University of Passau

Bachelor Thesis

DoS Detection in NodeRED

Author: Per Fuchs

Supervisor: Juan David Parra

November 22, 2015
Abstract

This thesis demonstrates the use of a Code Property Graph to statically detect NodeRED flows that perform a SYN flood, Slowloris or Slow Post attack.

A Code Property Graph is comprised of the abstract syntax tree, control flow graph, program dependence graph and the call graph of a JavaScript program. The Code Property Graph was to detect security breaches in the Linux kernel. This original implementation did not include the call graph. The call graph was integrated with the other graphs during this thesis.

The Code Property Graph of a NodeRED program is traversed to locate invariant code structures of an attack to determine if the flow performs a denial of service attack. Three common invariant code structures of denial of service attacks were identified during this thesis. All considered attacks need to instantiate a TCP socket, write to it and trigger this write periodically. If these three code structures can be found, they are connected in so-called Attack Complexes and the flow is marked to be suspicious. This detection approach is applied to identify SYN flood, Slowloris and Slow Post attacks without a lot of changes; in other words, it is general enough to find all three attacks. Therefore the author is optimistic that it can be easily extended to detect other kinds of denial of service attacks. The Attack Complex approach identifies and locates the most important parts of code in an attack. Hence many traversals to refine the approach will start at these parts. This makes Attack Complexes a great starting point for more precise detection.

Furthermore this thesis contributes a reusable and well-engineered module to produce a Code Property Graph for arbitrary JavaScript code, a program dependence analysis based on two well-respected algorithms and a data flow analysis capable of dealing with closures used as event handler.

Special thanks to my advisor Juan David Parra who supported me with important ideas and patient feedback.
Contents

1 Introduction ................................. 7
  1.1 Related Work ........................... 10
      1.1.1 Static Denial of Service Detection in JavaScript .................. 10
      1.1.2 Denial of Service Detection ................................. 10
      1.1.3 The Original Code Property Graph ............................ 11
      1.1.4 Static Analysis of JavaScript .............................. 11
  1.2 NodeRED ................................. 12
      1.2.1 NodeRED Node Types ................................. 12
      1.2.2 Custom NodeRED Nodes Types ............................... 13
      1.2.3 NodeRED Functions and Events ............................. 13
  1.3 Used Software ............................ 13
  1.4 Software Design ........................... 14
      1.4.1 JavaScriptToBlueprints .............................. 14
      1.4.2 NodeRedToBlueprints ............................... 15

2 Code Property Graph ....................... 17
  2.1 Gremlin Graph Format .................... 18
  2.2 Abstract Syntax Tree ........................ 18
      2.2.1 Implementation ............................... 18
  2.3 Control Flow Graph ........................ 21
      2.3.1 Implementation ............................... 22
  2.4 Program Dependence Graph ................. 22
  2.5 Control Dependence ........................ 22
      2.5.1 Implementation ............................... 23
  2.6 Data Dependence .......................... 24
      2.6.1 Data Flow Analysis ........................... 26
      2.6.2 Data Flow Analysis in Presence of Closures ............... 31
# Denial of Service Attacks

3

## Slow HTTP Denial of Service Attacks

3.1 Slowloris

3.1.1 Slowloris

3.1.2 Slow Post

3.1.3 Implementation in NodeRED

## SYN flood

3.2 SYN flood

3.2.1 Implementation in NodeRED

## Denial of Service Attacks in MQTT

3.3 Denial of Service Attacks in MQTT

# Denial of Service Attack Detection

4

## Attack Complexes

4.1 Attack Complexes

4.1.1 Gremlin Queries

4.1.2 Locating Attack Complexes on Slowloris

4.1.3 Putting the Parts Together: Building Attack Complexes

4.1.4 Applying Attack Complexes to SYN flood Attacks

4.1.5 Conclusion

## Detection Refinements

4.2 Detection Refinements

4.2.1 Multi Targeted Socket Writes

4.2.2 Suspicious Strings

4.3 Further Work to Fine-Tune Code Property Graph for Denial of Service Detection

4.4 Detection Reports

4.5 False Positive Tests

# Conclusion

5
List of Figures

1.1 Motivational Scenario ........................................... 8
1.2 Example of a NodeRED Program ................................. 12
1.3 UML for JavaScriptToBlueprint ............................... 15
1.4 UML for NodeREDToBlueprints ............................... 16

2.1 Gremlin Graph Model ............................................. 18
2.2 Code Property Graph Legend ................................... 19
2.3 AST Graph ......................................................... 20
2.4 AST and CFG Graph ............................................. 21
2.5 AST, CFG and Control Dependence Graph .................... 23
2.6 Complete Code Property Graph Example ...................... 25
2.7 Data Dependence Example ......................................... 27
2.10 Code Property Graph with Externals and Multiple Files ... 34
2.11 NodeRED Code Property Graph ............................... 35

3.1 Native Slowloris Implementation .............................. 39

4.1 Code Property Graph of Slowloris ............................ 43
4.2 Code Property Graph with Global Context Object .......... 50
4.3 Dataflow via NodeRED Messages .............................. 51
4.4 Example for an Imprecise Data Flow Traversal .............. 52
4.5 Code Property Graph Inject Node Wired to TCP Out Node ... 56
4.6 Example for a Repeated setTimeout ........................... 60
4.7 Single vs Multi Targeted Write ................................. 65
4.8 Data Dependence vs Points-To Analysis ...................... 68
4.9 Field Insensitive vs Sensitive Analysis ....................... 69
4.10 DoS Detection Report ........................................... 71
List of Tables

2.1 Reaching Definition and Uses ............................................. 27
2.2 IN and OUT Sets ................................................................. 28

