Brandenburgische
Technische Universität
Cottbus - Senftenberg

Faculty 1

Chair of Practical Computer Science

Software Systems Engineering

Study Program Cyber Security

Master Thesis

Design and Implementation of Taint Analysis for GraphQL-based Web Applications

presented by

Osama Zaid Saleem Al-Wardi
(Birth date: 29.11.1998)

Examiners:
Prof. Leen Lambers
Prof. Andriy Panchenko
Advisor:
Lucas Sakizloglou

Cottbus, April 5, 2023

Abstract

GraphQL has emerged as a popular query language for APIs due to its flexibility, efficiency, and ease of use. However, the adoption of GraphQL also brings new security challenges to be addressed. The security of GraphQL APIs has become a crucial concern for businesses and organizations that rely on these APIs to provide critical services to their customers. Despite the importance of the topic, there is a lack of security testing techniques and approaches that are specifically designed for GraphQL.

In this thesis, we propose a new taint analysis approach that combines static and dynamic taint analysis techniques to identify broken access control and data leaks in GraphQL APIs. Our approach builds on existing concepts in taint analysis and adapts them to the unique features of GraphQL. In this thesis we use graph transformations as a formal method for static taint analysis combined with dynamic taint analysis which is able to interact with the live API. This way it is possible to identify potential vulnerabilities through static taint analysis and verify them through dynamic taint analysis.

We implement each stage of our approach using a combination of tools and python programs. We evaluate the feasibility of our approach by applying it to a case study on GitHub's GraphQL API and executing testcases that simulate common access control violations and data leaks that can occur to a GraphQL API. We then compare our approach with a similar approach which also combines static and dynamic analysis. This comparison highlights the strengths of our approach in identifying broken access control, data leaks as well as highlighting points that we can adapt to our approach.

Table of Contents

Acronyms

STA	Static Taint Analysis
DTA	Dynamic Taint Analysis
API	Application Programming Interface
JSON	JavaScript Object Notation
CDA	Conflict and Dependency Analysis
CPA	Critical Pair Analysis
SAST	Static Application Security Testing
DAST	Dynamic Application Security Testing
SDL	Schema Description Language
SOAP	Simple Object Access Protocol
REST	Representational State Transfer
OWASP	Open Web Application Security Project

1 Introduction

GraphQL is an increasingly popular query language for APIs due to its flexibility, efficiency, and ease of use. However, this technology brings some unique security challenges that need to be addressed. According to the *Open Web Application Security Project*, the most common GraphQL vulnerabilities are abusive queries that cause denial of service, broken authorization including indirect object references and broken authentication [owaa] [owab]. Despite these unique challenges, there is currently a lack of security testing tools that are specifically designed for GraphQL. Therefore, existing tools and techniques that are mostly used to test and secure traditional REST APIs may not be sufficient for GraphQL, given its unique structure and query language. There is a need for new approaches and techniques that are specialised in creating unique test-cases that test the unique features of GraphQL. In this thesis, we propose a new design and implementation of taint analysis that combines static and dynamic taint analysis to ultimately create a *static application security testing* (SAST) approach integrated with a *dynamic application security testing* DAST tool to identify security vulnerabilities in GraphQL APIs.

Our approach aims to create a taint analysis technique adapted to handling the unique features of GraphQL. We aim to tailor static taint analysis to GraphQL APIs by formalizing GraphQL schemas to typed graphs and graph transformation rules and then performing a static analysis technique called dependency analysis to identify potentially sensitive sources and sinks of information. Then we consider access control roles in APIs to identify which data is considered confidential in that system. By identifying sources/sinks as well as the sensitive information, it is possible to perform dynamic taint analysis and interact with the API to send dangerous queries. Using this combination of static and dynamic taint analysis, we have the ability to identify potential security vulnerabilities by using static analysis and verify if the vulnerability actually exists by using dynamic analysis.

In the next chapters, we provide the theoretical background for this work and related work. Then, we outline our new approach and dive into the design and implementation of the approach. Finally, we evaluate the approach and compare it with another known tool. The contribution of this thesis is to improve the state of the art of GraphQL testing by providing a new approach and a tool for developers and security professionals to test their software. Creating tailored GraphQL solutions is essential to ensure that this technology continues to be adopted and used in a safe and secure manner.

2 Background

In order to understand taint analysis and its applicability to GraphQL, it is essential to understand the fundamentals of taint analysis as well as GraphQL. This chapter presents the topic of application security testing to understand the general background of taint analysis. In addition, this chapter introduces the topic of web services and motivates why GraphQL is a gaining traction. Finally, the basics of graph transformations and dependency analysis will be introduced as they form the theoretical basis for this work.

2.1 Application Security Testing

Application security testing is a crucial component of any secure software development lifecycle. It involves testing an application for vulnerabilities and weaknesses in order to identify and mitigate potential security risks. There are several methodologies of application security testing that can be performed, each with their own strengths and weaknesses.

Static Application Security Testing (SAST) is a type of application security testing that involves analyzing an application's source code for potential vulnerabilities [Li20]. SAST tools scan an application's source code to identify coding errors, insecure configurations, and other security weaknesses. SAST is useful because it can detect vulnerabilities fairly quickly and early in the development process, before the application is deployed to a production environment. One of the key advantages of SAST is that it enables the early detection of security vulnerabilities. By analyzing the source code before the application is deployed to production, developers can identify and address potential security issues before they become major problems. Also SAST can provide developers with a holistic view of the security of their application, allowing them to address issues before they are exploited. Finally, SAST can be automated, making it efficient and scalable for large codebases. This can help to reduce the time and cost associated with manual security testing.

While SAST is a powerful testing technique, it does come with some disadvantages. SAST does not provide insight into how an application may behave in a live environment, and it may produce false positives or miss certain types of vulnerabilities [sas]. SAST can generate a large number of false positives, which can be time-consuming to review and validate. Additionally, SAST has a limited scope, and may not detect all types of vulnerabilities, particularly those related to runtime behavior or environmental factors. Finally, SAST can only analyze code that is available for analysis, and may not cover code that is not included in the analysis, such as third-party libraries or dy-

namically generated code.

Dynamic Application Security Testing (DAST) is another type of application security testing that involves testing an application in a live environment [Son21]. DAST tools simulate attacks on an application to identify vulnerabilities that may not be visible during static testing. DAST is useful because it provides insight into an application's behavior in a real-world environment, and can identify vulnerabilities that may only be exploitable under specific conditions. DAST provides a realistic view of the security of an application, as it tests the application in a running state. Additionally, DAST can help to identify vulnerabilities that are not detectable through SAST, such as those related to runtime behavior or environmental factors. Another advantage of DAST is that it can test the entire application, including third-party components and dynamically generated code, providing a more comprehensive view of the security of the application.

While DAST can provide valuable insights into the security of an application, it does have some limitations. One of the main challenges is the potential for false negatives, as DAST relies on a set of predefined tests that may not capture all potential vulnerabilities. Additionally, DAST can be time-consuming and resource-intensive, as it requires running the application in a test environment [das]. Finally, DAST may not be suitable for testing applications with complex or distributed architectures, as it may be difficult to replicate the entire application environment in a test environment.

While both SAST and DAST have their strengths and weaknesses, a combination of both methodologies can provide a more comprehensive approach to application security testing. In this thesis we aim to create a taint analysis approach, combines elements of both methodologies to provide an accurate analysis of the application's security posture. Tools that implement this approach can perform static analysis but at the same time they are designed to interact with the application in real-time, providing immediate feedback on potential vulnerabilities during development and testing.

2.2 Taint Analysis

Taint analysis is a security technique that involves tracking the flow of sensitive data through a program to identify potential vulnerabilities [YY17]. The origins of taint analysis can be traced back to the late 1990s, when researchers began exploring techniques for identifying and mitigating security risks in software applications. The technique gained widespread recognition in the early 2000s, following the discovery of several high-profile security breaches that were caused by flaws in application code [STFW01], [NS05].

Taint analysis is particularly useful for identifying vulnerabilities related to input validation and data leaks, as it allows developers to track how user input is processed by the application and identify potential points of weakness. For example, taint analysis can be used to identify injection attacks by tracing the flow of user input through the application and identifying any points where untrusted data is used in potentially dangerous ways. Taint analysis is also useful for identifying privacy violations, as it can be used to track the flow of sensitive data through the application and identify any points where this data may be leaked or exposed.

Taint analysis defines sources, sinks, and propagation rules [YY17]. Information originates from a source, reaches a sink, and propagates through the different program paths. It is possible to track certain information from source to sink and ensure that sensitive information does not reach unintended sinks. One of the first steps to performing taint analysis is to identify which information needs to be tracked. When using taint analysis to find security issues, it is interesting to track untrusted information or confidential information such as, untrusted user input or private user data. This untrusted information is referred to as a "taint" [STFW01]. Therefore taint analysis works by adding taint tags to the data entering the program from the sensitive sources, then the propagation of the tainted data is monitored carefully. At the end of the analysis, the flow of tainted data through the program can be checked to see whether the data reaches an unintended sink. The identification of sensitive sources, sinks and tainted data is done manually or semi-automatically.

Taint analysis can be done in two ways: *static taint analysis* (STA) and *dynamic taint analysis* (DTA). In STA, the analysis is conducted without executing the program. *"Detecting Format String Vulnerabilities with Type Qualifiers"* published in 2001 [STFW01], is one of the first papers that use STA to detect format string vulnerabilities in C programs. However, it is not the first paper that formally introduced the concept of taint analysis, as some programming languages had already used taint analysis, such as Perl's taint mode [per], to prevent malicious users from executing commands on a host computer. The paper defined the term "taint" by describing all program inputs that could be controlled by the adversary as "tainted". The paper discusses the use of type qualifiers as well as a tool for performing STA.

Dynamic Taint Analysis (DTA) is a taint analysis technique used for identifying and tracking user-controlled input data that can influence the program during it's execution. The approach might involve different techniques such as program instrumentation to track the flow of tainted data during program execution. One of the earliest and most influential papers on the topic was published in 2005 with the name *"Dynamic*

Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software" by James Newsome and Dawn Song [NS05], which introduced one of the first dynamic taint analysis approach for detecting and analyzing software exploits. In this paper, the authors introduced a dynamic taint analysis approach that could be used to detect and analyze software exploits, and demonstrated its effectiveness by identifying previously unknown exploits in widely used software.

While taint analysis can be a powerful tool for identifying security vulnerabilities in software, it does come with some disadvantages. One significant issue for STA is the potential for false positives, which can generate a large number of irrelevant results that need to be reviewed and validated. Additionally, taint analysis is effective in identifying vulnerabilities related to the data flow of the program such as data leaks and injection attacks. It does not detect other vulnerability categories in the OWASP Top 10 list, particularly those related to cryptography or dential of service attacks. Another challenge of using DTA tools is the performance impact it can have on the execution of the program. The additional overhead required to track data flows can slow down the program, which can be problematic for large and complex applications.

2.3 Web Services: Performance and Security Comparison

The use of web services and web API is widely seen on the internet today. A web API could be defined as a service offered by a service provider, available via the web and which is a key element in the integration of systems, of different platforms, programming languages and technologies [BTZTC+20]. Web APIs can be based on many different protocols such as Simple Object Access Protocol (SOAP) which provides security and reliability, fewer errors and asynchronous requests. Therefore there are business, banking and payment information systems that use it today. In addition representational state transfer (REST) is another approach with a strong number of advocates in which web API solutions can be developed simply representing and exposing the resources of the system, and transferring data over HTTP. In this section, we will discuss the performance differences and security considerations of these three protocols. This will help to develop a better understanding of the trade-offs associated with each protocol.

GraphQL offers an alternative to the REST approach and mainly attempts to overcome the shortcomings of the REST specification. Some of the shortcomings of REST are over-fetching and under-fetching, meaning that queried data is sometimes delivered with too much or too little information. Over and under fetching are both problematic because they are either inefficient or provide information that will not be used in the case of over-fetching or force the need to make extra queries in the case of under-

fetching. Therefore GraphQL attempts to solve this exact problem.

A comparative analysis of the performance differences between SOAP, REST and GraphQL has already been performed and it has some interesting results [BTZTC+20]. In the comparative study, three web services were created for the respective specification and the web services were hosted in the same environment with many controlled variables in order keep such variables from influencing the results of the study. The results obtained from this experiment showed that the performance attributes such as throughput and response time in GraphQL are much lower than the performance attributes of REST and SOAP in the environment in which they were examined. Web services based on REST show better response times and throughput than SOAP [BTZTC+20].

