference in the control flow, either in the same function or block as the dereference, or through a call to another function that executes the check.

A dereference can be the access of a variable or the call of a function of an external contract, or the attempted transfer of funds to the contract. An exception would be accessing the *code* property of a variable with the type *address*. This property can also be accessed for an address set to zero.

An example for this pattern is shown in fig. 4.1. Here, the *receiver* address is dereferenced in line 2 by sending some funds to it. Later, in line 6, the address is checked for a value of zero. There are clearly two conflicting beliefs about the *receiver* address present. The check should most likely be moved above line 2.

```
1 function send_funds(address payable receiver) external payable {
2     (bool success, ) = payable(receiver).call{value: msg.value}("");
3
4     require(success, "Transfer failed.");
5
6     if (receiver == address(0)) {
7         revert("Can not pay zero address.");
8     }
9 }
```

Figure 4.1: Simple contrived example of a contract call inconsistency

Including calls to other functions in the control flow allows finding instances where the pattern occurs spread over those two functions, however it also introduces some inaccuracies. For example, if the called function is also called from other places, the check might still be necessary even if the variable has always already been dereferenced on certain paths. The variant of this pattern that is extended to function calls is implemented in the static code analysis tool while the Semgrep pattern is limited to occurrences in the same function.

4.1.2 Access control inconsistency

An access control inconsistency is present if two functions write to the same contract variable, but only one of them is protected by the OpenZeppelin *onlyOwner* modifier or a check of the caller's address. This modifier is provided by the Access Control library[1]. As this variable seems to be worth protecting, this indicates that the function that is not protected should also have some form of authentication. The variable can be a scalar value, an array or a map. In Semgrep, the pattern can only be found if the write operation occurs directly in the functions that have or are missing the access control modifier. In the tool, the pattern is extended to occurrences where the two functions call another function and that one makes changes a contract variable.

Just like in the contract call inconsistency, extending the pattern search to function calls may lead to a higher false positive rate due to the more complex interactions between functions, but it can also help to find a higher number of issues.

An illustrating example of the pattern can be found in fig. 4.2. Here, the *admins* mapping describes the administrators of the contract. The *addAdmin* function is correctly protected by the *onlyOwner* modifier. However, the *removeAdmin* function is missing any sort of authentication. This missing protection would be found by the pattern, as both functions write to the *admins* contract variable.

[1] OpenZeppelin AccessControl: https://docs.openzeppelin.com/contracts/2.x/access-control/

```
1 mapping(address => bool) public admins;
2
3 function addAdmin(address newAdmin) public onlyOwner {
4     admins[newAdmin] = true;
5 }
6
7 function removeAdmin(address toRemove) public {
8     admins[toRemove] = false;
9 }
```

Figure 4.2: An example of an access control inconsistency

4.1.3 Reentrancy guard inconsistency

The reentrancy guard inconsistency describes a similar issue to access control inconsistency. While access control inconsistency looks for inconsistencies in authentication between functions manipulating contract data, this pattern pays attention to protections against reentrancy.

Reentrancy is described in a bit more detail in section 3.1. The essence of the issue is a vulnerability in a function calling an external contract that allows an attacker to recursively call the affected function without allowing it to update the contract's state. In some situations, this can lead to the ability to completely drain a contract's balance.

OpenZeppelin's *nonReentrant* modifier[2] protects against reentrancy by using state to remember if a method has already been entered or not. Any attempts to call the method recursively will fail, thereby preventing reentrancy problems. Since reentrancy attacks aim to drain a contract's balance by preventing it from updating its state, any function that might become the target of such an attack will manipulate some contract variable describing the state. If a contract contains other externally callable functions that manipulate the same variables and those functions are not protected by a reentrancy guard, it is possible that those functions are vulnerable to reentrancy attacks. These functions are what the pattern looks for.

Just like the previous patterns, this one is extended in the tool to include function calls. If two functions call the same function and that function manipulates the state, but only one of the calling functions is protected against reentrancy, this may indicate an unprotected reentrancy vulnerability. Again, due to the more complex interactions between the functions, this may however also lead to a higher false positive rate. Semgrep limits the search to the functions executing the write operation.

4.1.4 Message sender inconsistency

The message sender inconsistency refers to a contract accessing the message sender's address using two different ways. This data can either be accessed directly through

[2] OpenZeppelin Reentrancy Guard: https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard

```
1 function invest() external payable {
2     require(isWhitelisted[_msgSender()], "Not whitelisted");
3     funds[msg.sender] += msg.value;
4 }
```

Figure 4.3: Example of a message sender inconsistency

msg.sender or through using OpenZeppelin's *Context*[3]. The *Context* contract provides a *_msgSender()* function that will retrieve the caller of the contract. For most contracts, these implementations work indistinguishably from another and *msg.sender* is probably the preferred method of retrieving the sender.

However, for contracts that will be inherited from or used in a library, using the OpenZeppelin *Context*'s variant will allow more flexibility as the method of retrieving the sender may differ for some use cases. For example, the OpenZeppelin implementation of the ERC2771 protocol for meta transactions uses a custom *Context* that overwrites the default implementation[4].

The documentation for *Context* says the following [Ope25]:

"Provides information about the current execution context, including the sender of the transaction and its data. While these are generally available via msg.sender and msg.data, they should not be accessed in such a direct manner, since when dealing with meta-transactions the account sending and paying for execution may not be the actual sender (as far as an application is concerned).
This contract is only required for intermediate, library-like contracts."

It can be derived that both variants of accessing the message sender are likely fine in most cases. Contracts not functioning as "intermediates" or libraries are not forced to use *_msgSender()*, but can use it if they prefer the variant. Libraries should always use *_msgSender()*. Mixing the variants indicates some confusion about the functionality on the developer's part. In all likelihood, the developer should decide on one variant and use it consistently to avoid bugs and to improve readability.

An example can be seen in fig. 4.3. Line 2 uses the *_msgSender()* function to retrieve the sender, while line 3 uses *msg.sender*. Consistent use of either variant would be fine, but the developer should decide on one and use it in both instances.

4.1.5 Message data inconsistency

This pattern functions similarly to the message sender inconsistency. OpenZeppelin's *Context* also offers the *_msgData()* function which retrieves the call data that the sender has included when calling the contract. The alternative is to call *msg.data*. The main

[3] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Context.sol/
[4] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/metatx/ERC2771Context.sol/

```
1 function donate() external payable {
2     charity.transfer(msg.value);
3 }
4
5 function sponsorCampaign(string calldata message) external payable {
6     bool success = charity.send(msg.value);
7     require(success, "Sponsorship failed");
8 }
```

Figure 4.4: An example of send/transfer inconsistency

reason for using the *_msgData()* variant is to support the substitution of this functionality by contracts inheriting from a library contract.

Again, both variants are fine to use on their own, but mixing them can cause confusion and is an inconsistency.

4.1.6 Send/Transfer inconsistency

Sending ETH from one contract to another is a necessary and very frequently used primitive in Solidity smart contracts. The language supports multiple variants for executing this functionality. The main ways are using the functions *send*, *transfer* and *call*. Before the Istanbul hard fork in January 2019[5], using *transfer* was the recommended way for most use cases. The function contains some protection against reentrancy by limiting the gas cost that the receiving contract is able to use, preventing reeentrant calls to a degree. It also implements some error handling, reverting the transaction in case something goes wrong on the receiver's side, in contrast to *send*. With the changes to the Ethereum block chain in 2019, the gas cost of an operation in the EVM was changed, breaking a number of contracts in the process [Mar19]. After these changes, both *send* and *transfer* are no longer recommended to be used, as this precedent showed that gas costs can change and the fixed gas budget of the *send* and *transfer* functions can not be relied upon. However, many contracts still use the *send* and *transfer* functions.

The send/transfer inconsistency pattern has been adapted from one originating in SmartCheck [TVI+18]. SmartCheck flags any use of *send* and suggests the use of *transfer*. Redefining this pattern as an inconsistency, we focused on the use of *send* and *transfer* for the same variable. Using *transfer* indicates the belief that the call may fail and error handling is required. A call to *send* for the same address suggests that the call is not expected to fail or custom error handling is used. In all likelihood, the same variant should be used.

We can see an example of such an inconsistency in fig. 4.4. The same address, *charity*, is being used with *transfer* in line 2 and *send* in line 6. A choice should be made for one variant and it should be used consistently.

[5]EIP-1679: https://eips.ethereum.org/EIPS/eip-1679/

```
1  pragma solidity >=0.8.0 <0.9.0;
2
3  contract Bank {
4      mapping(address => uint256) public funds;
5      address[] public users;
6
7      function totalFunds() public view returns (uint256 sum) {
8          for (uint256 i = 0; i < users.length; i++) {
9              sum += funds[users[i]];
10         }
11     }
12
13     function partialFunds(address[] calldata users) public view returns (
           uint256 sum) {
14         unchecked {
15             for (uint256 i = 0; i < users.length; i++) {
16                 sum += funds[users[i]];
17             }
18         }
19     }
20 }
```

Figure 4.5: An example of unchecked block inconsistency

4.1.7 Unchecked Block Inconsistency

The *unchecked* block, introduced in Solidity 0.8.0[6], allows the usage of unchecked arithmetic operations. A more detailed explanation is given in section 3.1.

Any arithmetic operations executed in such a block is not checked for under- and overflows. An inconsistency can occur if the same operation is used in an *unchecked* block and outside of one if the same variables are used. This suggests a belief that the operation is safe and can not over- or underflow inside the *unchecked block*. However, the same operation in another place is deemed not safe and checked math is used. The inconsistency shows that either the unchecked instance should also be protected with a check or that gas costs could be reduced by using unchecked math. This pattern only applies to contracts with a Solidity version over 0.8.0.