When it comes to security, GraphQL has slightly different threats and attack vectors than REST and SOAP. One of the potential security risks with GraphQL is the possibility of using abusive queries and batching attacks that can cause denial of service (DoS) by overloading the server with too many requests or with complex requests that take a long time to process. GraphQL also has potential vulnerabilities related to error messages that may inadvertently disclose sensitive information. Furthermore, unlike SOAP and some REST implementations that enforce access control by default, GraphQL does not enforce authorization by default. Therefore, it is up to the application to perform authorization enforcement to ensure that only authorized users can access certain data or perform certain actions [owab].

Despite these potential security concerns, GraphQL has implemented some measures to address them. For example, it provides features such as query complexity limits and query depth limits that can help mitigate the risk of abusive queries and DoS attacks. Additionally, GraphQL allows developers to define custom error messages, which can help prevent sensitive information from being disclosed unintentionally. Finally, GraphQL provides some flexibility in terms of how authorization can be enforced, allowing developers to implement their own authorization logic as needed.

2.4 GraphQL

GraphQL is an open-source query language for web APIs. It was developed internally by Facebook in 2012 before being publicly released in 2015. This query language introduces a new type of web-based data access that presents an alternative to REST and SOAP based interfaces. One of the key differences between GraphQL and other APIs is its focus on a strongly-typed schema. This schema defines the structure of the data available through the API, enabling clients to easily discover and interact with it. Therefore GraphQL offers the ability to define precisely the data needed for the

query, replacing multiple REST requests with a single call thus eliminating issue such as over-fetching and under-fetching. Since its release, GraphQL has gained significant momentum and has been adopted by an increasing number of users including Coursera, Github, GitLab, Neo4J, and Pinterest [HP18].

In GraphQL, queries and mutations are two types of operations that can be performed on the server. A query is used to read data from the server. It is similar to a GET request in REST. Queries are used to request specific fields from an API and can be nested to request related data. A mutation, on the other hand, is used to modify data on the server. It's like a POST, PUT, or DELETE request in REST. Mutations are used to create, update, or delete data. For example, a mutation might add a new user to a database or update an existing user's email address.

When it comes to the architecture of GraphQL, there are several key components involved. On the front end, there is the client application, which sends queries to the server and receives the data in response. On the back end, there are several elements, including the schema, which defines the structure of the data, and the resolver functions, which are responsible for fetching the data from external APIs or databases. One of the unique features of GraphQL is its use of resolvers, which act as middle-ware between the client and the data source. Resolvers are responsible for fetching the data required to fulfill a query, and they can be used to retrieve data from a variety of sources, including databases, external APIs, or even other GraphQL APIs. Fig. 1 illustrates the architecture of GraphQL APIs.

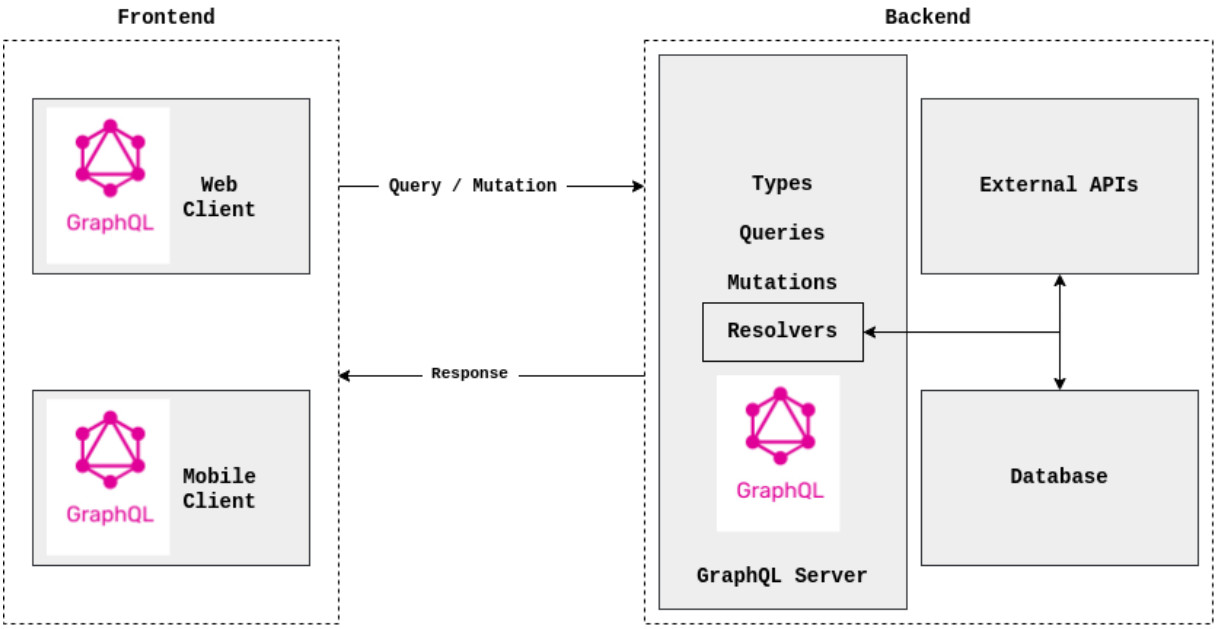

Figure 1: GraphQL API Architecture

2.5 Typed Graphs and Graph Transformations

One of the reasons for GraphQL having better performance metrics than SOAP and REST is that GraphQL leverages the power of the graph data structure, thus the name "Graph Query Language". The book *"Graph Transformation for Software Engineers"* by R. Heckel and G. Taentzer is a comprehensive book that covers the main literature on graph transformations [HT]. The authors trace the origin of graph transformations back to the early 1970s, when they were first introduced as a means of modeling and analyzing concurrent systems. This book is used as a primary reference in this thesis to introduce the concepts of typed graphs, graph transformations and rules.

Every GraphQL schema can be represented as a typed graph because it specifies data types and the relationships between them. In a typed graph, each node is assigned a type that describes its role or purpose in the graph. Similarly, edges between nodes are labeled with a specific relationship type. Fig. 2 represents an example of a typed graph. In this graph, vertices *Super*, *Client* and *User* represent different types. Edges represent connections between nodes and demonstrate their relationships.

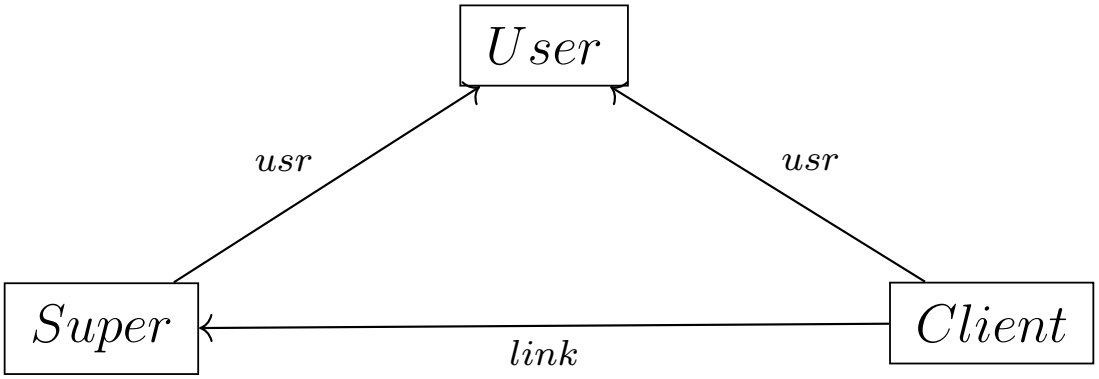

Figure 2: Typed Graph

The range of notations and techniques supporting the systematic manipulation of graphs using pattern-based rules is referred to as graph transformations [HT]. A *graph transformation rule* (GT Rule) specifies the changes, actions and conditions under which a graph transformation can take place. Each rule consists of a left-hand side, specifying the graph before the change, and a right-hand side, corresponding to the graph after the change. Fig. 3, shows an example of a graph transformation rule *endClient* which performs a deletion action to the client node *c:Client* and it's attached edges.

Often, in a graph transformation system the application of one rule at one match may disable other choices available, creating a conflict. Similarly, one rule application may enable another, leading to a dependency [HT]. *Critical pair analysis* is a technique used to analyze GT rules to detect potential conflicts and dependencies that may arise when applying two or more GT rules to a graph. In this thesis we are only focused on de-

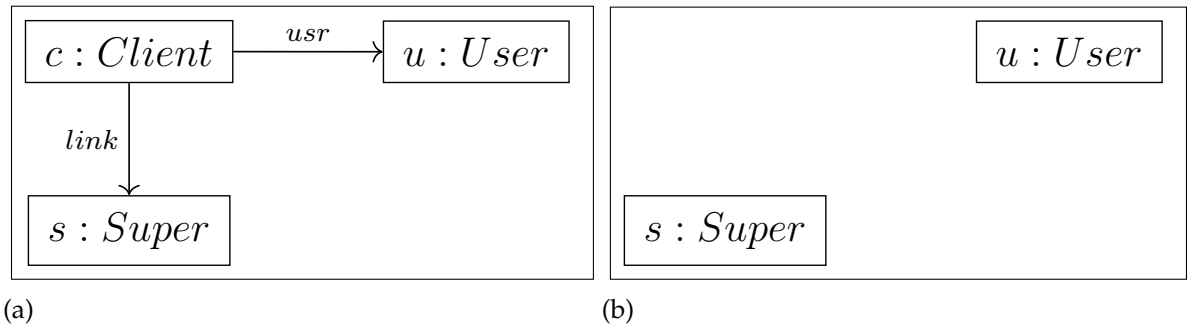

Figure 3: (a) Left Hand Side of the Rule, (b) Right Hand Side of the Rule

pendencies. To perform dependency analysis, a matching algorithm is typically used. The matching algorithm attempts to track nodes and edges that are being created or deleted by the transformation rules. This is as a type of taint analysis when considering rules that create edges as sources of taint and rules that delete nodes or updates them as sinks where taint can flow. Dependency analysis can be automated using software tools such as *Henshin* [ABJ+10] and AGG [agg] as a part of critical pair analysis. This allows for a set of GT rules to be analyzed. Later in this work we will illustrate how it is possible to generate testcases using the results of dependency analysis.

3 Related Work

In this chapter we review the state of the art of related scientific work and proposed solutions. We categorise related works by the testing techniques they utilise. In this chapter we classify related works in three categories: taint analysis, static application security testing (SAST) and dynamic application security testing (DAST). The different approaches as well as application areas and use cases of each category will be presented as well as the advantages and shortcomings of each technique. This is important in order to see areas for improvement to design a more comprehensive approach.

3.1 Taint Analysis

Taint analysis is a software testing technique that attempts to identify errors by observing the flow of information within a program. There are many taint analysis approaches proposed over the years that are applied on a wide range of application areas. The paper *"FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps"* presents a static taint analysis approach for Android applications called FlowDroid [ARF+14]. FlowDroid uses a precise model of the android application lifecycle which allows the analysis to properly handle callbacks invoked by the Android framework, while context, flow, field and object-sensitivity allows the analysis to reduce the number of false alarms. This approach enables FlowDroid to accurately track the propagation of sensitive information within an app and identify potential security vulnerabilities.

As for dynamic taint analysis (DTA), this technique is more versatile as it can also be used to detect vulnerabilities in realtime by interacting with or monitoring the application. The paper *"TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones"* introduces a DTA approach and tool called TaintDroid that tracks the flow of sensitive information in real-time on Android smartphones [EGC+10]. TaintDroid is designed to help users understand how their personal information is being used by apps and to protect their privacy by detecting and preventing unauthorized information leaks. The system works by adding taint tags to sensitive data when it enters the device and then tracking the propagation of these tags as the data is used by apps. Over the years many tools have been introduced to perform taint analysis. Table 1 lists some of the most notable taint analysis tools.

While it is apparent that taint analysis can be applied in different application areas, there is still a lack of tools and approaches applying this technique to GraphQL APIs. Snyk Code is currently the only tool that claims to perform STA on GraphQL [snya]. Overall, Snyk Code uses a combination of static analysis techniques including control

flow analysis, taint analysis and pattern matching. As for DTA There is currently not available tool nor approach that is applicable to GraphQL APIs.

Tool Name	Type	Application Domain	Focus Area
FlowDroid	STA	Android Applications	Data Leaks
TAJ	STA	Web Applications	Vulnerability Analysis
Snyk	STA	Application Source Code	Vulnerability Analysis
SAINT	STA	C Code	Data Leaks
TaintDroid	DTA	Android Applications	Data Leaks
TaintEraser	DTA	Desktop Applications	Data Leaks
PoL-DFA	Monitoring	Web Server Programs	Forensic Analysis
CloudTaint	Monitoring	Cloud Applications	Malware Analysis

Table 1: Notable Taint Analysis Tools

3.2 SAST for GraphQL

Static application security testing is a testing technique that analyzes the source code of an application to identify security vulnerabilities without interacting with the live application and without executing the code of the application. There is currently very many SAST solutions on the market including commercial and open source solutions. For this review we narrow the scope to SAST tools that can perform SAST on GraphQL. Below is a state-of-the-art review of some of the most notable GraphQL SAST tools:

1. *CodeQL* - CodeQL [cod] is a powerful SAST tool developed by GitHub that can perform static analysis on GraphQL APIs to identify potential security vulnerabilities. It uses a semantic code analysis engine to provide a comprehensive view of the security of the application.

2. *SonarQube* - SonarQube [son] is a popular open-source SAST tool that can perform static analysis on GraphQL APIs to identify security vulnerabilities. It uses a variety of plugins to support a wide range of programming languages and frameworks.

3. *Snyk* [snyb] - Is a leading security testing platform that provides a range of tools for identifying and addressing security vulnerabilities. Snyk offers a powerful SAST engine called Snyk Code which utilizes a combination of pattern matching, taint analysis and other source code analysis to identify security vulnerabilities.

Snyk's SAST engine also has custom checks that analyze GraphQL APIs to identify potential security risks such as authorization issues, authentication issues, denial of service and many other vulnerability categories.

4. *Graphql-schema-linter* [gqlb] - Is a command line tool to validate GraphQL schema definitions against a set of rules and patterns. It is a tool designed to identify potential issues specifically in GraphQL schema definitions. It checks for various schema mistakes and programming bugs such as deprecated fields, missing descriptions, and non-null fields.

While there are many SAST tools that can detect security vulnerabilities in GraphQL, there are relatively few that are capable of performing taint analysis. One notable exception is Snyk Code, which uses a combination of static analysis techniques to identify security issues in GraphQL source code. However, there is still a need for more advanced taint analysis tools that can accurately track the flow of data through complex GraphQL queries to identify potential vulnerabilities.

3.3 DAST for GraphQL

Dynamic application security testing is a set of techniques that analyse the application in its running state to identify security vulnerabilities. Unlike SAST, which analyzes the application's source code, DAST tools interact with the application to simulate attacks and identify security vulnerabilities. There are several related tools that perform DAST on GraphQL APIs. Below is a review of some of the most notable GraphQL DAST tools:

1. *Schemathesis* [sch] - an open-source tool for testing GraphQL APIs. It uses property-based testing to generate a large number of test cases automatically. It derives fuzzing testcases by parsing Swagger and OpenAPI[1] files for REST APIs. It is also capable of deriving testcases based on GraphQL *schema description language* (SDL) files.

2. *GraphQL Playground* [gqla] - an interactive environment for experimenting with GraphQL. It provides a GUI interface to test GraphQL queries and mutations for debugging or development purposes. It can be used as a DAST tool, however most testing stages are manual or semi-automatic such as creating a testcase specification or and finding appropriate testcase data.

3. *Jest* [jes] - a popular JavaScript testing framework that can be used to test GraphQL APIs. Jest provides a set of assertions for GraphQL, such as validating the test query results and the schema definition. This tool is able to preform DAST by executing testcases and evaluating them in a fully automated fashion. However, the tool does not automate designing the testcases and preparing the testcase values.

[1] https://www.openapis.org/

4. *Postman* [pos] - a popular API testing tool that can be used to test GraphQL APIs. Unlike Jest this tool can generate testcase values in an automated fashion. It can derive testcase values from the API documentation if available. Postman is able to parse API documentations either in OpenAPI format or as a GraphQL schema in SDL format. Postman can then display some default testcase values that can be directly executed. The main shortcoming however is that it lacks automation in the other areas such as testcase design and evaluation.

There are several GraphQL testing tools available in the market, each with its unique features and capabilities. Table 2 shows an overview of the mentioned tools in terms of their levels of automation.

Tool	**Testcase Design**	**Testcase Data**	**Testcase Execution**	**Testcase Evaluation**
Schemathesis	Manual	Automated	Automated	Automated
GraphQL Playground	Manual	Manual	Semi-Automated	Manual
Jest	Manual	Manual	Automated	Automated
Postman	Manual	Automated	Semi-Automated	Manual

Table 2: Capabilities of Current GraphQL Dynamic Analysis Tools

4 Approach

The approach in this thesis aims to combine static and dynamic taint analysis techniques. The proposed approach is based on the idea that STA determines dependent GraphQL queries, meaning pairs of queries for which the first query enables the second query. This process involves analyzing the schema and modeling its queries and mutations as graph transformation rules. After, we can perform dependency analysis on the model and get a set of dependent queries. This allows to narrow the scope to a few pairs of queries representing sources (queries introducing tainted information into the system) and sinks (queries using or displaying tainted information).

In addition, access control policies allow us to infer which information is meant to be kept private for different types of users in the system. DTA can then execute testcase where sources are used by one user to inject tainted data into the system. Then using another user with different access control privileges, we attempt to use sinks in such a way that violates access control policies in order to determine if there are access control violations or data leaks taking place. This technique involves interacting with the GraphQL API and analyzing the response for potential security issues by executing testcases derived from the STA phase. Therefore, DTA is able to verify if the vulnerability is actually present in the application. Fig. 4 is an activity diagram that illustrates our approach.

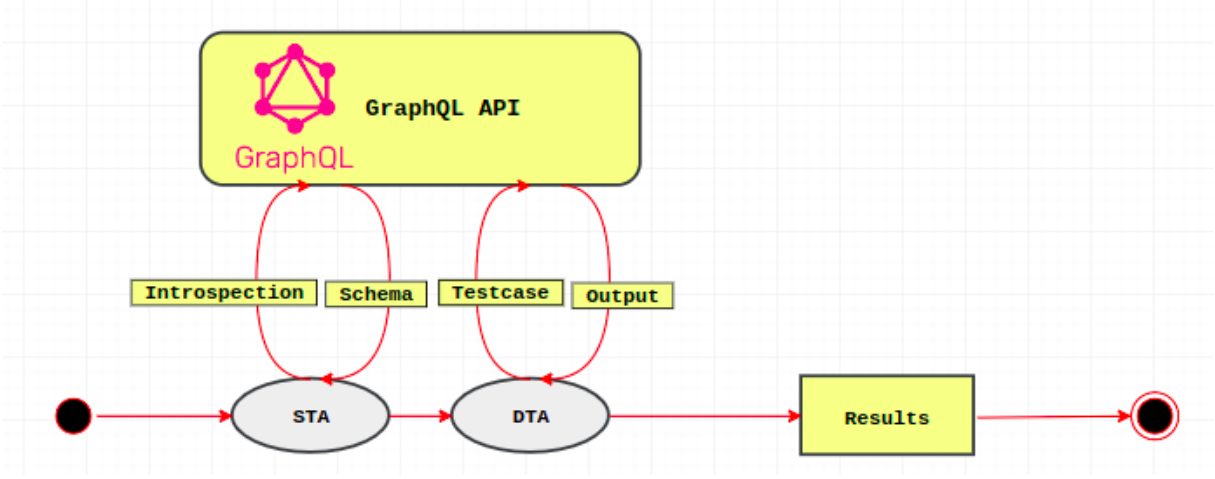

Figure 4: Activity Diagram for Sequential STA and DTA

In the next chapters we aim to describe the exact steps we took for designing and implementing the approach. GitHub's GraphQL API will be used as a case study as it provides a rich API with multiple queries/mutations and a well documented access control policy. We aim to use the Eclipse Modeling Framework along with a tool called Henshin [ABJ+10] for static analysis. We subsequently write python programs that perform the dynamic analysis steps.

5 Static Taint Analysis

In this chapter the design as well as the implementation of the STA phase will be explained. In the design section, each design decision of our static analysis approach will be explained and justified. The entire approach is implemented and tested on a real life case study of GitHub's GraphQL API [2] [3] as it offers a realistic example to test the proposed testing approach. For the implementation we use a simplified version of GitHub's GraphQL schema (only focusing on certain features). In the next sections, we aim to describe the design and technical details of each process, including the tools, in a precise and reproducible way.

The main requirement to start STA is a GraphQL schema. In white-box tests, the schema is typically provided in SDL format. As for black-box engagements, *introspection* must be performed. Introspection is a GraphQL native feature that allows end users to obtain information about the API and it's available queries. After obtaining the schema we can then model the schema in Henshin and create a set of graph transformation rules (GT Rules). This is the necessary input for us to execute dependency analysis and finally get a set of dependent rule pairs. Fig. 5 show an overview of different stages of the STA process including the inputs/outputs for each step.

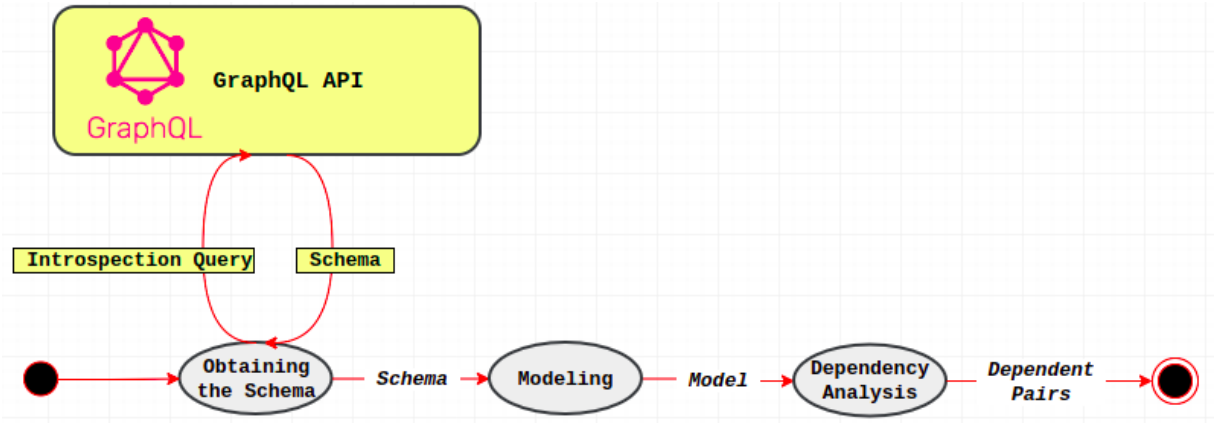

Figure 5: Activity Diagram of our Static Taint Analysis

5.1 Design

Introspection

The first step of static analysis is to obtain the schema of the API. GraphQL offers it's users the ability to query information about the API itself. This can be done using GraphQL's schema introspection feature, which allows the client to retrieve the schema

[2] https://api.github.com/graphql
[3] https://docs.github.com/en/graphql

definition of the API. In white-box scenarios with source code access the schema is already provided. In this case the modeling step can be performed directly. However in black-box engagements, introspection of the API is necessary. Most GraphQL APIs have the introspection feature enabled which allows the user to use GraphQL introspection queries to obtain the current API schema. The schema definition contains the types of data that the API supports, as well as the fields, queries, and mutations that can be executed. The schema obtained is needed to identify some important definitions such as the data type definitions and the query/mutation definitions. These definitions will be used to create a model of the schema in the next step.

Modeling

Based on the introspection results, a model of the schema can be constructed. The Eclipse IDE offers a multitude of modeling frameworks including the Eclipse Modeling Framework or (EMF) [emf]. Henshin [ABJ+10] is a graph transformation system that is based on EMF. It provides a set of tools for modeling and analyzing complex systems using graphs. It allows also to define rules for transforming one graph into another, and provides a mechanism for executing those rules to achieve a desired result. Henshin is built on the concept of graph transformations, where a graph is transformed by applying a sequence of graph transformation rules.