An illustration of the pattern is shown in fig. 4.5. The contract provides two functions that provide insight into the total available funds. The *totalFunds* function calculates the funds of all users, *partialFunds* allows the caller to include an array of addresses, the sum of funds of that set of users will be calculated. The latter implementation uses an *unchecked* block while *totalFunds* does not use one. Since the arithmetic inside the functions is essentially the same, most likely either both or neither functions should use an unchecked block. In case *sum* cannot overflow, the operations should be executed in an unchecked way, otherwise the operation should always be checked.

[6]https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic

```
1  pragma solidity 0.5.0;
2  import "@openzeppelin/contracts/math/SafeMath.sol";
3
4  contract Bank {
5      using SafeMath for uint256;
6
7      mapping(address => uint256) public funds;
8      address[] public users;
9
10     function totalFunds() public view returns (uint256 sum) {
11         for (uint256 i = 0; i < users.length; i++) {
12             sum = sum.add(funds[users[i]]);
13         }
14     }
15
16     function partialFunds(address[] memory users) public view returns (
           uint256 sum) {
17         for (uint256 i = 0; i < users.length; i++) {
18             sum += funds[users[i]];
19         }
20     }
21 }
```

Figure 4.6: An example of SafeMath inconsistency

4.1.8 SafeMath Inconsistency

As the main way of doing checked arithmetic operations before Solidity 0.8.0 was to use the OpenZeppelin SafeMath[7] library, for contracts using this older version, inconsistencies are also possible. Here, the checked operations use the library, for example the *add* function, while unchecked operations use the built-in + operator. The same concept as for the unchecked block inconsistency applies here as well: If the exact same operation with the same variables is defined with the SafeMath functions once and in the unchecked variant in another place, this indicates two clashing beliefs. In case the operation can not possibly over- or underflow, gas costs can be lowered by using the unchecked operator. However, if the operation is unsafe, the checked function should be used in both implementations.

This pattern is only valid for contracts using Solidity 0.7.6 or lower, as later versions check arithmetic operations for over- and underflows by default.

To better illustrate the similarity between unchecked block inconsistency and SafeMath inconsistency, fig. 4.6 shows the example from the previous pattern adapted to this one. The contract now uses Solidity 0.5.0 and utilizes the SafeMath *add* function to sum up the values in *totalFunds*. The function *partialFunds* uses the unchecked + operator for its sum. Most likely, the SafeMath variant or the unchecked variant should be used in both of the implementations.

[7] OpenZeppelin SafeMath: https://docs.openzeppelin.com/contracts/2.x/api/math#SafeMath

4.2 Analysis

The inconsistency patterns were implemented using two different static analyzers, Semgrep and SIA. Initially, the patterns were defined in Semgrep. The main reason for this choice was that Semgrep allows the relatively quick and easy development of patterns. A general scope for the patterns and some edge cases can be discovered early on, leading to refinement of the rules. If all the patterns could have been fully implemented in Semgrep, the framework alone would have fulfilled the requirements for this work.

However, while the initial development of patterns in Semgrep was relatively quick, issues with the tool were also discovered early on. Simple versions of most patterns were able to be created and refined, but with an increasing number of edge cases and rules, the development complexity started to rise and lead to hard to maintain pattern definitions. Additionally, one of the patterns could not be developed in Semgrep at all due to the incomplete support for the Solidity language. The encountered problems will be detailed in the following chapter.

Due to the mentioned limitations in Semgrep's imperfect Solidity support and the inability to implement all the selected patterns, the decision was made to develop SIA, the Solidity Inconsistency Analyzer. SIA's architecture was based on established Solidity analyzers, such as Securify [TDD+18]. The tool chain consisting of a translation of the program structure to Datalog and then relying on Soufflé, a high-performance Datalog engine, for the analysis proved successful. The inconsistency patterns can be defined in Datalog, providing more control and enabling more complete pattern definitions.

CHAPTER 5

Implementation

The following chapter details the implementation process of the patterns in Semgrep and in SIA, the Solidity Inconsistency Analyzer. The development methods for both tools will be described based on the practical implementation of the chosen inconsistency patterns. Any challenges that were encountered and respective solutions for them will be explained.

5.1 Semgrep

Semgrep is a framework implementing static code analysis using pattern search in source code. It features support for several languages and includes a set of predefined patterns for most of them. Getting started with the addition of new rules is relatively easy and quick, for simple patterns an example occurrence of the pattern in real code is often enough to develop a prototype of the rule.

Semgrep operates with a custom parser to recognize language structures and types in source code, rather than using a preexisting compiler. This approach has two main advantages. It allows Semgrep to run as a high-performance tool, accelerating analysis even for large programs. It also enables the framework to analyze even partially valid programs. This makes it much more useful for programs under development, as it can already run while code is still being written. However, Semgrep cannot take advantage of the complete feature set that, for example, using the official Solidity compiler to retrieve the program's AST would allow.

For more popular and widely used languages, Semgrep's support is generally comprehensive, allowing all language primitives and concepts to be analyzed[1]. For these languages, the tool even supports some notions of data flow in the analysis. For Solidity, only

[1]Semgrep Supported Languages: https://semgrep.dev/docs/supported-languages/

```
1  contract Bank {
2      mapping(address => uint256) public balances;
3
4      function transfer(address payable target) public {
5          uint256 bal = balances[msg.sender];
6          require(bal > 0);
7
8          (bool sent,) = target.call{value: bal}("");
9          require(sent, "Failed to send Ether");
10
11         require(address(target) != address(0), "Cannot send to address");
12
13         balances[msg.sender] = 0;
14     }
15 }
```

Figure 5.1: An instance of the problem we are looking for with Semgrep

experimental support is available. In the development of inconsistency rules for Solidity with Semgrep, a number of problems were encountered, which will be explained in more detail later in this section.

The tool is executing using a CLI. While a popular approach is the integration into CI pipelines, Semgrep also offers extensions for IDEs like Visual Studio Code, allowing users to run the rules live and provide warnings and fixes to code during development. Additionally, a platform-based solution is available, constantly performing analysis on the code repositories in the background and providing an overview of the identified issues on a dashboard.

Semgrep uses a tiered licensing model where the core tool can be used freely, together with individually defined custom rules and the community defined rules. However, the officially developed rule set and support for the program is only provided to paying customers. Licensed users also receive access to the “Pro Engine”, which is promised to offer enhanced data flow analysis and better results for interprocedural analysis. Furthermore, the output of matching code snippets and other relevant data points for identified findings has recently been restricted to the paid version. Free users are limited to receiving the line number and have to manually look up the code with the violation. The framework has proved to be effective for prototyping patterns and for implementing simple patterns quickly. However, some issues arise for patterns that show more depth or are based on edge cases, especially for Solidity as the language is not fully supported yet.

The development of a new pattern with Semgrep usually starts with a sample occurrence in a program. The following section will illustrate the development process and highlight some of the challenges when creating analyzers with Semgrep through a concrete example.

Our example is based on the contract call inconsistency. Fig. 5.1 shows a program displaying this inconsistency. We can see that *target* is dereferenced in line 8 with the transfer of funds using *call*. Later, in line 11, the address of *target* is checked for the zero

address, leading to the contract call inconsistency. The check is executed using a call to *require*, which will revert the execution in case the condition is not fulfilled.

To detect instances of this issue using Semgrep, we remove all code unrelated to the pattern. The critical lines in this context are lines 8 and 11. Since there is also code between our two lines of interest, we use ellipses (...) between the sections of interest. This is a feature supported by Semgrep that will allow any number of lines to occur between our focus points. What is noteworthy is that Semgrep patterns do not have to be fully syntactically correct Solidity programs. In many cases, constructs like function definitions and contract definitions around our patterns can be left out. However, certain definitions may not support this convenience, as will be explained later.

The pattern under development is defined within a YAML file along with a number of other properties that we will ignore for now. These properties include configuration like the supported language, the severity of the pattern and the message shown to the user. However, for the purposes of this section, we will focus exclusively on the pattern language. Our initial pattern, after removing irrelevant lines, looks like this:

```
(bool sent,) = target.call{value: bal}("");
...
require(address(target) != address(0), "Cannot send to address");
```

Applying this pattern in Semgrep will already successfully find the occurrence present in the program seen above. However, to generalize the pattern and detect instances involving alternative variable names for example, all specific elements must now be replaced by placeholders. In Semgrep, these placeholders are called meta variables and they are defined in the pattern as well. Meta variables can be included in the warning message shown, descriptive names for them are preferable. Their names are limited to uppercase characters, underscores and digits. Any meta variables with multiple occurrences in the pattern will have to match in an instance, otherwise the pattern will not recognize it as such. The more generalized pattern can be seen here:

```
$RETURN = $CONTRACT.$FUNCTION{$VALUE: $BALANCE}($ARGUMENT);
...
require(address($CONTRACT) != address(0), $MESSAGE);
```

Upon further consideration of this pattern, some thought has to be given to identify the core meta variables of interest in the message shown to the user. The meta variable *$RETURN*, representing the return value of the function call, and *$ARGUMENT*, the parameter used in the function call, are not relevant. Furthermore, *$ARGUMENT* currently restricts the function calls to calls with a single parameter. Similarly, *$MESSAGE*, replaced by the message given in the call to *require*, is not important to the pattern. These variables can be replaced with ellipses to make clear that they are not relevant. The core version of the pattern based on the initial example now looks like this:

```
$CONTRACT.$FUNCTION{$VALUE: $BALANCE}(...);
...
require(address($CONTRACT) != address(0), ...);
```

For the initial occurrence, this is as far as the pattern can be generalized. However, considering some variations of the pattern that might also need to be identified using Semgrep, the pattern will need to be extended further. This is also partly to be blamed on Semgrep's incompleteness for the Solidity language. For example, the current pattern would fail to detect instances where the operands in the *require* call are reversed, such that *address($CONTRACT)* is on the right-hand side of the expression and *address(0)* is on the left.