Henshin allows to define a meta-model which is a model that defines the structure, constraints, and semantics of other models. It is used to describe the elements, relationships, and behavior of a domain-specific language or modeling language. This meta-model represents our typed graph and is exactly what is needed to create a model of GitHub's GraphQL schema. Fig. 6 shows the type graph of a simplified version of GitHub's GraphQL schema, modeling only the entities user, repository, issue and project. The corresponding ecore file for this typed graph will be shown in the implementation section.

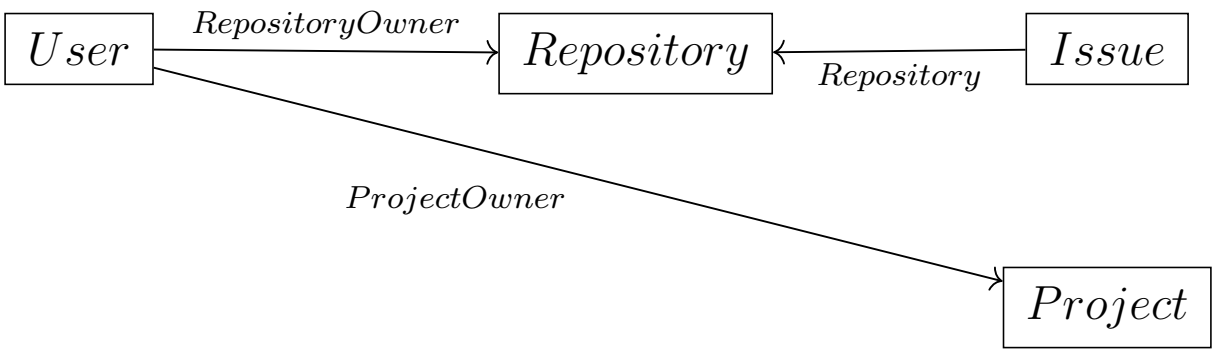

Figure 6: GitHub's Schema Typed Graph

Finally when the modeling steps are completed, our model should define an entire graph grammar and semantics that allows to use dependency analysis to observe the effect of executing different rule pairs and whether such rules have dependencies. In Fig. 7 we see an example of a GT rule called *createProject*, which creates a new project node and links it with the owner.

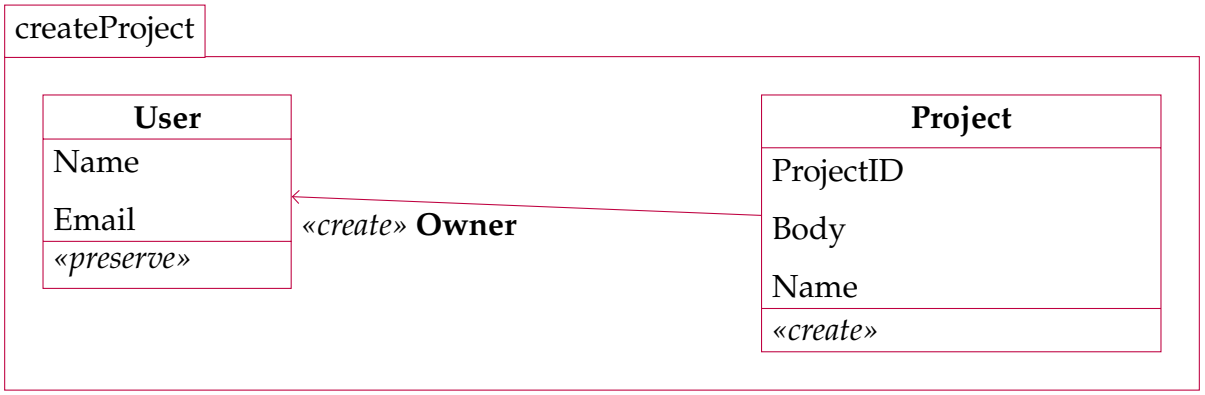

Figure 7: Graph Transformation Rule Modeling GitHub's createProject Query

Dependency Analysis

After modeling the GraphQL schema and creating GT rules for available queries and mutations, we can perform critical pair analysis. Henshin offers a dependency analysis feature. The results of the analysis show if one rule application may enable another rule, leading to a dependency. In the overall taint analysis picture, dependency analysis is a static analysis technique that is able to identify created nodes/edges in a graph transformation rule (acting as a source that introduces tainted data) that are potentially used within another rule (acting as a sink). This knowledge is later used to create test-cases that use such rules and sources to add sensitive information then check if this information will flow to a dangerous sink.

Rule	Dependencies
createRepository	updateRepository, deleteRepository
createIssue	updateIssue, deleteIssue
createProject	getProject, deleteProject
deleteIssue	deleteRepository

Table 3: Dependencies Result Table

Table 3 below shows an example of the dependencies obtained from dependency analysis. We can see some GT rules that are dependent on other ones for example, the

updateIssue rule is dependent on the *createIssue* rule. This is because an issue must be created in order for it to be updated. This table is used in the next phases of the taint analysis as a part of the testcase specification.

5.2 Implementation

5.2.1 Introspection

According to the approach in Fig. 4 and the design presented in Fig. 5, it is now possible to discuss the implementation details including the specific tools and techniques used. The first step of static analysis is obtaining the GraphQL schema. In order to know what types are available on the API we can start by making an introspection query of the "__schema" field, which is available on the root type of GraphQL. In general, GraphQL introspection queries are prefixed with "__" and a specific introspection query starts with "__schema". The code listing 1 shows an example introspection query used to fetch existing data types including their name and description. The full introspection query used to fetch the full schema of GitHub's API can be found in the appendix in code listing 12.

Listing 1: Simple GraphQL Introspection Query

```
{
  __schema {
    types {
      name
      description
    }
  }
}
```

We have automated this process using a simple python script that sends an introspection query that yeilds the full API documentation. The result of this query is the entire GraphQL schema definition in JSON format as this is the standard output of any GraphQL API response. It is necessary to convert the schema into the *Schema Definition Language* (SDL) as it shows a the schema in a more clear structure presenting types, queries and mutations. This conversion is possible using online tools such as codesandbox [4].

The full schema for an API like GitHub's is large because it contains all possible types and operations that can be executed. Code listing 13 shows the function *execute_introspection_query*

[4] https://codesandbox.io/s/graphql-introspection-sdl-svlx2

function is a Python function that takes three arguments: *endpoint* which is the API endpoint, *introspection_query* which is the introspection query to be executed according to 12, and an authentication token if needed. The function performs a GraphQL introspection query by sending a POST request to the specified endpoint with the provided introspection query and token in the headers.

Listing 2: Utility Python Function that Performs GraphQL Introspection

```
def get_introspection_query(endpoint, introspectio_query, token):
    headers = {"Authorization": f"Bearer {token}"}
    data = introspection_query
    response = requests.post(endpoint, json=data, headers=headers)
    return response.json()
```

5.2.2 Modeling

Now that the schema has been obtained, the *system under test* (SUT) needs to be specified before we proceed to the second step of STA which is modeling. In this thesis our SUT is a simplified version of GitHub's schema. Therefore, we focus on modeling specific features of the API instead of modeling the entire API. Overall we model 9 different queries and mutations that are related to users, repositories, issues and projects. After obtaining the schema and converting it to SDL, it is then possible to omit all the unrelated types and queries. The code listing in the appendix 14 shows the GraphQL schema after applying our simplifications.

The modeling steps in this thesis are performed through Henshin's GUI editor. In this step we need to define a Henshin ecore file. Henshin's ecore files define a meta-model. The structure of Henshin's ecore files consists of a set of classes, each of which defines a type of object that can appear in Henshin models. These classes are represented in the ecore file as "EClass" elements, which define the name, attributes, data types and relationships of each class [5]. The method for modeling GraphQL schemes are the following:

1. A type defined in a GraphQL schema can be modeled as an *EClass* in Henshin. An EClass is a class, with zero or more attributes and zero or more references which directly correspond to a type in a GraphQL schema.

2. An attribute of a type in GraphQL can be represented as an *EAttribute* in Henshin. An EAttribute has a name and a type such as EString, EInt, EFloat ..etc.

[5] https://www.vogella.com/tutorials/EclipseEMF/article.html

3. Sometimes an attribute in GraphQL can be of another type which has been previously defined in the schema. This way different types in GraphQL can reference each other. Such references in GraphQL can be represented as an *EReference*. An EReference in Henshin represents one end of an association between two EClasses.

In Fig. 8 the structure of the created ecore file can be seen. This figure represents the meta-model of a portion of GitHub's GraphQL schema.

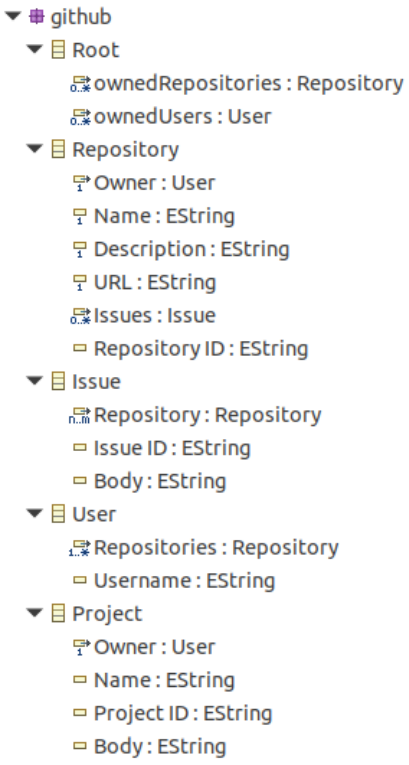

Figure 8: Ecore File Representing GitHub's GraphQL Schema

Now that the meta-modeling is complete, it is now possible to proceed to creating the different graph transformation rules which correspond to the GraphQL queries and mutations. Graph transformation rules can consist of several elements [6]:

1. Nodes are used to represent the different GraphQL types and also correspond to the different EClasses defined in the ecore file.

2. Edges are used to specify a link between two nodes. This is only possible if the meta-model contains a reference between the two corresponding classes.

3. Attributes can be created within nodes and are specified by the name of the attribute and an '=' followed by an expression which is evaluated during the rule application.

[6] https://wiki.eclipse.org/Henshin/Getting_started

4. Actions are used to annotate nodes and edges. A number of actions are supported: «preserve» matches an object and preserves it during the rule application, «create» creates a new object or link between objects, «delete» deletes an existing object or link.

5. Rule parameters are also very important elements which are used to identify which nodes are to be transformed and which attributes must be changed. A parameter has a kind, which can be either in or out. An in parameter is passed into the rule to match and identify specific nodes with specific attributes. An out parameter is passed out of the rule as a return value.

Fig. 9 below, shows how the graph transformation rule *createProject* is represented in Henshin. The *createProject(out projectId, in body, in newName)* rule should create a new project for a user. In this example there are two nodes, one for a user and one for the project as well as an edge from project to user which represents that this user is the owner of the project. The action *«create»* that labels the edge and project node denotes that these components will be created as a result of executing the rule. The input parameters are the new name of the project and it's body while the output parameter is the project ID for the newly created project.

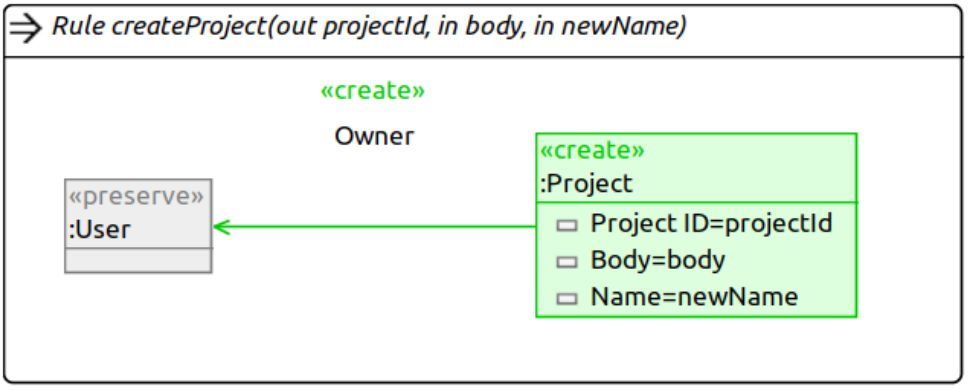

Figure 9: Henshin GT Rule to Create a Project

5.2.3 Dependency Analysis

Now that the modelling step has been completed, it is possible to carry out a dependency analysis in Henshin. The process of executing dependency analysis can be realised through following steps:

1. Open the Henshin diagram file created in the previous step through Eclipse's package explorer menu.
2. Select the rules needed for the analysis through the CDA configuration window. For this thesis we modeled 9 different rules and all of them were selected. We select as well dependencies since conflicts are not relevant for our work.
3. Select which granularity level is desired for the analysis. Coarse granularity was appropriate for the purpose of this thesis.
4. Now, it is possible to view the results of the CDA analysis in the package explorer or by viewing the corresponding directory located in the current workspace.