To enable the detection of these variations, Semgrep's meta variable pattern matching can be used. By replacing the condition in the call to *require* with the meta variable *$CONDITION* and using *metavariable-pattern* in the pattern definition file, constraints can then be introduced to limit the possible values for it. The following addition demonstrates the use of this feature:

```
- pattern: |
    $CONTRACT.$FUNCTION{$VALUE: $BALANCE}(...);
    ...
    require($CONDITION, ...);
- metavariable-pattern:
    metavariable: $CONDITION
    patterns:
    - pattern-either:
        - pattern: >
            address($CONTRACT) != address(0)
        - pattern: >
            address(0) != address($CONTRACT)
```

Furthermore, another useful addition to the pattern would be to remove the reliance on the calldata in the function call ("*{$V: $B}*"). Not all functions utilize calldata and the objective for this pattern is to find all types of function calls, as a dereference is given whether there is calldata involved or not. Ideally, this aspect would be treated as optional, it does not affect the core logic of the pattern.

However, Semgrep does not have a way to designate this component as optional. Removing the calldata from the pattern, the tool will no longer find instances where calldata is used. With calldata being part of the pattern, Semgrep will not find occurrences without it. Using a meta variable or ellipses for this construct is also not an option, at the time of development neither of these features is supported for calldata. While Semgrep provides more extensive support for edge cases like this in popular languages like Python, Solidity is not designated as a priority for the development team. The current limited feature set as an experimental language results in a worsened user experience.

To make the requirement of optional calldata for our pattern work, an additional top-level layer has to be introduced in our pattern definition using *pattern-either*. This directive allows the definition of pattern variants inside the same rule file. However, this approach requires the maintenance of both variants despite their similarities. The two versions are practically identical except for the calldata in the function call. Omitting the meta variable pattern, the resulting pattern definition looks like this:

```
- pattern-either:
    - pattern: |
        $CONTRACT.$FUNCTION(...);
        ...
        require($CONDITION, ...);
    - pattern: |
        $CONTRACT.$FUNCTION{$VALUE: $BALANCE}(...);
        ...
        require($CONDITION, ...);
- metavariable-pattern:
    ...
```

Essentially, the pattern has to be duplicated inside the *pattern-either* constraint. This solution has a strong negative effect on the pattern's readability. An alternative approach could be to define these variants in separate files and naming them accordingly to improve the clarity. However, this would also introduce significant management effort for the multiple files over which the pattern would span. Using comments to explicitly describe each variant can enhance the maintainability as well and will be used from here on.

Another variant of the pattern is possible because the *require* function supports two call variants: one including a failure message, another omitting it and just taking the condition as an argument. To introduce this variant, two more alternative pattern definitions have to be included in the rule file. Additionally, as this variant can also occur in combination with the differences in calldata in the function call, the rule has a total of four distinct definitions at this point. Evidently, the number of these combinations scales exponentially. For complex patterns, this leads to a significant loss of clarity and maintainability.

In the context of the contract call inconsistency, checks against the zero address with *if* statements should also be introduced. The pattern may reuse the *$CONDITION* defined previously and could simply replace the *require* statement with an *if* statement. A basic sketch of the pattern is outlined here:

```
$CONTRACT.$FUNCTION(...);
...
if (<... $CONDITION ...>) { ... }
```

In this example, the deep match operator (*<...>* is used around the condition. This operator allows the matching of any expression inside the *if* statement that contains *$CONDITION*. Deep matching is also integrated into the previous implementation of the pattern using *require*. This feature is a noteworthy advantage of Semgrep, it simplifies working with complex expressions that might otherwise be overwhelming. Since the *if* variant can also appear together with the dereference using call data, two more variants have to be added in our top level *pattern-either* construct.

When using *if* statements for the address checks, two distinct types of checks are commonly employed. In one variant, the address is checked for zero and the code for the safe case is executed inside the true block, in case the address is not zero. Alternatively, the opposite

condition is used and contract reverts in case the address is zero. Both of the following implementations are valid:

```
if (address(target) != address(0)) {
    target.call(...);
}

OR

if (address(target) == address(0)) {
    revert(...); // or similar
}
```

The current implementation would only detect the first of these two execution flows. To ensure detection of both of these variants, the checks based on *if* must also include comparisons using *==*, not just *!=*. The meta variable pattern for *$CONDITION* is extended to support this.

During testing, it was observed that Semgrep differentiates between different notations for number literals. Specifically, this becomes a problem when both decimal and hexadecimal notations are used. For the contract call inconsistency, this means that the definition using the decimal number 0 will not match with any use of hexadecimal numbers. Since some developers prefer this notation, it becomes necessary to introduce variants of all the relevant patterns utilizing number literals with both notations. This can be achieved by using a nested meta variable *$ZERO* within the meta variable pattern for *$CONDITION*. *$ZERO* will be restricted to the possible values of '0' and '0x0'. Finally, the resulting meta variable patterns with these additions are defined as follows:

```
- metavariable-pattern:
    metavariable: $CONDITION
    patterns:
    - pattern-either:
        - pattern: >
            address($CONTRACT) != address($ZERO)
        - pattern: >
            address($ZERO) != address($CONTRACT)
        - pattern: >
            address($CONTRACT) == address($ZERO)
        - pattern: >
            address($ZERO) == address($CONTRACT)
    - metavariable-pattern:
        metavariable: $ZERO
        patterns:
        - pattern-either:
            - pattern: >
                0
            - pattern: >
                0x0
```

The nesting of meta variables diminishes readability, but is to be preferred over the alternative of doubling the number of variants defined in the *pattern-either*. The final

pattern, defined alongside the meta variable patterns found above, looks like this:

```
patterns:
    - pattern-either:
        #// require, no calldata, message
        - pattern: |
            $CONTRACT.$FUNCTION(...);
            ...
            require(<... $CONDITION ...>, ...);
        #// require, no calldata, no message
        - pattern: |
            $CONTRACT.$FUNCTION(...);
            ...
            require(<... $CONDITION ...>);
        #// require, calldata, message
        - pattern: |
            $CONTRACT.$FUNCTION{$VALUE: $BALANCE}(...);
            ...
            require(<... $CONDITION ...>, ...);
        #// require, calldata, no message
        - pattern: |
            $CONTRACT.$FUNCTION{$VALUE: $BALANCE}(...);
            ...
            require(<... $CONDITION ...>);

        #// if, no calldata
        - pattern: |
            $CONTRACT.$FUNCTION(...);
            ...
            if (<... $CONDITION ...>) { ... }
        #// if, calldata
        - pattern: |
            $CONTRACT.$FUNCTION{$VALUE: $BALANCE}(...);
            ...
            if (<... $CONDITION ...>) { ... }
```

The final version of the pattern is much less readable and far longer compared to the initial implementation. This shows that the complexities of the pattern, combined with some limitations in Semgrep's Solidity support, lead to an increase in implementation complexity and developer frustration. The development process of the contract call inconsistency pattern demonstrates some of the challenges to maintainability that can occur when using the framework. Patterns often require much higher development effort than initially expected.

Of the eight inconsistency patterns that were selected for this work, 7 were successfully realized in Semgrep. At the time of implementation, the unchecked block inconsistency could not be fully implemented in Semgrep, as the tool did not support the language construct of the unchecked block in rule definitions with more than one function.

Furthermore, during pattern development, some usability issues with Semgrep were noted. Patterns that describe the interaction between two functions have to contain a complete structure of Solidity contracts, meaning the pattern has to define a contract declaration

and can only describe the definition of the two functions inside it. As a consequence, the whole contract is flagged as the location of any matches, complicating the identification of the affected sections. Additionally, this approach to defining patterns leads to duplicate variations, as Semgrep differentiates between the order of functions when they are defined in a contract declaration in the pattern. For most patterns, the order of functions is not significant, all definitions then have to include variants with different function ordering.

Overall, Semgrep proved valuable for refining the selected patterns and identifying for edge cases. The development of the patterns highlighted some problems with the tool and its incomplete support for Solidity. Nevertheless, 7 out of 8 inconsistency patterns were successfully implemented and can be detected using Semgrep.

5.2 SIA - The Solidity Inconsistency Analyzer

The Solidity Inconsistency Analyzer is our implementation of a tool performing static code analysis on Solidity smart contracts, with a specific focus on identifying inconsistency patterns. It is executed via a command-line interface and implemented in TypeScript, using the Node.js runtime environment. Deployment is supported on both Linux and Windows operating systems. Windows installations require the Windows Subsystem for Linux. Furthermore, the Soufflé Datalog engine is required and must be installed within the WSL environment for Windows or directly on a Linux system. Node.js is also a prerequisite for tool execution.

The tool provides a range of configurable parameters. Firstly, a user-specified path determines the input source, which can be either a single file or a directory. When a file path is provided, the tool performs analysis on that specific file. Alternatively, if a directory path is specified, the tool recursively scans the directory for Solidity source files or compilation artifacts. When compiling from source code, a configuration option allows for the selection of either individual file compilation or collective compilation and analysis of all files. This functionality is particularly beneficial when contracts are mutually independent, as it can speed up the process. Furthermore, dependency remappings for external libraries can be configured. Finally, the user can choose between outputting the analysis results to a designated file or displaying them directly in the command-line interface.

The tool accepts both Solidity source code and preexisting compilation artifacts as input. Compilation artifacts generated by build tools such as Hardhat or Foundry are supported. When compilation artifacts are provided, the AST is extracted from the embedded JSON structure and passed to the parsing library, which will be detailed in the following sections. Alternatively, if source code is provided as input, the *solc-typed-ast* library is utilized, which provides a JavaScript interface to the *solc* compiler. The library is maintained by Consensys[2] and enables the tool to invoke multiple compiler versions and, upon successful compilation by any version, retrieve the corresponding AST.

[2]Consensys solc-typed-ast: https://github.com/Consensys/solc-typed-ast/

The parsing of the Abstract Syntax Tree is also facilitated by the *solc-typed-ast* library. Upon receiving the AST as input, this library generates a parsed representation in the form of a type-safe object tree. Each node within this tree corresponds to a specific element of the Solidity code, ensuring accurate typing. This type safe and structured representation accelerates the development process and simplifies the traversal and extraction of targeted language constructs.

In our implementation, the Abstract Syntax Tree is traversed on a contract-by-contract basis. First, for each contract, the contract variables and other properties about the contract, like its name, are translated. After that, each function within the contract is traversed in a top-down manner. Information about the function, such as its visibility and applied modifiers, is recorded. During the traversal of the function's components, control flow and part-of relationships between all nodes are captured. The language primitives translated include blocks, assignments, variable declarations and function calls. To accurately represent these primitives, the expressions they include must also be translated.

For instance, an assignment statement comprises an expression for the assigned variable and an expression for the assigned value. Expressions are translated recursively, as most expression types can contain sub-expressions. Finally, the translated data is serialized into a format compatible with the Datalog language. Here is an example of an assignment statement in Solidity:

```
uint256 sum = summand1 + summand2;
```

This assignment would then be translated to the following Datalog facts:

assignment.facts

```
$Variable("sum")
$BinaryExpression($Variable("summand1"),"+",$Variable("summand2"))
=
[123, "sum_up", "SampleContract"]
```

In our Datalog representation, an assignment has four components. The first is the expression that is being assigned to. Since in our case we're simply assigning to the *sum* variable, this expression is quite simple. The second component is the expression that represents the assigned value. Here, we are adding two variables together, this is translated to a binary expression. The binary expression has a left and a right subexpression and an operator in the middle. In our case, the left variable is *summand1*, the right variable is *summand2* and the operator is $+$. The assignment operator, here $=$, is also translated, as an assignment can also happen with other operators, like $+=$. Finally, we include an identification of the statement containing this assignment. This is necessary so we can later identify where in the code our statement can be found. This statement identification record is made up of a node id, the name of the function containing the statement and the name of the contract the function is implemented in. Together, this data can uniquely identify any node of the AST.

In our example, since the assignment also contains the declaration of the variable, we would also translate a fact about the declaration of the variable *sum*:

`variable_declaration.facts`

```
"sum"
[122, "sum_up", "SampleContract"]
"function"
"internal"
"mutable"
"default"
```

The declaration of a variable is translated using six components. The first is the name of the variable, in our case *sum*. Just like for assignments, we include an identification record of the statement containing this declaration. Additionally, for declarations we're interested in some other details. The remaining four properties represent the type of the scope of the variable, its visibility, mutability and storage location. Since this is a local function variable, the scope is limited to the function the variable is declared in, this also makes it only internally visible. Since the declaration is not for a constant value, the variable is mutable and its storage location is default, meaning the value is simply stored in memory.

The Soufflé Datalog engine requires input in the form of fact files, as they have been shown in the examples above. These files are simple TSV files by default. To represent the data types derived from the AST, a corresponding representation within Datalog is necessary, achieved through a combination of Datalog type declarations and rule definitions. Fact files contain factual data, forming the basis for rule evaluation. Examples include representing the existence of a contract with a specific name or function invocation within specific statements. Types describe how the data that is operated on looks, while relations define the relationships between data. Relations can consist of multiple rules that contain the logic of the relationship. Soufflé supports record types, which are useful for constructing complex types from more basic ones. In the examples seen above, the statement identification is implemented as a record type, it contains a node id, a function name and a contract name.

For expressions, we use algebraic data types. This variant of types allows branches and recursive type declarations, which are particularly well-suited for representing expressions. The expressions in the illustrative example above show this well. Algebraic data types can have multiple branch constructors, such as the *$Variable* and *$BinaryExpression* shown before.

The defined Datalog rules expand the input data and can inference implicit relationships. For instance, the input of direct control flow relationships between nodes, such as a node directly succeeding another, is expanded by the application of transitive property rules. This process leads to a more complete view of all the potential control flow paths within the program. What is more, function calls are included in this control flow inference, this allows for the findings to include inter-procedural control flow. For a number of patterns, this information can lead to a higher quality of analysis.

In addition to control flow relationships, composite-of relationships are essential primitives. For instance, to accurately represent the branching structure of an if statement, it is necessary to translate which statements are contained within each block. Similarly, for unchecked blocks, the contained statements must be identified. These composite-of relationships are integrated with the control flow relationships previously described. For example, given a variable declaration followed by an if statement, it is important to maintain the information that all statements within both branches follow the variable declaration. We can make this connection through the fact that the if statement follows the declaration and the statements are contained in the blocks of the if statement. This requires the combined application of these two primitive relationships.

The Soufflé Datalog engine processes the fact files and includes our custom language representation and the defined patterns. It generates the results of the selected rules as output. Each of the patterns has a single, designated output fact. These outputs, formatted as CSV files, are parsed when Soufflé's processing completes. Typically, the output includes statement identifiers that point to the locations of the detected inconsistencies. As these identifiers contain node IDs rather than direct source code references, the nodes within the AST must be located first. This is achieved by reusing the representation provided by the *solc-typed-ast* library. All nodes matching the identified IDs are retrieved, and their associated source code locations are extracted. The library provides data about the node's position within the source code, enabling our tool to open the correct file and extract the relevant sections. Line number inference is also performed at this stage, as the library does not provide this information. Following this, all necessary information regarding each pattern instance is put together to construct the final output. For CLI output, a formatted message containing the required information is generated. For JSON file output, the warnings are grouped by their respective identifier and a JSON structure is created that includes line numbers, file paths, and the same warning messages consistent with the CLI output. Currently, standardized output formats, which would allow integration with IDEs or other warning display interfaces, are not supported.

The implementation of each inconsistency pattern is encapsulated in its own Datalog file. Core language constructs and functionalities shared across multiple patterns are put into separate, reusable Datalog files. This modular design helps the construction of complex patterns from simpler conceptual building blocks. An advantage of this hierarchical structure is a higher level of expressiveness compared to other methods of pattern definition. We can build from simpler, reusable concepts and break down the complexity this way.

The following sections describe the broad implementation of each of the used patterns. The exact implementations can be found in the code repository.

5.2.1 Contract call inconsistency

This inconsistency pattern is built up by two supporting subcomponents.

Firstly, it requires the presence of a variable dereference. Dereferencing can manifest in several forms: accessing a member of a non-primitive variable, invoking a function on the variable, or, in the case of an address type, transferring funds using *send*, *transfer*, or *call*. We identify all statements that perform such variable dereferencing operations with the *dereferences_variable* fact.

Secondly, the pattern requires a check on the variable's address. This check can be implemented through multiple mechanisms. If the variable is not of the *address* type but represents an external contract, its address must be obtained with a call to *address(<variable>)*. It is important to account for both direct variable checks and checks involving the address function.

The address check itself can then be performed using *require*, which reverts the execution context if its containing condition does not hold. It may also optionally include an error message in case the condition fails. A check using *require* may look like the following:

```
require(address(external_contract) != address(0),
    "Expected external_contract to have an address.");
```

Alternatively, the check can be implemented using an if statement, as shown here:

```
if (address(external_contract) == address(0)) revert();
```

It is also important to note that the order of operands in these address comparison expressions may of course vary.

With both subcomponents defined, the final condition for identifying this pattern is the control flow relationship. Specifically, the address check must follow the variable dereference in the control flow. If all the mentioned conditions are satisfied for a set of statements, a contract call inconsistency is identified. An illustrating example of this pattern is found in fig. 4.1.

5.2.2 Access control inconsistency

This pattern identifies instances where two functions within the same contract modify a shared contract variable, but show different access control mechanisms. Specifically, one function must have the *onlyOwner* modifier, restricting access to the contract owner, while the other function is missing any explicit access control, such as a modifier or an equivalent check.

To detect any modifications to contract variables, the abstraction *writes_to_contract_variable* is utilized, a construct that will also be used in other patterns. This abstraction is defined as follows: Firstly, any statement that assigns a value to a contract variable, including assignments to indexable data structures like arrays, is considered a write. Secondly, a recursive definition is used to identify functions that invoke other functions which, in turn, write to contract variables. This allows the detection of inter-procedural writes, where the called functions act as setters.

The access control inconsistency pattern searches for pairs of functions within the same contract where one function has the *onlyOwner* modifier, the other does not, and both functions modify the same contract variable. Exceptions are made for constructors, private or internal functions, and functions that perform manual caller address checks. Such checks are typically implemented using require statements that compare *_msgSender()* or *msg.sender* values against a specific address. If a statement identified by *writes_to_contract_variable* is preceded by such a check, the pattern is not triggered.

However, if none of the exceptions apply, we have found an access control inconsistency. We can see an example of this pattern in fig. 4.2.

5.2.3 Reentrancy guard inconsistency

Similarly to access control inconsistency, reentrancy guard inconsistency aims to find two functions that write to the same contract variable, where only one of these functions uses a reentrancy guard. In principle, these two patterns work exactly the same way, they just look for different modifiers.

The *writes_to_contract_variable* abstraction is reused to identify pairs of functions that write to the same contract variable. Refining the pattern, one of the functions needs to have a reentrancy guard while the other one does not. The exceptions for this rule again include cases where the function with the potential reentrancy issue is private, internal or a constructor. Additionally, any functions protected with an *onlyOwner* modifier are exempt from the pattern. The latter case is an exception because we can assume that the owner of a contract would not use a reentrancy vulnerability.

An illustrative example of this pattern is shown in fig. 5.2. In this example, both the *withdraw* and *transferToSomeoneElse* functions allow the caller to withdraw funds that they deposited earlier. However, the difference between the two functions is that *withdraw* will always send the money to the caller, while *transferToSomeoneElse* accepts a target address as a parameter. Both of the functions modify the *balances* mapping, but only *withdraw* is protected with the *nonReentrant* modifier.