The results table obtained from Henshin is shown in table 10.

	updateRepo	updateIssue	deleteRepo	getProject	createIssue	createProject	deleteProject	createRepo	deleteIssue
updateRepo									
updateIssue									
deleteRepo									
getProject									
createIssue		Dependency							Dependency
createProject				Dependency			Dependency		
deleteProject									
createRepo	Dependency		Dependency		Dependency				
deleteIssue			Dependency						

Figure 10: Henhsin Dependency Analysis Table

6 Dynamic Taint Analysis

This thesis aims to combine static analysis with dynamic analysis which enables developers to test GraphQL APIs by sending requests and observing the responses in real-time. For the DTA phase of our approach we follow the dynamic testing procedure described by the book *"Software Engineering"* by Ian Sommerville [Som10] which is a primary reference for this chapter. Fig. 11 shows the dynamic testing procedure mentioned. For the implementation, we develop a python program that executes queries against a running GraphQL server to confirm if security vulnerabilities actually exist. In this chapter, the design and implementation of dynamic analysis on GraphQL APIs will be presented, including the tools and techniques used to perform the analysis. By presenting this work, we aim to contribute to the development of more effective testing methodologies for GraphQL APIs.

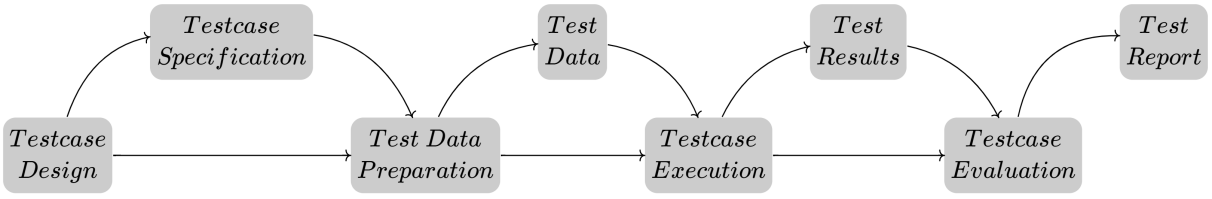

Figure 11: Dynamic Testing Procedure [Som10]

6.1 Design

Testcase Specification

The testcase specification is based on the results of CPA. We specify testcases for two vulnerability categories, broken access control where an unauthorized user is able to execute unauthorized queries and data leaks where an unauthorized user is able to view private information. The testcases are composed of the GraphQL queries/mutations, and the expected output which is the API response to the queries/mutations. To create testcases, we must first analyze GitHub's access control policies [7] to identify which information is considered sensitive and should be inaccessible to unprivileged users. This step is done manually. Below we can see an example of some permissions for a personal account repository:

- Only the creator of a repository is able to update it, if there are no collaborators in the repository.
- Only the creator of a repository is able to delete the repository.

[7] https://docs.github.com/en/get-started/learning-about-github/access-permissions-on-github

Thus, we can see that users can collaborate on repositories but certain actions are only allowed by the creator or the repository. This provides us with enough information to then use the results of dependency analysis, which specify the taint sources and sinks, to build our testcases. Looking at table 3, we see that many dependent rule pairs consist of one rule that creates information (taint source) and another rule that accesses the information created (a sink).

The first category of testcases is access control testcases. These testcases can be constructed using dependent pairs of queries that first creates and then deletes/updates a repository, project or an issue in a repository. For example, we can first execute a mutation such as *createRepository*, which is a taint source where we insert repository information into the API. This information is supposed to be inaccessible to unauthorized users according to GitHub's policies. Then we execute a second mutation, *deleteRepository*, which is executed using an unauthorized user with without any permissions or with insufficient permissions such as read-only permission. The second mutation attempts to delete the private data created in the first mutation. Fig. 12 shows how such an access control testcase can be executed.

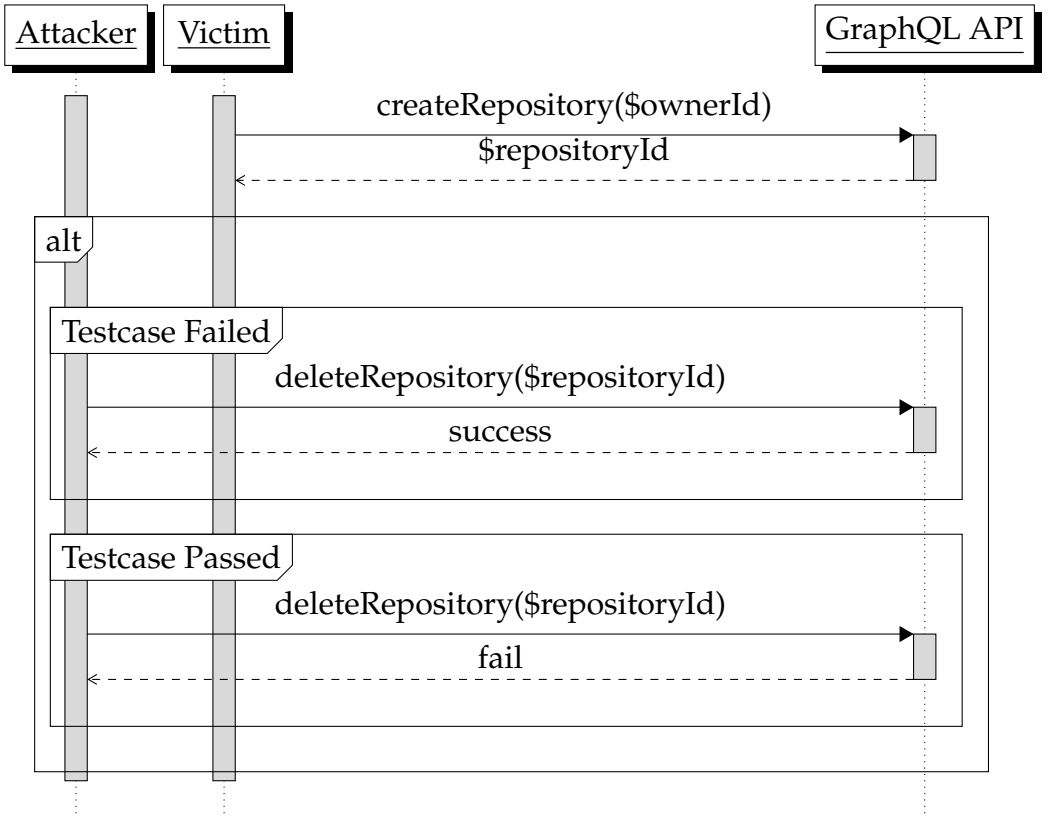

Figure 12: Sequence Diagram Showing an Example Access Control Testcase

The testcases for data leaks are also similar to the testcases for broken access control. The difference is that in the case of data leaks, the sensitive information is accessed but

not modified. The input to the testcase is a GraphQL mutation which is a source of sensitive data and a query that attempts to access the sensitive data. The second query is executed by a user without sufficient permissions. For example fig. 13 shows a mutation *createProject* that can be executed to create a private project and is created by the owner of the project. The second mutation *getProject* allows to view the project and is executed by a user with insufficient permissions. The second mutation should fail as the second user does not have the necessary permissions. If such mutation succeeds then it is considered as a data leak as one user is able to view a private project without having the necessary permissions.

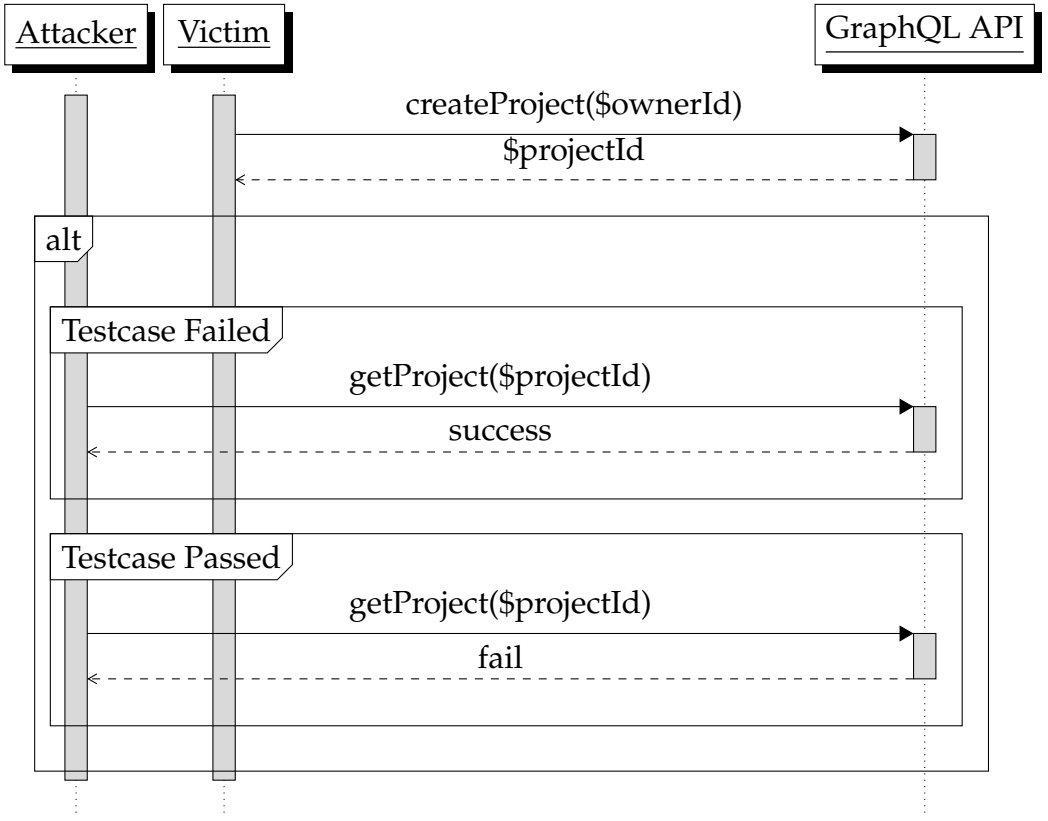

Figure 13: Sequence Diagram Showing an Example Data Leak Testcase

The expected output for both broken access control and data leaks is the same. The first mutation should be executed correctly as it is a legal operation. However, the second operation should fail because the second user does not have the necessary permissions and is not allowed to update or read the sensitive data. This constitutes our expected output which is an error message from the API denying the second user access to the sensitive data.

6.2 Implementation

After the completion of STA and the testcase specification, we can proceed with implementing DTA. In this section, we will outline the implementation of our dynamic analysis process and delve into the details of these methods. Fig. 14 shows the different components of DTA process. The CDA parser and unit tests are python scripts that we created. The technical details regarding the programming of these components are presented in this section. Overall, this STA and DTA testing approach for GraphQL APIs can help ensure that the API is robust, secure, and performs as expected in different scenarios.