The relevant facts translated to Datalog will look like this:

```
assignment.facts
...
$IndexAccess($Variable(balances), $MemberAccess($Variable(msg), sender))
$Number(0)
=
[58, withdraw, Bank]
...
$IndexAccess($Variable(balances), $MemberAccess($Variable(msg), sender))
$Number(0)
=
[98, transferToSomeoneElse, Bank]
...
```

```
1  contract Bank is ReentrancyGuard {
2
3      mapping(address => uint256) public balances;
4
5      function deposit() public payable {
6          balances[msg.sender] += msg.value;
7      }
8
9      function withdraw() public nonReentrant {
10         uint256 bal = balances[msg.sender];
11         require(bal > 0);
12
13         (bool sent,) = msg.sender.call{value: bal}("");
14         require(sent, "Failed to send Ether");
15
16         balances[msg.sender] = 0;
17     }
18
19     function transferToSomeoneElse(address payable target) public {
20         uint256 bal = balances[msg.sender];
21         require(bal > 0);
22
23         (bool sent,) = target.call{value: bal}("");
24         require(sent, "Failed to send Ether");
25
26         balances[msg.sender] = 0;
27     }
28 }
```

Figure 5.2: Example of a reentrancy guard inconsistency

Both statements that assign to *balances* are shown here. One of them has the id 58 and is inside *withdraw*, the other with id 98 is in *transferToSomeoneElse*.

```
function_modifier.facts

[withdraw, Bank]  nonReentrant
```

The fact file containing function modifiers then shows us that only *withdraw* has the *nonReentrant* modifier. The pattern will combine this knowledge, together with other facts like the function visibility that is not shown here, and find the reentrancy guard inconsistency between the two functions.

5.2.4 Message data inconsistency

This pattern identifies instances within a single contract where two access the message data inconsistently. One of the statement calls *msg.data* and the other invokes *_msgData()*. This pattern's implementation is relatively straightforward, the only constraint is that the two statements need to be in the same contract. Due to its brevity and simplicity,

```
1  .decl calls_msgData(contract: contract_t, statement: statement_t)
2  calls_msgData(contract, statement) :-
3      member_access($MemberAccess($Variable("msg"), "data"), statement, _),
4      statement = [_id,_function_name,contract].
5
6  .decl calls_msgDataFunction(contract: contract_t, statement: statement_t)
7  calls_msgDataFunction(contract, statement) :-
8      function_call(["_msgData", "Context"], "nil", statement),
9      statement = [_id,_function_name,contract].
10
11 .decl msg_data_inconsistency(contract: contract_t,
12                                msgDataFunctionStatement: statement_t,
13                                msgDataStatement: statement_t)
14 msg_data_inconsistency(contract,
15                         msgDataFunctionStatement,
16                         msgDataStatement) :-
17     contract(contract),
18     calls_msgDataFunction(contract, msgDataFunctionStatement),
19     calls_msgData(contract, msgDataStatement).
20
21 .output msg_data_inconsistency
```

Figure 5.3: Datalog implementation of the message data inconsistency

we have an opportunity to show the pattern in full in fig. 5.3 and can use it to illustrate the approach used to define the patterns.

The Datalog implementation uses three relations: *calls_msgData*, *calls_msgDataFunction* and finally *msg_data_inconsistency*. Since the latter is the output relation, line 21 instructs the Datalog engine to output all occurrences of that relation. The relation *calls_msgData* is defined to look for any statements that call *msg.data*. For this purpose, we're using the *member_access* relation that contains all member access statements. We define that the statement *statement* will have to be a member access with the expression *$MemberAccess($Variable("msg"), "data")*, which is our Datalog equivalent of the respective Solidity code.

In the *calls_msgDataFunction* relation, we limit *statement* to be any statement in the *function_call* relation that calls the *_msgData* function from the *Context* contract. This is the *OpenZeppelin* contract that contains the function.

Finally, the *msg_data_inconsistency* relation defines a rule that, for any contract that exists (line 17: "*contract(contract)*"), any pair of statements where one is in *calls_msgDataFunction* and one is in *calls_msgData*, if those two statements are in that contract, we have an occurrence of the inconsistency.

5.2.5 Message sender inconsistency

This pattern mirrors the message data inconsistency very closely, but instead targets inconsistencies in calls to *_msgSender()* and *msg.sender*. The implementation is exactly

the same, with only the function and property names exchanged. An example of this pattern is found in fig. 4.3.

5.2.6 SafeMath inconsistency

This pattern aims to find any instances where identical arithmetic expressions are used with different implementations. Specifically, arithmetic operations where one uses the language's operator while the other uses the corresponding SafeMath function are identified, limited to statements defined in the same contract. This initial definition is broad and could yield a large number of false positives, a further constraint is used to require an exact match of the expression, including variable names. The variable names within the assignment statement using the operator must match with the ones used in the equivalent call to the SafeMath function. Additionally, any assignments that utilize a combined assignment and arithmetic operator are also deemed equivalent, for example $+=$, the $+$ operator and the *add* function. Some illustrative matching operations are shown here:

```
// SafeMath
x = x.add(y);

// unchecked math
x += y;
x = x + y;
```

In the preceding statements, the pattern aims to match the SafeMath function call with both the use of the $+=$ and of the $+$ operators. If such a pair is identified within the same contract, the inconsistency is flagged.

An exception is implemented such that the SafeMath contract implementation itself will be excluded, as the contract inherently relies on these operations. The contract is included explicitly in many Solidity files, flagging it would not prove helpful. An example of this pattern is illustrated in fig. 4.6.

5.2.7 Send/Transfer inconsistency

This pattern identifies instances where *send* and *transfer* are invoked on the same address in a contract. The basis for the verification of this pattern lies in the *function_call* relation, which includes any variable that a function is called on. To illustrate, the following Solidity code shows the simplest version of this inconsistency:

```
target.send(1);
target.transfer(1);
```

The translation to Datalog would yield the following entries in the *function_call* relation:

```
function_call.facts

["send", ""] target [30, functionName, contractName]
["transfer", ""] target [35, functionName, contractName]
```

Using this data, any two matching statements can be identified and in case they are implemented in the same contract, the statements are flagged. An example of this inconsistency can be found in fig. 4.4.

5.2.8 Unchecked block inconsistency

Conceptually, this pattern is implemented the same way as the SafeMath inconsistency. The two patterns are exclusively possible in different major versions of Solidity. Unchecked block inconsistency targets instances where an identical operation, performed on the same variables, is executed both within and outside an unchecked block. Determining whether a statement is defined inside of an unchecked block is relatively simple. A relation including all node ids of unchecked blocks is translated to Datalog. Additionally, hierarchical relationships between nodes are translated. Through these relationships, the definition of the *in_unchecked_block* relation is possible, including all nodes that are defined inside of an unchecked block. The next step is ensuring whether two assignments use the same variables and operators. While this is conceptually simple, the same challenges are encountered here as in the SafeMath inconsistency pattern. For the simplest case, if the left side and right side expressions of an assignment statement are the same, the two statements match.

However, combined assignment and arithmetic operators like $+=$ complicate the implementation. This requires the deconstruction and reconstruction of expressions using equivalent assignment operators. In case two statements are semantically equivalent and they are defined in the same contract, but exhibit a difference in whether they are contained in an unchecked block, an instance of this pattern is identified. An example of this pattern is shown in fig. 4.5.

CHAPTER 6

Evaluation

To evaluate the effectiveness of the two tools, SIA and Semgrep were both run on a set of Solidity smart contracts. The two tools were both run on a small set of 100 randomly chosen contracts. The setup for this experiment is described in section 6.1. Additionally, the analyzers were run on a larger data set of around 1600 contracts. The findings that were obtained from the experiment are analyzed in detail in section 6.2. A discussion of the findings is presented in section 6.3.

6.1 Experiment

As a data set for the experiment, the Ethereum Smart Contract Sanctuary [OE23] was used. This repository is part of an initiative aimed at collecting all contracts deployed on the Ethereum network. Consequently, these contracts are real world, deployed contracts and no incomplete or under development programs are part of the data set. This selection offers higher quality code compared to, for example, scraping random Solidity contracts from GitHub. However, it should be acknowledged that in comparison to contracts still in development, a different spectrum of inconsistencies may be discovered and potentially prevented.

A significant advantage of this data set, in comparison to directly pulling contract byte code off the block chain, is the availability of Solidity source code, which is the necessary input for both tools. What is more, most of the source code files in the repository include the dependencies and any libraries necessary to compile the contracts. While this is not important for Semgrep, as the tool does not need complete valid contracts as input, for SIA, this is necessary and prevents the need for dependency management and library version inference.

For the experiment, a random sample of 100 contracts is selected from the smart contract sanctuary. A Python script is used to randomly select the contracts from the repository

and copy the files to a designated directory. Both analyzers then run on the same contract files.

To ease the deployment of Semgrep and ensure that the latest version is used, the Docker image provided by the developers is used to create a container. The contract files are then mounted into the container. The custom Semgrep inconsistency patterns are pulled into the container by cloning the Git repository containing them. Semgrep is then configured to only use the custom inconsistency patterns and ignore all predefined patterns. Furthermore, the tool is configured to output its findings to a JSON file to ease the evaluation and comparison of results.

On the data set of 100 contracts, the execution of Semgrep completes in around 7 seconds. The tool reports an analysis time of 2 seconds, however, the total elapsed time from starting the program through the CLI to receiving the results is around 5 seconds longer than the reported value. All contracts are successfully analyzed, with Semgrep reporting no errors during execution. The resulting JSON file can be copied from the Docker container for evaluation of the results.

SIA is executed directly on the folder containing the randomly selected contracts. As SIA relies on the Solidity compiler *solc* for retrieving the AST, all contracts are required to be fully valid Solidity programs. Many contracts in the smart contract sanctuary require dependencies like the popular OpenZeppelin libraries to run. For most contracts, the sanctuary repository includes the library contracts in the correct version in the same file, which eases the compilation process. However, import statements from the original developer are frequently still present, many of them directly importing a file, as dependencies are often explicitly kept next to the code files in their own repository. Since the library contracts are found in the same file, these import statements are invalid and, without changes, would prevent the contracts from compiling. Additionally, the contracts and their dependencies are often found in the wrong order in the Solidity file. In Solidity, base contracts must occur above their dependent contracts in a code file, a constraint that is often violated in the repository.

To ensure that SIA is able to analyze as many contracts as possible, a transformation layer is introduced that can be activated with a configuration flag. When the flag is activated, SIA will first try to compile the input contract without changes using *solc*. In case the compilation succeeds, the tool runs as normally with the received AST. However, if the compilation fails, a series of transformations are executed on the contract file before attempting another compilation.

The transformations are implemented using regular expressions. Since many of the transformations are relatively simple or only involve deleting parts of the file content, regular expressions are powerful enough to execute them. This approach prevents the need to use a parser for Solidity, which could introduce its own set of complexities.

Firstly, all import statements are removed, since all the necessary dependencies can likely already be found in the same file. Secondly, all comments describing a contract's license have to be removed, as a file is only allowed to have a single license set or none

at all. Running with multiple licenses in place also leads to compilation failures. The contract files in the sanctuary contain all the library contracts with their respective licenses. Thirdly, a reordering of the contracts in the file is necessary. The contracts must be ordered such that base contracts always precede inheriting contracts. Since the dependency graph is a directed acyclic graph, a simple topological sort algorithm can be used for this. In essence, a depth-first traversal of the dependency graph is used to make sure dependencies are always in place before their dependents.

Finally, the dissected contracts are concatenated and the transformed Solidity source code is written to a temporary file. The temporary file is then given as input to the solidity compiler.

In practice, this transformation significantly improves the compilation rate of the repository's contracts. Before the transformation, the tool was run on 1000 contracts of the repository and was able to complete the analysis of 57% of them. After the addition of the transformation, 89% of the contracts compiled successfully and could be analyzed. The remaining portion of contracts that can not be compiled is mostly due to cases where the smart contract sanctuary does not contain dependencies of contracts. Since the project has to match deployed contracts with the source code found in public repositories, it is not always able to find all dependencies or components of contracts. These contracts will then be incomplete in the repository and can not be used by SIA.

Running SIA on the experiment data set takes 83 seconds in total. Of that time, 54% of it is spent in *solc* compiling the contracts, 38% is used for running the Datalog engine and the remaining 8% of time is spent inside SIA's transformation logic, translating the *AST* to Datalog, redirecting the data and formatting the results. Out of the 100 contracts, 92 of them can be compiled and analyzed by SIA. 18 out of those 92 could only be compiled thanks to the transformations described previously. SIA is configured to output the findings to a JSON file as well, similarly to Semgrep.

With both tools successfully completing their runs on the experiment contracts, their respective output files can now be compared. The Semgrep output file contains the findings unordered, each of them is tagged with a check id, the name of the file and some information on the span inside the code that is affected, in addition to some metadata and the user facing message. SIA's output file is grouped by the pattern's names, the findings contain the pattern's id, the analyzed file's name, information regarding the location of the findings and the message that would be shown to the user.

The two tools' findings are now unified to a common data format. Since both contain similar data, the translation is relatively simple. File names differ for the two tools, since Semgrep uses the path inside the Docker container while SIA outputs the path in the machine's file system. To circumvent this, only the file name itself and not the path is used, since there are no duplicate file names in the data set. SIA and Semgrep also use different diagnostic ids. Semgrep often has multiple distinct ids for the same inconsistency, as the patterns are sometimes split over multiple definition files. The diagnostic ids are simply translated with a look up table to unify the patterns from both

tools.

The main challenge in matching the findings is the affected span in the code. Semgrep, as described previously, often marks a whole contract as the location for a pattern. SIA pinpoints the locations more precisely, but in the contracts that SIA could only analyze because of the previously mentioned transformations, the spans will not match with the spans of Semgrep at all. These differences are evaluated manually, to make sure that the tool's output fits together consistently.

For running the two tools on the larger data set of 1605 contracts, the exact same setup is used. Semgrep takes 13 minutes and 13 seconds to run on this data set. Since the data set contains some large contracts, a configuration flag needs to be set to prevent timeouts. On 18 contracts, Semgrep fails to complete the analysis or is only able to partially analyze the files. This is due to internal errors or syntax that Semgrep does not understand. Errors like these are relatively rare, Semgrep can usually also run on incomplete contracts. The errors are also contained in Semgrep's output file.

SIA takes 31 minutes and 18 seconds to complete the analysis of the 1605 contract files. Compilation takes up 37% of the runtime for this data set, 57% is used for running Soufflé and the remaining 6% are used for SIA's translation steps. The compilation fails for 187 contracts and succeeds for 1414. 501 out of those contracts could only be compiled because of the aforementioned transformations.

6.2 Findings

In the following section, the findings discovered in the experiments will be analyzed. Section 6.2.1 will focus on the smaller data set with 100 contracts. All the flagged inconsistencies from this experiment will be analyzed in detail. In section 6.2.2, the results of the experiment with the big data set of around 1600 contracts are explained. Since there are a lot more instances of inconsistencies in that experiment, not all of them can be manually verified.

6.2.1 Detailed data set

In the experiment with 100 randomly selected contracts from the smart contract sanctuary, Semgrep found 4 instances of inconsistencies in total. SIA found a total of 14 inconsistencies. An overview of the detected inconsistencies can be found in table 6.1.

All of the inconsistencies found by Sempgrep are message sender inconsistencies, these discovered issues were also all found by SIA. Each of the detected instances were manually verified to be true positives, meaning all the contracts use both *_msgSender()* and *msg.sender*. An exact reason for why these contracts use the two variants interchangeably can not be determined, though an assumption can made that copying code may have led to these inconsistencies. An example of one of these inconsistencies can be seen in fig. 6.1.

	SIA	Semgrep
Contract call inconsistency	0	0
Access control inconsistency	7	0
Reentrancy guard inconsistency	2	0
Message data inconsistency	0	0
Message sender inconsistency	5	4
SafeMath inconsistency	0	0
Send/Transfer inconsistency	0	0
Unchecked block inconsistency	0	0
Total	14	4

Table 6.1: Comparison of the number of contracts with detected inconsistencies in SIA and Semgrep in the detailed data set

In the example, the *withdraw* function uses *msg.sender* to retrieve the caller's address while *mint* and a number of other functions use *_msgSender()*. Since the contract is not intended to be used as a library, in practice, the inconsistency is unlikely to lead to bugs or unintended behavior. However, it can still be considered a bad practice and consistent coding style can lead to higher readability and maintainability of the program.

SIA found more instances of inconsistencies than Semgrep, the tool also discovered one message sender inconsistency that went undetected by Semgrep. The program containing this inconsistency can be seen in fig. 6.2.

In the affected program, the inconsistency occurs between the function *receive* and the constructor of the contract. The reason Semgrep did not find this inconsistency is due to the use of *msg.sender* in the *receive* function. Semgrep can detect and match the code in the constructor, however the *receive* function is a feature of Solidity released with version 0.6. Its purpose is to be used as the receiving function when the contract is sent funds. It can be assumed that Semgrep does not have support for this special function yet, multiple attempts to extend the pattern to also work with these functions were unsuccessful.

SIA also reports a total of two reentrancy guard inconsistencies and 7 access control inconsistencies, Semgrep does not report any of these. The two reentrancy guard inconsistencies both stem from two functions calling the same subprocedure, where only one of them is protected by a reentrancy guard. In these cases, the called subprocedure makes changes to a contract variable. However, in both of these findings, the unprotected function does not execute any calls to external contracts before the contract variable is updated, meaning that the function is not vulnerable to reentrancy attacks and these are false positives. Since the Semgrep pattern does not follow the calls to subprocedures, it did not find these instances.

In two of the access control inconsistencies flagged by SIA, the contract variable that is written to has the mapping type. In both of these instances, the write operations

```
1  contract HighCouncilOfKingz is ERC721, Ownable {
2
3      function mint(uint256 _mintPassTokenId, uint256[] memory _burnTokenIds)
4          external
5          callerIsUser
6          onlyAllowMintEnabledAndValidCount(1)
7      {
8          ...
9          _totalMintSupply++;
10         _safeMint(_msgSender(), _getTokenToBeMinted(_totalMintSupply - 1));
11         // ensure owner of mint pass
12         require(
13           _msgSender() ==
14             IMintPassContract(mintPassContract).ownerOf(_mintPassTokenId),
15           'Not owner'
16         );
17         ...
18     }
19
20     ...
21
22     function withdraw() external payable onlyOwner {
23         (bool success, ) = payable(msg.sender).call{ value: address(this).
               balance }(
24           ''
25         );
26         require(success);
27     }
28 }
```

Figure 6.1: One of the detected message sender inconsistencies

```
1  contract Fungle is ERC721, Ownable {
2
3      constructor() ERC721("Pick-A-Twin", "FNGLPT") {
4          _safeMint(0x28F6A2BEe9Ce20fc0aB14dB3A7dB444f6f5988B7, 1);
5          _safeMint(_msgSender(), 2);
6      }
7
8      ...
9
10     receive() external payable {
11
12         payrollArtist1.transfer(msg.value/2);
13         payrollArtist2.transfer(msg.value/2);
14
15         emit PaymentReceived(msg.sender, msg.value);
16     }
17 }
```

Figure 6.2: The message sender inconsistency only found by SIA

```
function _transfer(address from, address to, uint256 amount) private {
    require(from != address(0), "ERC20: transfer from the zero address");
    require(to != address(0), "ERC20: transfer to the zero address");
    require(amount > 0, "Transfer amount must be greater than zero");
    _feeAddr1 = 2;
    _feeAddr2 = 10;
    ...
}

function callAirdrop() public onlyOwner {
    _feeAddr1 = 0;
    _feeAddr2 = 0;
    for(uint i = 0; i < airdropKeys.length; i++){
        _tokenTransfer(msg.sender, airdropKeys[i], airdrop[airdropKeys[i]]);
        _isExcludedFromFee[airdropKeys[i]] = false;
    }
    _feeAddr1 = 2;
    _feeAddr2 = 10;
}
```

Figure 6.3: The access control inconsistency with unusual use of contract variable

inside the two functions target different indexes of the mapping, the protected function modifies the owner's value while the publicly callable function writes to the caller's value. Consequently, these cases are not actual vulnerabilities and can be viewed as false positives. The reason Semgrep did not find these instances is that the Semgrep pattern checks if the index access is identical in the two functions, whereas SIA does not enforce this constraint.