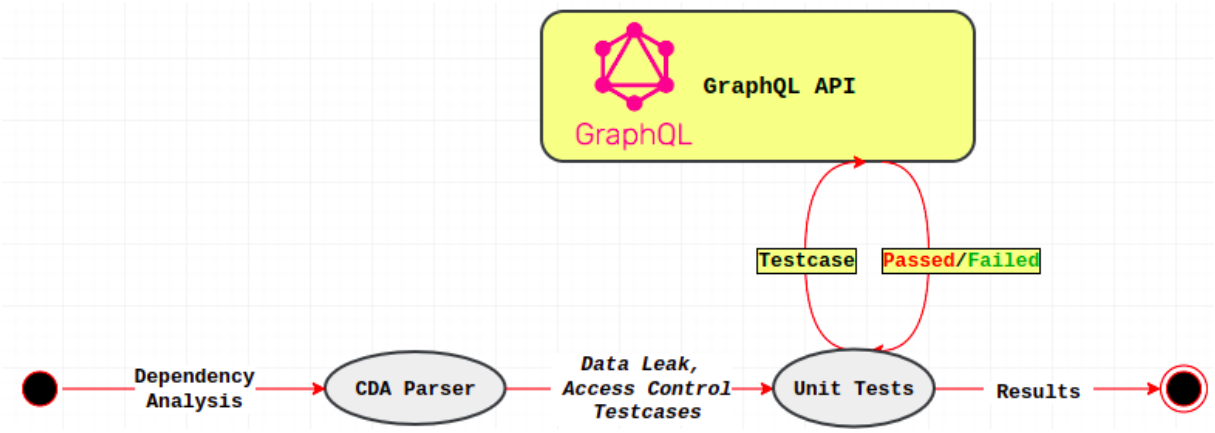

Figure 14: Activity Diagram of our Dynamic Taint Analysis

6.2.1 Test Data Generation

In order to execute the testcases, it is necessary to prepare valid GraphQL queries and mutations with their corresponding input parameters. This step is done manually by analyzing the schema definition and supplying the queries with correct input parameters. These parameters may include identifiers and other data that are relevant to the repositories and users being tested. To generate this test data, the test developer may need to use other GraphQL queries to retrieve the necessary information, such as repository IDs, user IDs, and other data points specific to the application being tested. Code listing 3 shows three GraphQL queries that are able to find the IDs of particular user, repository or issue. After obtaining the necessary parameters to execute the query, it can then be manually entered into the input parameters of the test queries and mutations to ensure that they are ready to be executed.

Listing 3: Queries for Test Data Preparation

```
query findRepo {
  repository(owner: "Username", name: "Repository Name") {
    id
  }
}

query findUserID{
  viewer {
    login
    id
  }
}

query findIssue{
  repository(owner: "Username", name: "Issue Name") {
    issue(number: 1) {
      id
    }
  }
}
```

6.2.2 Testcase Execution

Once the testcases have been specified and the necessary test data have been generated, the testcases can be executed against GitHub's API. This can be achieved using a program that sends the queries and mutations to the API similar to fig. 13. In the code listing 4, an example python function that executes a data leak testcase can be seen. Specifically, it tests the functionality of creating and retrieving a project in a repository. The code defines a function called *"test_data_leak"*, which takes an instance of the *GraphqlClient* class, which is used to send GraphQL queries to GitHub. The function defines two GraphQL queries: *createProject* and *getProject*. *createProject* is a mutation that creates a new project in a repository, while *getProject* is a query that retrieves the details of a project given its ID. The function also sets the headers variable to a dictionary containing an authorization token. This token is required to access the GitHub API with a user who has admin permissions in the respective repository. The function executes the createProject mutation. The function extracts the ID of the newly created project from the *createProjectResult* variable and replaces the *$ProjectId* variable in the *getProject* query with the ID of the newly created project. The function then executes the

getProject query from another user with a different access token and different access control permission. The function then stores the result in the *getProjectResult* variable.

Listing 4: Dynamic Testing Code Snippet

```
def test_data_leak(client, createProjectQuery, getProjectQuery):
    headers = {"Authorization": "Bearer $TOKEN"} # Victim's Token
    createProjectResult = client.execute(createProject, headers=headers)
    print(createProjectResult)

    projectId = createProjectResult['data']['createProjectV2']
              ['projectV2']['id']
    getProject = getProject.replace('$ProjectId', '"' + projectId + '"')

    headers = {"Authorization": "Bearer $TOKEN"} # Attacker's Token
    getProjectResult = client.execute(getProject, headers=headers)
    print(getProjectResult)
```

To ensure that the testcases are executed correctly, the program must send the queries and mutations in the correct order and from different users with different access control privileges. For example, in the case of the broken access control testcases, the first mutation should be executed by an admin user, while the second mutation should be executed by a user with lower access control privileges, such as a read-only user. The program used to execute the testcases should also contain the access tokens of the two different GitHub users, as well as the test data, including the queries and mutations and their corresponding input parameters. These input parameters are typically retrieved during the test data generation step, as discussed earlier.

6.2.3 Testcase Evaluation

In this section we use Pytest [pyt] to evaluate the executed testcases. As seen in the previous section, testcases can be created as functions and then executed. The python program can be adjusted slightly by including assertions to allow the Pytest unit testing framework to evaluate if the testcases passed or failed. Listing 5 show how such assertions look like in code. The first assertion checks if the createProjectV2 mutation has been executed properly by checking if a projectId has been returned by the API. The second assertion checks if the output of the getProject is an error message by GraphQL not being able to fetch the project. If one of the assertions fail then our testcase is failed and should be further investigated.

Listing 5: Assertions for Testcase Evaluation

```
    assert projectId in createProjectResult['data']['createProjectV2']
          ['projectV2']['id']
    assert "Could not resolve to a node with the global id of" in
          getProjectResult["errors"][0]["message"]
```

All testcases can then be programmed in one file and prefixed with "test". In this way pytest can parse the file and execute the testcases. Listing 6 shows a bash command that allows pytest to evaluate the testcases in the file "graphql_unti_test.py" where it determines that 3 testcases were executed and all passed.

Listing 6: Testcase Evaluation using Pytest

```
$ pytest --no-header -v graphql_unit_test.py
============================================================================
test session starts
============================================================================
collected 3 items
graphql_unit_test.py::test_delete_update_issue_conflict PASSED [33%]
graphql_unit_test.py::test_data_leak PASSED [66%]
graphql_unit_test.py::test_update_issue_pararmeter_fuzzing PASSED [100%]

============================================================================
3 passed in 8.42s
============================================================================
```

7 Evaluation

In this thesis, we propose a new taint analysis approach for testing GraphQL APIs using combined SAST and DAST techniques. To evaluate our approach, we conducted an experiment where we compared our approach with an existing tool called Schemathesis [HDD21]. Schemathesis is an open-source tool written in Python that can automatically generate and run fuzzing tests for GraphQL APIs. It uses a schema-based testing approach, where it generates random queries and mutations derived from the GraphQL schema and then sends them to the API for testing. Schemathesis is similar to our approach of testing GraphQL APIs because it uses combined SAST and DAST techniques. Schemathesis performs fuzzing which uncovers a verity of program bugs including security vulnerabilities. While not necessarily designed to find security vulnerabilities, Schemathesis is a good candidate to evaluate our approach because it is a mature tool with a similar architecture of combining SAST and DAST. This allows us to evaluate the strengths and weaknesses of our approach.

Schemathesis is a tool for automated blackbox or whitebox testing of web APIs, deriving structure and semantic aware fuzzers from OpenAPI or GraphQL schemas [HDD21]. It is based on a python library called Hypothesis [MHDC19] which is a testing toolkit for property-based testing, swarm testing, and coverage-guided fuzzing. Schemathesis is able to generate test inputs including utilizing techniques such as random generation and feedback-guided structured mutation. Using the mentioned techniques Schemathesis is ultimately able to offer multiple features that distinguishes it from other tools such as:

- **OpenAPI & GraphQL:** Tests a wide range of APIs with ease, regardless of the specification used.
- **Positive & Negative Tests:** Ensures that the API handles valid and invalid inputs, incl. unexpected ones.
- **Stateful Testing:** Automatically generates sequences of API requests where subsequent requests built on previous ones for testing complex and interdependent scenarios.
- **Session Replay:** Quickly stores and replays test sessions to easily investigate and resolve issues.
- **Targeted Testing:** Allows to guide data generation towards specific metrics like response time or size. This allows to uncover performance or resource usage issues and optimize API behavior under different conditions.

Schemathesis can be used via a command-line interface or a Python library. It is able to fuzz services written in any language via HTTP. For these reasons Schemathesis is a thriving open-source project with thousands of downloads every week. It has already been widely adopted [HDD21].

7.1 Experiment

In this section, we aim to conduct an experiment to compare our approach and Schemathesis. To start the experiment, we execute our taint analysis approach on GitHub's GraphQL API. We execute the following steps:

1. We perform introspection using a python script to obtain GitHub's schema. The script can be found in listing 13 in the appendix.
2. We simplify the schema to focus only on repositories, issues and projects. The simplified schema can be found in the appendix 14.
3. We create a Henshin model by creating a Henshin ecore file. The steps for modeling are listed in section 5.2.2.
4. We create graph transformation rules modeling the queries and mutations in the schema. They are created as Henshin diagram file.
5. We execute dependency analysis. The steps are listed in section 5.2.3.
6. We use our dependency analysis parser script (can be found in the appendix 11) to get a summary of the dependencies. We select dependent pairs that represent sources and sinks.
7. We review GitHub's access control policies [git] to design access control testcases and data leak testcases.
8. We create two GitHub accounts and obtain access tokens for both users. We prepare the test data using queries as in listing 3.
9. We execute the dependent queries as unit tests using our python in listing 10. The first query is performed with an admin user's token for a repository or a project. The second query is performed using a user's token with low permissions.
10. We use pytest to execute the python script and to evaluate our testcases. This can be done similar to listing 6.

Then we execute a standard Schemathesis parameter fuzzing test. For this we need an API endpoint and an authorization token. We execute the following command that specifies a user token and an API endpoint to start a fuzzing test:

Listing 7: Schemathesis Command to Execute a Fuzzing Test on GitHub's API

```
st run -H "Authorization: Bearer TOKEN" https://api.github.com/graphql
```

Schemathesis starts by executing an introspection query to obtain the GraphQL schema, then performs static analysis on the schema to start generating fuzzing testcases. Schemathesis performs parameter fuzzing of all GraphQL queries and mutations. The default configuration of Schemathesis sets 100 fuzzing testcases per query/mutation. The entire test for GitHub's GraphQL API lasted around 3 hours.

7.2 Results

After executing our taint analysis approach and Schemathesis on GitHub's API, we can compare the findings, testcases and general flow of both tools. The comparison results are presented in table 4. Schemathesis performs automatic fuzzing testcases based on the GraphQL schema. The fuzzing approach of Schemathesis requires generating and sending many testcases. The testcases attempt to test if the API handles valid and invalid input without crashing or having an error. If a testcase fails it could be failing because of an error in input validation, authorization or configuration. Listing 8, shows a Schemathesis testcase. The testcase attempts to insert invalid inputs such as negative numbers for the *itemId* input parameter.

Listing 8: Typical Schemathesis Testcase

```
mutation deleteProjectV2Item(input: {itemId: -1435395303, projectId: 17656,
   clientMutationId: \"E\\u001AsE\u00ea\\u008CT?\u00b4\u00ec\"}) {
   deletedItemId
   clientMutationId
   }
}
```

On the other hand, our approach, involved a combination of manual and automated steps. Using the simplified GitHub schema we modeled only 9 GT rules for which there were 8 dependent rule pairs. Listing 9, shows an example testcase where two dependent rules pairs are executed by different users. The testcase uses a privileged user to create a project and obtain a project ID and then executes a delete project mutation by an unprivileged user. This testcase is able to identify if the access control policies are enforced.

Listing 9: Typical Testcase of Our Approach

```
# Executed by a privileged user
mutation createProject(input: {name: $name, body: $body, repositoryId:
   $repositoryId}) {
```

```
    project {
      id
      name
      body
    }
  }
}

# Executed by an unprivileged user
mutation deleteProject(input: {projectId: $projectId}) {
    clientMutationId
  }
}
```

Our approach is able to test for vulnerabilities that were are not tested for by Schemathesis. One of the biggest differences are that our testcases are performed by two different users with different privileges. Schemathesis only accepts one user token and does not check for vulnerabilities originating by the interaction between two different types of users. Our approach is significantly less time-consuming as we only executed 8 testcases on the specific features we modeled. As for Schemathesis, by default it executed 900 testcases only for the specific features we considered for our approach. Overall, our experiment has demonstrated the strengths and weaknesses of both approaches, and highlights the importance of using multiple testing methods for comprehensive security testing. Table 4 shows a comparison of both approaches in terms of level of automation, vulnerabilities discovered and testing technique used.