Four of the identified access control inconsistencies were found in functions that invoke a common subprocedure, with the two functions having an inconsistency in access protection. In all four instances, the protected functions execute identical logic, but with fewer and less restrictive checks compared to the public functions. This design allows the owner to execute some privileged operations. Consequently, these cases are not considered vulnerabilities and can be identified as false positives. Again, Semgrep did not find these instances as it does not follow calls to subprocedures.

One of the access control inconsistencies found by SIA represents a unique scenario. This contract facilitates a fee system, where the contract takes a percentage of transactions processed by it. The owner of the contract is allowed to skip this fee and can use the contract's functionalities without paying. The developers of this contract took an unusual approach to implement this requirement. Their implementation is illustrated in fig. 6.3.

Contract variables are used to store the transaction fee rates, the names of these variables in the contract are *_feeAddr1* and *_feeAddr2*. When the owner calls the privileged function, the fees are temporarily set to zero, their operation is executed, and the fees are set back to the original value. A more conventional approach would be to store these fee rates as constant values and use a function parameter to enable or disable

the fees. Additionally, before every transaction from an external user, the fee rates are set to the values intended for these users anyway, as seen in the *_transfer* function. Consequently, this operation essentially changes nothing, as the fees are always already set to the correct value. This private function is called from publicly available functions, SIA discovered the inconsistency because of its interprocedural control flow capability. However, because both the publicly callable function and the privileged function write to the contract variables storing the rates, the contract shows up in SIA as an access control inconsistency. The instance may not fit the core definition of the pattern, but it still shows an unmaintainable coding practice that should be refactored by the developers. Semgrep does not find this instance as the public function only writes to the contract variables in a subprocedure.

Additionally, an interesting point about the previously discussed contract is that two different types of inconsistencies were discovered in it. This contract showed both the access control inconsistency described previously as well as a message sender inconsistency. It could be assumed that this is due to a general lack of coding knowledge or care about code quality by the developers of this contract.

To summarize, both tools found the same inconsistencies in 4 of the contracts in the data set. Additionally, SIA found one message sender inconsistency that Semgrep was not able to discover. One contract flagged by SIA with an access control inconsistency uncovered some unusual coding practices that should likely be refactored. SIA also flagged 6 contracts with access control inconsistencies and 2 contracts with reentrancy guard inconsistencies that can be considered false positives. In this data set, only instances of 3 out of the 8 inconsistencies were discovered.

6.2.2 Extended data set

In the experiment running on the larger data set, more types of inconsistencies were detected by both tools. In total, matches for 7 of the 8 inconsistency patterns were found. The 1605 contracts investigated here are also part of the smart contract sanctuary. Since the number of flagged contracts is much higher in this setup, the individual instances cannot all be manually investigated.

A table containing the number of findings for each of the inconsistency patterns can be found in tab. 6.2. What becomes immediately clear is that SIA flags a lot more contracts than Semgrep, finding a total of 407 contracts with inconsistencies while Semgrep only detects issues in 116.

The only inconsistency the two tools agree and match on is the message sender inconsistency, where both find the same inconsistencies in 86 contracts. There are 24 message sender inconsistencies that only Semgrep detected and there is 1 such inconsistency that only SIA detected. The inconsistency found only by SIA is due to the call to *msg.sender* residing inside the constructor of the contract, that is something the Semgrep pattern is not able to detect.

[1] Pattern not available in Semgrep, see 5.1

	Both	Only SIA	Only Semgrep
Contract call inconsistency	0	3	0
Access control inconsistency	0	264	1
Reentrancy guard inconsistency	0	43	0
Message data inconsistency	0	0	0
Message sender inconsistency	86	1	24
SafeMath inconsistency	0	0	5
Send/Transfer inconsistency	0	1	0
Unchecked block inconsistency[1]	-	9	-
Total	86	321	30

Table 6.2: Comparison of the number of contracts with detected inconsistencies in the extended data set

The contract call inconsistency pattern was exclusively flagged by SIA, the tool found 3 such instances. Since there is such a low number of contracts flagged with this pattern, the instances can be verified manually. In the first of these instances, the dereference of the address happens using the following call:

```
address(...).supportsInterface(type(IERC1155).interfaceId);
```

The *supportsInterface* is a function provided by the ERC165 standard. Similarly to the exception for *address.code* described in section 5.2.1, this function can run on an empty address unproblematically and will simply return *false*. The call is therefore not a dereference and the instance is a false positive.

The other two instances of the contract call inconsistency found by SIA are flagged because of the interprocedural control flow checks. A variable is dereferenced in a function, then a subprocedure is called with the variable as a parameter and the variable is checked for the zero address there. However, in both contracts, the subprocedure is called from multiple points in the program, making the check necessary and therefore a false positive.

Access control inconsistency is the pattern that is by far flagged the most often by SIA. Semgrep only found one such instance. Interestingly, this detection occurred on a contract that is not identified as an inconsistency by SIA. Upon manual investigation of the instance detected by Semgrep, this flag seems to be a false positive as the variable flagged by Semgrep is only ever written to in functions with the *onlyOwner* modifier.

The instances of reentrancy guard inconsistency are all exclusively found by SIA as well. None of the tools found a single occurrence of the message data inconsistency, making this the only pattern without any findings.

Semgrep is the only tool to have found SafeMath inconsistencies. The instances have been manually checked and can all be classified as potential issues. Here is a snippet of one of the offending contracts:

```
1  function balanceOfClaimable(address account) returns (uint256) {
2      ...
3      uint32 today = uint32(blockTimestamp().div(SECONDS_PER_DAY));
4      ...
5  }
6  ...
7  function balanceOfUnstakable(address account) returns (uint256) {
8      ...
9      uint32 today = uint32(blockTimestamp() / SECONDS_PER_DAY);
10     ...
11 }
```

The division operation used here is in all likelihood never able to to under- or overflow, making the use of SafeMath for it unnecessary. In one of the flagged instances, the unchecked operator is used in an *initialize* function while the SafeMath functions are used in all other instances, in all likelihood the values used in the operations are therefore unable to overflow. Consequently, that instance is not an actual vulnerability, however, using the SafeMath function may still be preferrable.

The send/transfer inconsistency found by SIA is the only instance of this type found by either tool. A closer look at the contract where the inconsistency was flagged reveals that the two calls to *send* and *transfer* take place in the same function. However, the two functions are called on different addresses, indicating an issue with SIA. The inconsistency's definition says that the pattern should only match when the functions are called on the same address, this instance is a false positive. In the example, the *send* function first tries to send an amount of ETH to one address and, in case of failure, *transfer* is used as a fallback to send the funds to another address. This is a valid way of using the functions.

The unchecked block inconsistency cannot be taken into account for this comparison as Semgrep does not implement this pattern, however, for completeness, it is still included in the table. SIA found a total of 9 such instances.

6.3 Discussion

The experiments show that the different architecture of the two tools does not just have a big impact on the implementation process of the patterns, but also makes a significant difference on the kinds of inconsistencies they are good at finding, how many false positives they flag and for what reasons the tools fail to fulfill their purpose.

The only pattern for which both tools showed a similar effectiveness is the message sender inconsistency. Since this is a relatively simple pattern, both Semgrep and SIA are able to find a large number of instances of this inconsistency and match on most of the contracts that they flag. Semgrep seems to have a slight advantage for finding this pattern. It can be speculated that this is due to the deep expression matching that Semgrep is able to do and its ability to run on contracts that cannot be compiled and therefore cannot be analyzed by SIA.

However, Semgrep has some problems as well. For edge cases, the framework regularly has issues or does not behave as expected. Examples of this are the instances where a pattern is instantiated through a constructor or one of the newer language features like the *receive* function are used. In these cases, Semgrep's incomplete support for the Solidity language shows and leads to missed findings. Additionally, the experimental state of Semgrep for the language has a negative impact on the development experience and maintainability of patterns. For example, extending the patterns with special cases for occurrences of the patterns in connection with constructors is not an expected behavior of the tool. The initial omission of these edge cases cannot just be blamed on missing experience with Semgrep, a strong case can be made that constructors behave like other functions in many ways and should therefore also match with function patterns in Semgrep. Adding variants with constructors to all the pattern definitions would also have a significant negative effect on the readability of the patterns. For other missed instances of patterns, like the cases with the *receive* function, there is no way in the current version of Semgrep to get the patterns to work with it at all. Even explicitly defining a pattern with this function name will not allow matches, the tool simply ignores the function.

Finally, Semgrep performed much better on the SafeMath inconsistency. This pattern was only found in the large data set, SIA was unable to flag any of the contracts where Semgrep found the inconsistency. It can again be suspected that the deep match operator for the arithmetic expressions gives Semgrep a significant advantage for this pattern over SIA. All the flagged contracts contain real problems where the SafeMath function is used unnecessarily or where the arithmetic operator should be replaced by the respective safe function.

Another point where Semgrep has an advantage is the speed of analysis. On the large data set, Semgrep took an average of 0.49 seconds per contract for its analysis, on the small data set even less with 0.07 seconds. The large data set contains some enormous contracts, it can be suspected that Semgrep's performance does not scale well for such big programs, as for those contracts the timeout had to be disabled. However, even with the slower time for the second data set, Semgrep still performed its analysis faster than SIA. The Solidity Inconsistency Analyzer took 0.83 seconds per contract in the small data set and an average of 1.17 seconds for each program in the bigger data set. Still, for both tools, the performance is adequate for real time analysis and the analyzers could likely be integrated into code editors for continuous checks.

SIA's ability to follow interprocedural control flow detection makes it much more sensitive for the patterns that can take advantage of it and leads to more findings. Concretely, this affects the access control and reentrancy guard inconsistencies. This can lead to a high false positive rate for the mentioned patterns. However, it also allows SIA to find real issues like in the described contract with unusual coding practices that was flagged with an access control inconsistency. A redefinition of the affected patterns influenced by the analysis of the discovered true positives could lead to a better performance of the tool in those patterns. For example, the reentrancy guard inconsistency might be extended with a restriction that the unprotected function must execute a call to an external contract.

This would likely eliminate a number of false positives, as the initial definition of the pattern requiring the functions with the inconsistency to only write to the same contract variables is not a restrictive enough scope. Other possible improvements for SIA include changes to the expression matching in the send/transfer inconsistency and including the index accessed for write operations to mappings. These changes may eliminate a number of false positives that were found in the two experiments.

The only pattern without any findings in both data sets was the message data inconsistency and, after eliminating the false positive by SIA, the send/transfer inconsistency. It could be suspected that these patterns are either very uncommon in deployed contracts or that the patterns were defined too restrictively. Concerning the send/transfer inconsistency, it seems unlikely that a contract would call the two functions on the exact same address. A redefinition of the pattern to better fit real life occurrences may be in order. As for the message data inconsistency, it is unknown why no instances of it were found. In principle, the pattern is defined the exact same way as the message sender inconsistency, which found a number of occurrences in both tools. The reason may be that message data is simply accessed far less often than the message sender address and therefore fewer such inconsistencies are possible.

Overall, the experiments highlighted some issues in both code analysis tools. The results uncovered some possible adaptations to the patterns that could improve their effectiveness. However, both SIA and Semgrep showed some strengths on certain types of patterns that should be further built upon and that can lead to the discovery of problematic inconsistencies in real world smart contracts.

CHAPTER 7

Conclusion and Future Work

The aim of this work was to implement inconsistency patterns in a static analysis tool so that inconsistency bugs can be detected in Solidity smart contracts. Initially, a set of the most relevant inconsistency patterns for Solidity were identified and defined. These patterns were then implemented in Semgrep, a pattern-matching framework. The framework allowed for a relatively high speed of development, which was utilized to iteratively refine the selected patterns. However, Semgrep also showed some weaknesses, mainly stemming from the incomplete support for the Solidity language. These issues lead to unexpected behavior, the inability to implement certain patterns or edge cases and had a negative impact on the development experience and maintainability of the patterns.

To address some of these problems, SIA, the Solidity Inconsistency Analyzer, was developed. The tool works by translating Solidity code to Datalog facts and querying the data using a high performance Datalog engine. The patterns can be implemented as Datalog queries, leading to a higher level of expressiveness and completeness of the patterns. A disadvantage of this approach is that all input must be compilable Solidity code, whereas Semgrep can also run on incomplete programs.

The two tools were run on two data sets of real world Solidity smart contracts and their results were compared. In one of the experiments, the data set contained a smaller number of contracts, to allow for the detailed investigation of all findings. The second data set contained a higher number of contracts to evaluate how well the tools scale on bigger contracts. The findings suggest that both tools have their strengths and weaknesses. The overlap of instances that were found by both tools is relatively small, they only show similar effectiveness on one of the eight implemented inconsistency patterns.

Some patterns were found by both tools while others were only flagged by one of them. Two of the patterns were, apart from one false positive, not discovered at all in the data set. A number of real problems caused by inconsistencies were found by both tools.

In the experiments, Semgrep showed its strengths in the speed of analysis it is able to perform. The average time taken is dependent on the size of the analyzed contract, but Semgrep outperformed SIA's speed by a factor of 11 in the smaller data set and by a factor of 2 in the extended data set. Another advantage of Semgrep is its powerful deep expression operator, which is able to match contained subexpressions with a higher level of success compared to SIA on certain patterns.

SIA is able to find a higher number of instances compared to Semgrep on certain inconsistencies thanks to its better understanding of interprocedural control flow. Concretely, this affects the access control inconsistency and reentrancy guard inconsistency. The Solidity Inconsistency Analyzer is able to run on all of the selected patterns, while Semgrep could not be used to implement one of the eight inconsistencies. Since the official Solidity compiler *solc* is used for the retrieval of program structure data, the tool is able to analyze all language concepts and is more easily extensible.

The experiments also highlighted some possible improvements and future work for the patterns and tools. In Semgrep, the issues related to language support should be fixed to enable more complete pattern definitions. SIA's interprocedural control flow detection is a powerful capability, but in its current form leads to a high number of false positives. Further refinement of this feature and the patterns utilizing it is necessary. Some exceptions for patterns and a bug relating to expression matching were discovered in the experiments in SIA as well.

In its current form, SIA also lacks support for the integration into code editors and, for public availability, requires further refinement and maturity. For integration into code editor plugins, the output of standardized formats like SARIF[1] may be necessary. Currently, SIA only supports providing output in a proprietary JSON format or in formatted messages on the command line.

Overall, the experiments called attention to some of the issues in both code analysis tools. The results uncovered some possible adaptations to the patterns that could improve their effectiveness. However, both SIA and Semgrep showed some strengths on certain types of patterns that should be further built upon and that can lead to the discovery of problematic inconsistencies in real world smart contracts.

[1] https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html

Overview of Generative AI Tools Used

For the translation of the english abstract to the german Kurzfassung, Google Gemini 2.0 Flash was used to translate the technical phrases like “inconsistency patterns” or “static code analyis tool”.

List of Figures

List of Tables

Glossary

AST Abstract Syntax Tree. A data structure representing the structure of a program. 6, 15, 27, 34–37, 46

CI Continuous Integration. The software engineering practice of frequently integrating code changes in a central repository. 13, 28

CLI Command Line Interface. 14, 15, 28, 37, 46

CRUD Create, Read, Update, Delete. A paradigm of simple data management applications. 5

CSV Comma Separated Values. A file format where columns are separated by commas. 37

DAO Decentralized Autonomous Organization. DAOs are types of smart contracts that are designed to coordinate decisions within their stakeholders. 1

DSL Domain Specific Language. A custom language specialized to a specific domain. 6, 7

ETH Ether. The native crypto currency of the Ethereum block chain. 1, 23, 54

EVM Ethereum Virtual Machine. The computational engine executing smart contracts in the Ethereum network. 6–8, 23

IDE Integrated Development Environment. 2, 28, 37

JSON JavaScript Object Notation. A popular, human readable data-exchange format. 15, 34, 46, 58

RBAC Role-based Access Control. The practice of grouping permissions together into roles, used for access control. 12

SAST Static Application Security Testing. Static code analysis with the purpose of identifying security vulnerabilites. 6

TSV Tab Separated Values. A file format where columns are separated by tabs. 36

WSL Windows Subsystem for Linux. A feature allowing the access of a Linux system from inside Windows. 34

YAML Yet Another Markup Language/YAML Ain't Markup Language. A human readable data serialization language. 6, 29

Bibliography

[BGL+20] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *PLDI*, pages 454–469. ACM, 2020.

[Bin07] David Binkley. Source code analysis: A road map. In *Future of Software Engineering (FOSE '07)*, pages 104–119, 2007.

[BOP+19] Gaurav Batra, Rémy Olson, Shilpi Pathak, Nick Santhanam, and Harish Soundararajan. Blockchain 2.0: What's in store for the two ends—semiconductors (suppliers) and industrials (consumers). *New York: McKinsey Digital. Retrieved April*, 10:2024, 2019.

[Bro17] Ryan Browne. 'Accidental' bug may have frozen $280 million worth of digital coin ether in a cryptocurrency wallet. *CNBC*, 2017. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html.

[CB16] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 332–343, New York, NY, USA, 2016. Association for Computing Machinery.

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166, 1989.

[Cyf] Cyfrin.io - reentrancy example. `https://www.cyfrin.io/glossary/re-entrancy-hack-solidity-code-example`.

[ECC01] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72. ACM, 2001.

[Fou24] Ethereum Foundation. The history of Ethereum. *https://ethereum.org*, 2024. `https://ethereum.org/en/history`.

[GBSS19] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *ICSE*, pages 1176–1186. IEEE / ACM, 2019.

[JSS16] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, pages 422–430. Springer, 2016.

[LSS20] Bowen Liu, Siwei Sun, and Pawel Szalachowski. Smacs: Smart contract access control service, 2020.

[Mar19] Steve Marx. Stop using solidity's transfer() now. *diligence.consensys.io*, 2019. https://diligence.consensys.io/blog/2019/09/stop-using-soliditys-transfer-now/.

[Nag] Bence Nagy. Detect complex code patterns using semantic grep. `https://owasp.org/www-chapter-dorset/assets/presentations/2020-10/OWASP%20Dorset%202020-10-08.pdf`. Retrieved 2025-02-20.

[NSB22] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 532–543, New York, NY, USA, 2022. Association for Computing Machinery.

[OE23] Martin Ortner and Shayan Eskandari. Smart contract sanctuary. 2023.

[O'M] O'Malley, Luke. Important updates to Semgrep OSS. `https://semgrep.dev/blog/2024/important-updates-to-semgrep-oss/`. Retrieved 2025-02-20.

[Ope25] Documentation OpenZeppelin. OpenZeppelin Utils - Context. *docs.oppenzeppelin.com*, Accessed 06.03.2025. `https://docs.openzeppelin.com/contracts/5.x/api/utils#Context`.

[Pra22] Zubin Pratap. Reentrancy Attacks and The DAO Hack. *Chainlink*, 2022. `https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/`.

[Roz23] Shamshod Rozikov. Private variables in solidity are not private. 2023.

[sol] Solidity documentation. `https://docs.soliditylang.org/en`.

[TDD+18] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In *CCS*, pages 67–82. ACM, 2018.

[TVI+18] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. SmartCheck: Static analysis of Ethereum smart contracts. In *WETSEB@ICSE*, pages 9–16. ACM, 2018.