Criteria		**Approach**	**Schemathesis**
Level of Automation	Introspection	Automatic	Automatic
	Testcase Specification	Manual	Manual
	Test Data Generation	Manual	Automatic
	Testcase Execution	Automatic	Automatic
	Testcase Evaluation	Automatic	Automatic
Vulnerability Categories		Broken Access Control, Data Leaks	Denial of Service, Input Validation
Technique		Taint Analysis	Parameter Fuzzing

Table 4: Comparison Table of Our Approach and Schemathesis

8 Conclusion

The use of GraphQL revolutionized Web APIs by offering an efficient and easy to alternative. Unfortunately, the adaptation of this technology also comes with security risks associated with other web APIs as well as specific risks with GraphQL. The existing approaches to testing GraphQL APIs are limited and do not consider the underlying graph structure of GraphQL APIs. Thus, this research is focused on the development and implementation of a novel approach that combines both static and dynamic taint analysis to test GraphQL APIs for security vulnerabilities. The approach takes into account the underlying graph structure of the GraphQL schema by formalizing it using typed graphs and graph transformations rules. Then we utilize dynamic taint analysis to verify if the vulnerabilities exist. One of the primary advantages of this approach is that it can identify data leaks and broken access control vulnerabilities that are not easily detected by other GraphQL tools.

As future work, it is valuable to explore ways to automate the manual components of our approach, which currently require some degree of human intervention. One possible avenue for this would be to further develop the pipeline of programs presented to automates every single step discussed in the design and implementation of the approach. The most challenging step, is the automation of the modeling stage due to the use of Henshin, however a text-to-model approach from the GraphQL schema language to a typed graph and s set of GT rules seems to be a promising direction to explore. This takes the state of the art one step closer to combining the entire approach into one tool which would be scalable, efficient and practical to use. Another area for future work would be to investigate ways to integrate our approach with existing security testing frameworks, such as OWASP ZAP and Burp Suite, to provide a more streamlined testing experience for developers and security professionals.

In conclusion, this work proposes a new approach for testing GraphQL APIs that addresses the limitations of existing methods. This approach can help developers test their APIs for broken access control and data leaks. Moreover, the proposed approach can be extended to other APIs such as REST based APIs with some adjustment. This research provides a contribution to the field of GraphQL security. Further research could explore how this approach can be fully automated and further extended to test other types of APIs. Overall, this work promotes better API testing practices and has the potential to positively impact the security of GraphQL APIs.

9 Appendix

9.1 Source Code

GraphQL Dynamic Taint Analysis Unit Tests

Listing 10: Dynamic Testing Code

```
import os
import pytest
from python_graphql_client import GraphqlClient

# Define the test cases
def test_delete_update_issue_conflict(client =
    GraphqlClient('https://api.github.com/graphql')):
    # Construct the GraphQL query to get the user's name, email, and
        repositories
    findIssue = """
    query {
        repository(owner: $username, name: $repositoryName) {
            issue(number: 5) {
            id
            }
        }
    }
    """

    createIssue = """
    mutation CreateIssue {
    createIssue(input: {repositoryId: $repositoryId, title: "Unit
        Testing!!!", body: "Unit Testing!!!"}) {
        issue {
        id
        }
    }
    }
    """

    deleteIssue = """
    mutation {
        deleteIssue(input: {issueId: $IssueId}) {
```

```
        clientMutationId
      }
  }
  """

  updateIssue = """
  mutation {
  updateIssue(input: {id: $IssueId, body: "My updated issue body"}){
      issue {
        id
        }
  }
  }
  """

  # Set the authorization header using a GitHub personal access token
  headers = {"Authorization": "Bearer $TOKEN"}

  # Send the query to the GitHub API using the client fixture
  createIssueResult = client.execute(createIssue, headers=headers)
  print(createIssueResult)
  issueId = createIssueResult['data']['createIssue']['issue']['id']

  deleteIssue = deleteIssue.replace('$IssueId', '"' + issueId + '"')
  deleteIssueResult = client.execute(deleteIssue, headers=headers)
  print(deleteIssueResult)

  updateIssue = updateIssue.replace('$IssueId', '"' + issueId + '"')
  updateIssueResult = client.execute(updateIssue, headers=headers)
  print(updateIssueResult)

  # Verify that the response contains the expected data
  assert issueId in
      createIssueResult['data']['createIssue']['issue']['id']
  assert deleteIssueResult["data"]["deleteIssue"]["clientMutationId"] ==
      None
  assert "Could not resolve to a node with the global id of" in
      updateIssueResult["errors"][0]["message"]

```

```
def test_data_leak(client =
    GraphqlClient('https://api.github.com/graphql')):
   createProject = """
   mutation {
   createProjectV2(input: {
      title:"Testing for data leaks",
      ownerId: $ownerId
   }) {
      projectV2 {
      id
      title
      }
   }
   }
   """

   getProject = """
   query{
   node(id: $ProjectId) {
      ... on ProjectV2 {
      fields(first: 20) {
         nodes {
         ... on ProjectV2Field {
            id
            name
         }
         ... on ProjectV2IterationField {
            id
            name
            configuration {
            iterations {
               startDate
               id
            }
            }
         }
         ... on ProjectV2SingleSelectField {
            id
            name
            options {
```

```
            id
            name
            }
         }
         }
      }
      }
      }
   }
   """
   headers = {"Authorization": "Bearer $VictimToken"} # Vicitim Token
   createProjectResult = client.execute(createProject, headers=headers)
   print(createProjectResult)

   projectId =
      createProjectResult['data']['createProjectV2']['projectV2']['id']
   getProject = getProject.replace('$ProjectId', '"' + projectId + '"')

   headers = {"Authorization": "Bearer $Attacker_Token"} # The Attacker
      Token
   getProjectResult = client.execute(getProject, headers=headers)
   print(getProjectResult)

   assert projectId in
      createProjectResult['data']['createProjectV2']['projectV2']['id']
   assert "Could not resolve to a node with the global id of" in
      getProjectResult["errors"][0]["message"]


def test_update_issue_pararmeter_fuzzing(client =
    GraphqlClient('https://api.github.com/graphql')):
   issueIds = ["I_kwDsalkjda", "I_kwDOI8ss21f4f", "I_osospbyy21msa9"]
   updateIssue = """
   mutation {
   updateIssue(input: {id: $IssueId, body: "My updated issue body"}){
      issue {
         id
         }
   }
   }
```

```
    """
    headers = {"Authorization": "Bearer $TOKEN"}

    for issueId in issueIds:
        updateIssueTestcase = updateIssue.replace('$IssueId', '"' + issueId
            + '"')
        print(updateIssueTestcase)
        updateIssueResult = client.execute(updateIssueTestcase,
            headers=headers)
        print(updateIssueResult)
        assert "Could not resolve to a node with the global id of" in
            updateIssueResult["errors"][0]["message"]



test_update_issue_pararmeter_fuzzing()
```

Dependency Analysis Parser

Listing 11: Python Utility to Parse Dependency Analysis Results

```
from bs4 import BeautifulSoup
import pprint
import argparse

# Define the argument parser
parser = argparse.ArgumentParser(description='Provide a Henshin CPA HTML
    File')
parser.add_argument('-f', '--file', type=str, required=True, help='path to
    your file')

# Parse the command-line arguments
args = parser.parse_args()

# Access the argument value
if args.file:
    print("Parsing the File at Location: ", args.file, "\n")
else:
    print("No argument provided.")
    exit
```

```
18
19 # Open the local HTML file
20 with open(args.file, 'r') as file:
21     html = file.read()
22
23 # Parse the HTML content using BeautifulSoup
24 soup = BeautifulSoup(html, 'html.parser')
25
26 # Find the table element in the HTML
27 table = soup.find('table')
28
29 data = []
30 headers = [header.text.strip() for header in table.find_all('th')[1:]]
31 is_first_row = True
32 for row in table.find_all('tr')[1:]:
33     cells = [cell.text.strip() for cell in row.find_all('th')[1:]]
34     if cells:
35         row_name = row.find_all('th')[0].text.strip()
36         for col_name, cell in zip(headers, cells):
37             if cell:
38                 data.append([row_name, col_name, cell])
39
40 # Print the list of tuples to see the extracted data
41
42 for i in data:
43     if i[2][-2:] == "CR":
44         i[2] = "Conflict"
45     elif i[2][-2:] == "DR":
46         i[2] = "Dependency"
47
48 pprint.pprint(data)
49
50 print("\nTestcases:")
51 counter = 1
52 for i in data:
53     if i[2] == "Conflict":
54         print("Conflicting Rules Testcase #", counter, " = (", i[0], ", ",
              i[1], ")", "(Error)")
55         counter += 1
56     elif i[2] == "Dependency":
```

```
57        print("Dependent Rules Testcase # ", counter)
58        print("Dependent Rules: (", i[0], ",", i[1], ")")
59        print("Victim =>", i[0], "\nHacker => ", i[1], "\nExpected Output =>
              (Error)")
60        counter += 1
61
62  print("\nTestcase are Interpreted as: ((Rule Pair to be
      Executed),(Expected Output))")
```

Introspection

Listing 12: Obtaining a GraphQL Schema using Native GraphQL Introspection

```
1   query IntrospectionQuery {
2     __schema {
3       queryType {
4         name
5       }
6       mutationType {
7         name
8       }
9       subscriptionType {
10        name
11      }
12      types {
13        ...FullType
14      }
15      directives {
16        name
17        description
18        locations
19        args {
20          ...InputValue
21        }
22      }
23    }
24  }
25  fragment FullType on __Type {
26    kind
27    name
```

```
  description
  fields(includeDeprecated: true) {
    name
    description
    args {
      ...InputValue
    }
    type {
      ...TypeRef
    }
    isDeprecated
    deprecationReason
  }
  inputFields {
    ...InputValue
  }
  interfaces {
    ...TypeRef
  }
  enumValues(includeDeprecated: true) {
    name
    description
    isDeprecated
    deprecationReason
  }
  possibleTypes {
    ...TypeRef
  }
}
fragment InputValue on __InputValue {
  name
  description
  type {
    ...TypeRef
  }
  defaultValue
}
fragment TypeRef on __Type {
  kind
  name
```

```
  ofType {
    kind
    name
    ofType {
      kind
      name
      ofType {
        kind
        name
        ofType {
          kind
          name
          ofType {
            kind
            name
            ofType {
              kind
              name
              ofType {
                kind
                name
                ofType {
                  kind
                  name
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Introspection Script

Listing 13: Script that Perfroms a Full Introspection Query

```
def get_introspection_query(endpoint, token):
    headers = {"Authorization": f"Bearer {token}"}
    data = {"query": """
        query IntrospectionQuery {
          __schema {
            queryType { name }
```

```
    mutationType { name }
    subscriptionType { name }
    types {
      ...FullType
    }
    directives {
      name
      description
      locations
      args {
        ...InputValue
      }
    }
  }
}

fragment FullType on __Type {
  kind
  name
  description
  fields(includeDeprecated: true) {
    name
    description
    args {
      ...InputValue
    }
    type {
      ...TypeRef
    }
    isDeprecated
    deprecationReason
  }
  inputFields {
    ...InputValue
  }
  interfaces {
    ...TypeRef
  }
  enumValues(includeDeprecated: true) {
    name
```

```
        description
        isDeprecated
        deprecationReason
      }
      possibleTypes {
        ...TypeRef
      }
    }

    fragment InputValue on __InputValue {
      name
      description
      type { ...TypeRef }
      defaultValue
    }

    fragment TypeRef on __Type {
      kind
      name
      ofType {
        kind
        name
        ofType {
          kind
          name
          ofType {
            kind
            name
            ofType {
              kind
              name
            }
          }
        }
      }
    }
  """}
```

9.2 Modeling Artifacts

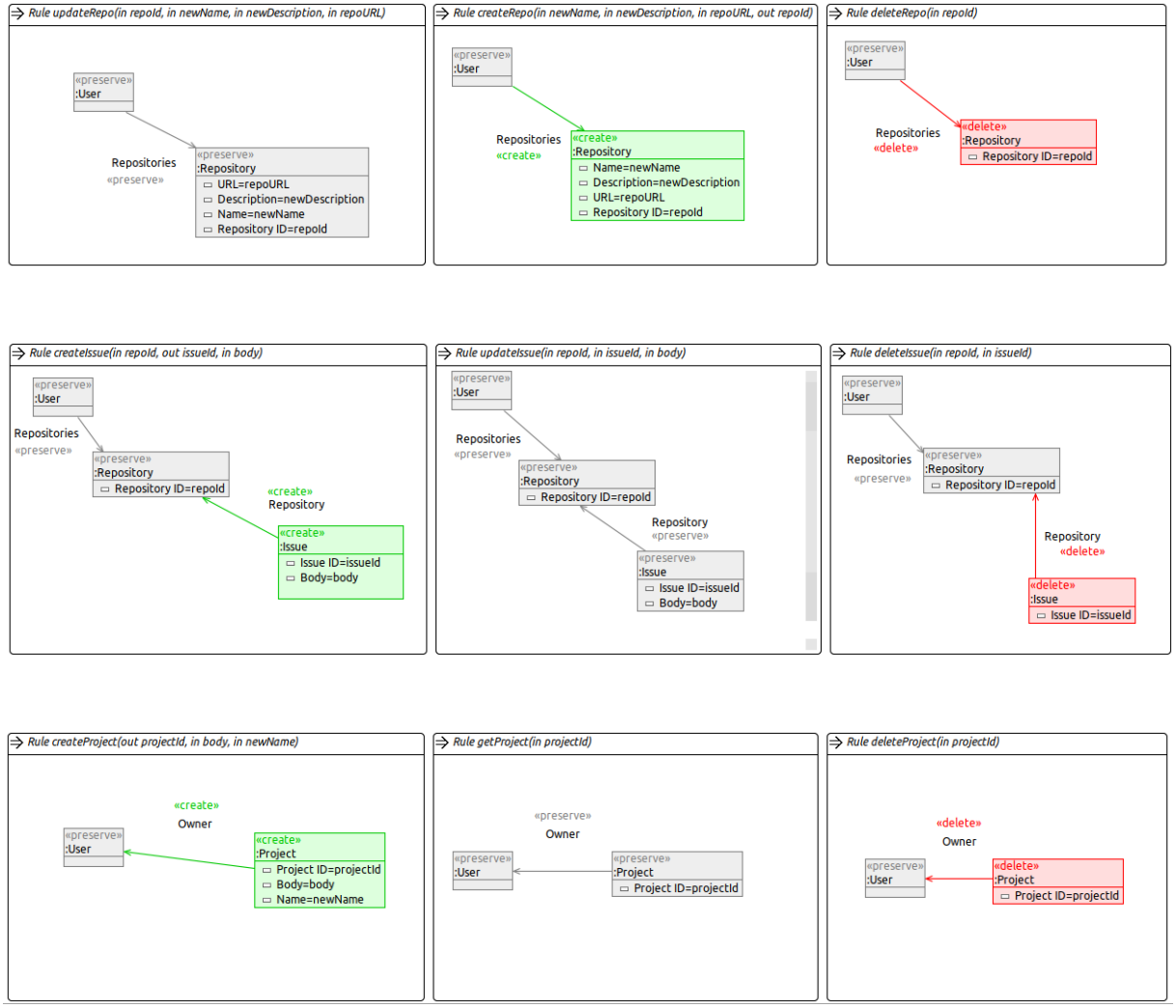

Figure 15: Graph Transformation Rules Modeling GitHub'sHub's Queries/Mutations

Listing 14: Simplified GraphQL Schema

```
type Query {
  User {
    email: String!
    repositories: [Repository]
  }

  Repository {
    description: String
    name: String!
    isPrivate: Boolean!
    owner: RepositoryOwner!
    url: URI!
```

```
  }

  Issue {
    body: String!
    repository: Repository!
    id: ID!
  }

  Project{
    name: String!
    owner: ProjectOwner!
    body: String!
    id: ID!
  }

}

interface RepositoryOwner {
  login: String!
  repositories: [Repository]!
}

interface ProjectOwner {
  id: ID!
  projects: [Project]!
  projectsResourcePath: URI!
  projectsUrl: URI!
  viewerCanCreateProjects: Boolean!
}

type Mutation {
  createRepository(input: CreateRepositoryInput!): CreateRepositoryPayload
  updateRepository(input: UpdateRepositoryInput!): UpdateRepositoryPayload
  createIssue(input: CreateIssueInput!): CreateIssuePayload
  deleteIssue(input: DeleteIssueInput!): DeleteIssuePayload
  updateIssue(input: UpdateIssueInput!): UpdateIssuePayload
  createProject(input: CreateProjectInput!): CreateProjectPayload
  deleteProject(input: DeleteProjectInput!): DeleteProjectPayload
}
```

References

[ABJ+10] ARENDT, Thorsten ; BIERMANN, Enrico ; JURACK, Stefan ; KRAUSE, Christian ; TAENTZER, Gabriele: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: PETRIU, Dorina C. (Hrsg.) ; ROUQUETTE, Nicolas (Hrsg.) ; HAUGEN, Øystein (Hrsg.): *Model Driven Engineering Languages and Systems*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. – ISBN 978–3–642–16145–2, S. 121–135

[ADS19] ALASHJAEE, Abdullah M. ; DURAIBI, Salahaldeen ; SONG, Jia: Dynamic Taint Analysis Tools: A Review. In: *International Journal of Computer Science and Security (IJCSS)* 13 (2019), December, Nr. 6, 231 - 243. http://www.cscjournals.org/library/manuscriptinfo.php?mc=IJCSS-1518. – ISSN 1985–1553

[agg] *AGG Documentation*. https://www.user.tu-berlin.de/o.runge/AGG/WWW/agg-docu.html, . – Accessed: 02.04.2023

[ARF+14] ARZT, Steven ; RASTHOFER, Siegfried ; FRITZ, Christian ; BODDEN, Eric ; BARTEL, Alexandre ; KLEIN, Jacques ; LE TRAON, Yves ; OCTEAU, Damien ; MCDANIEL, Patrick: FlowDroid. In: *ACM SIGPLAN Notices* 49 (2014), 06, S. 259–269. http://dx.doi.org/10.1145/2666356.2594299. – DOI 10.1145/2666356.2594299

[BTZTC+20] BOTTO-TOBAR, Miguel ; ZAMBRANO, Marcelo ; TORRES-CARRION, Pablo ; MONTES L., Sergio ; PIZARRO-VASQUEZ, Guillermo ; DURAKOVIC, Benjamin: *Applied Technologies First International Conference, ICAT 2019, Quito, Ecuador, December 3–5, 2019, Proceedings, Part I: First International Conference, ICAT 2019, Quito, Ecuador, December 3–5, 2019, Proceedings, Part I*. 2020. http://dx.doi.org/10.1007/978-3-030-42517-3. http://dx.doi.org/10.1007/978-3-030-42517-3. – ISBN 978–3–030–42516–6

[BV20] BRITO, Gleison ; VALENTE, Marco T.: *REST vs GraphQL: A Controlled Experiment*. http://dx.doi.org/10.48550/ARXIV.2003.04761. Version: 2020

[cod] *CodeQL Tool Homepage*. https://codeql.github.com/, . – Accessed: 20.03.2023

[das] *Vulnerability Scanning Tools*. https://owasp.org/www-community/Vulnerability_Scanning_Tools, . – Accessed: 02.04.2023

[EGC+10] ENCK, William ; GILBERT, Peter ; CHUN, Byung-Gon ; COX, Landon ; JUNG, Jaeyeon ; MCDANIEL, Patrick ; SHETH, Anmol: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones, 2010, S. 393–407

[emf] *Eclipse Modeling Framework Homepage*. https://www.eclipse.org/modeling/emf/, . – Accessed: 22.03.2023

[git] *Access permissions on GitHub*. https://docs.github.com/en/get-started/learning-about-github/access-permissions-on-github, . – Accessed: 02.04.2023

[gqla] *GraphQL Playground Homepage*. https://www.apollographql.com/docs/apollo-server/v2/testing/graphql-playground/, . – Accessed: 20.03.2023

[gqlb] *GraphQL Schema Linter Homepage*. https://github.com/cjoudrey/graphql-schema-linter, . – Accessed: 20.03.2023

[HDD21] HATFIELD-DODDS, Zac ; DYGALO, Dmitry: *Deriving Semantics-Aware Fuzzers from Web API Schemas*. http://dx.doi.org/10.48550/ARXIV.2112.10328. Version: 2021

[HP18] HARTIG, Olaf ; PÉREZ, Jorge: Semantics and Complexity of GraphQL. In: *Proceedings of the 2018 World Wide Web Conference*. Republic and Canton of Geneva, CHE : International World Wide Web Conferences Steering Committee, 2018 (WWW '18). – ISBN 9781450356398, 1155–1164

[HT] HECKEL, Reiko ; TAENTZER, Gabriele: *Graph Transformation Concepts*. Springer International Publishing. http://dx.doi.org/10.1007/978-3-030-43916-3_2. http://dx.doi.org/10.1007/978-3-030-43916-3_2. – ISBN 978–3–030–43916–3

[jes] *Jest Homepage*. https://jestjs.io/, . – Accessed: 20.03.2023

[Li20] LI, Jinfeng: Vulnerabilities Mapping based on OWASP-SANS: a Survey for Static Application Security Testing (SAST). In: *CoRR* abs/2004.03216 (2020). https://arxiv.org/abs/2004.03216

[MHDC19] MACIVER, David ; HATFIELD-DODDS, Zac ; CONTRIBUTORS, Many: Hypothesis: A new approach to property-based testing. In: *Journal of Open Source Software* 4 (2019), 11, S. 1891. http://dx.doi.org/10.21105/joss.01891. – DOI 10.21105/joss.01891

[NS05] NEWSOME, James ; SONG, Dawn: Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software., 2005

[owaa] *OWASP GraphQL Cheatsheet*. https://cheatsheetseries.owasp.org/cheatsheets/GraphQL_Cheat_Sheet.html, . – Accessed: 16.03.2023

[owab] *OWASP Web Security Testing Guide 4.2*. https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/12-API_Testing/01-Testing_GraphQL, . – Accessed: 16.03.2023

[per] *Perl Security*. https://perldoc.perl.org/perlsec, . – Accessed: 02.04.2023

[pos] *Postman Homepage*. https://www.postman.com/, . – Accessed: 20.03.2023

[pyt] *Pytest Homepage*. https://docs.pytest.org/en/7.2.x/, . – Accessed: 02.04.2023

[SAB10] SCHWARTZ, Edward J. ; AVGERINOS, Thanassis ; BRUMLEY, David: All You Ever Wanted to Know about Dynamic Taint Analysis and Dynamic Symbolic Execution (but Might Have Been Afraid to Ask). In: *2010 IEEE Symposium on Security and Privacy*, 2010, S. 317–331

[sas] *Source Code Analysis Tools*. https://owasp.org/www-community/Source_Code_Analysis_Tools, . – Accessed: 02.04.2023

[sch] *Schemathesis Tool Homepage*. https://github.com/cjoudrey/graphql-schema-linter, . – Accessed: 20.03.2023

[Sha12] SHARMA, Asankhaya: A Critical Review of Dynamic Taint Analysis and Forward Symbolic Execution. (2012), 01

[snya] *Improving GraphQL security with static analysis and Snyk Code*. https://snyk.io/blog/graphql-security-static-analysis-snyk-code/, . – Accessed: 02.04.2023

[snyb] *Snyk Tool Homepage*. https://snyk.io/product/snyk-code/, . – Accessed: 20.03.2023

[Som10] SOMMERVILLE, Ian: *Software Engineering*. 9. Harlow, England : Addison-Wesley, 2010. – ISBN 978–0–13–703515–1

[son] *SonarQube Tool Homepage*. https://www.sonarsource.com/products/sonarqube/, . – Accessed: 20.03.2023

[Son21] SONI, Monika: A Scientific Review of DAST (Dynamic Application Security Testing) and Its Specification. In: *International Journal of Engineering Research* 1 (2021), 06. http://www.ijeres.org/index.php/journal/article/view/1/1

[STFW01] SHANKAR, Umesh ; TALWAR, Kunal ; FOSTER, Jeffrey ; WAGNER, David: Detecting Format String Vulnerabilities with Type Qualifiers. In: *USENIX Security* 10 (2001), 06

[Yaz20] YAZDIPOUR, Shahriar: *Github Data Exposure and Accessing Blocked Data using the GraphQL Security Design Flaw*. http://dx.doi.org/10.48550/ARXIV.2005.13448. Version: 2020

[YY17] YAN, Lok K. ; YIN, Heng: SoK : On the Soundness and Precision of Dynamic Taint Analysis, 2017

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

_____

(Ort, Datum) (Unterschrift)