4.1 States of Loop Pattern Example ........................................... 44
4.2 States of Loop Pattern Example 2 .......................................... 48
4.3 States of Data Flow Traversal ............................................... 49
4.4 States of Second Control Flow Query ..................................... 57
4.5 Iterations of Algorithm to Connect Triggers and Writes ............... 59
4.6 Categorising and Scoring of Reporting .................................... 70
Listings

2.1 Code Property Graph Example ........................................ 20
2.2 Abstract, Iterative Data Flow Algorithm .......................... 28
2.3 Combining Reaching Uses and Definitions ......................... 30
2.4 Closure Example .......................................................... 31
2.6 Supporting Closures in Data Flow Analysis ......................... 32
2.5 Event Handler Example .................................................. 32
3.1 Pseudo Slowloris ........................................................... 38
3.2 Pseudo SYN flood .......................................................... 41
4.1 Traversal to Locate Externals .......................................... 47
4.2 Data Flow Traversal ...................................................... 48
4.3 Imprecise Data Flow Example .......................................... 52
4.4 Variable Names to Follow ............................................... 54
4.5 Control Flow Traversal 1 ................................................. 54
4.7 Control Flow Traversal 2 ............................................... 55
4.8 Control Flow Traversal 3 ............................................... 55
4.6 NodeRED Event Extended Control Flow ............................. 56
4.9 Algorithm to Connect Trigger and Writes .......................... 58
4.10 setTimeout Detection Refinement .................................... 60
4.11 Alternative SYN Flood Trigger ....................................... 62
4.12 Multi vs Single Targeted Write ...................................... 64
4.13 Suspicious Strings Example 1 ......................................... 66
4.14 Suspicious Strings Example 2 ......................................... 67
Chapter 1

Introduction

The heterogeneity of the IoT makes connecting things a difficult and time-consuming job. NodeRED tries to help out with a visual programming approach, which makes it easier to try new ideas faster and has many plugins to connect to different devices. NodeRED was nominated for the Jax Innovation Award in 2015, so it is an up-to-date solution to connect devices in the IoT.

Visual programming enables a lot of people to write programs even if they are not able to use a programming language. This is great, but also provides the possibility to implement malicious programs to many people, who may have not been able to do so before.

DoS attacks are often quite simple to implement and the explanation of how to implement them is easily found on the internet [14, 13]. However they can cause a lot of harm due to unavailable services and can have high financial and non-monetary impact on their victim. These attributes make them especially dangerous in combination with visual programming as in NodeRED. The usually high frequency of messages in the Internet of Things could render some traditional protection techniques useless.

The setup of a public NodeRED server, allowing technical inexperienced users to run their programs on it, could be a paramount step to facilitate connections in the Internet of Things and towards higher publicity of NodeRED. However, the owner of this server may be liable for programs uploaded to it and launch a denial of service attack from his server. This scenario is depicted in figure 1.1. IBM offers NodeRED servers within their development platform IBM Bluemix. After a registration with any email address everybody is allowed to test this platform for 30 days. It is possible to create a NodeRED server instance running on Servers owned by IBM. All dependencies for a simple Slowloris attack are available. Furthermore the user is allowed to add custom packages and dependencies [1, 2].

The owner of the server would greatly mitigate the risk of being liable or having a program performing a denial of service attack at all if she or he could review all uploaded programs. But this approach becomes infeasible if the service wins popularity and many programs are uploaded every hour. A static analysis reporting suspicious programs, that may perform an denial of service attack, would be of great help and could extremely reduce the workload of the owner of the server. Static analysis has several advantages compared to a dynamic analysis, e.g. the program
Figure 1.1: Motivational scenario for this thesis: An attacker uploads a flow performing a denial of service attack to a public NodeRED server and the owner of this server is charged for the attack.

does not need to be executed; therefore, it cannot cause harm to the NodeRED server or any other computer. In addition, static approaches are often more scalable and normally faster than dynamic ones; so, this solution can grow with the publicity of the server.

The concept of a static analysis of this work is mainly inspired by Yamaguchi’s Code Property Graph [31]. Yamaguchi introduces a graph combined of the abstract syntax tree, control flow graph and program dependence graph to identify security problems in the Linux kernel. This graph is a powerful tool for static analysis since it merges syntax, control flow and dependence representations in one place. This thesis adds also a call graph to the Code Property Graph.

This thesis implements a Code Property Graph generation for general JavaScript code. On top of this a NodeRED specific Code Property Graph is built and analysed for denial of service attacks. Before the analysis starts, the graphs are saved in a Neo4J graph database to offer a standardised and well documented interface for further processing.

The main goals of this thesis are:

- Develop a framework to detect SYN flood, Slowloris and Slow Post attacks in NodeRED flows using a Code Property Graph.
- Find similarities between these attacks to generalise the detection process.
- Report the results to support the owner of a public NodeRED server to review programs uploaded to his server.
This work further contributes independent modules which could be reused for general JavaScript or NodeRED analyses:

- A Code Property Graph module for general JavaScript and another one specific for NodeRED programs.
- A generic framework to detect denial of service attacks using the Code Property Graph.

Some advanced static analysis techniques are not used in this work because they are very challenging in a highly dynamic and type less language like JavaScript. Many of them are still subjects in papers from 2014 and 2015, which offer rather fundamental solutions. E.g. Loop sensitive analysis [27], sound analysis of events [26] or type inference [25]. Therefore, these analyses, inter procedural and exception handling go beyond the framework of this thesis. Even without these techniques, it was possible to prove that the Code Property Graph approach is useful to identify NodeRED programs performing a denial of service attack. Indeed, inter procedural analysis and exception handling seem to be of low importance for NodeRED analysis; as, inter function interaction and exception handling are not widely used in NodeRED nodes source code or at least are not especially important to detect denial of service attacks. However, some special cases of inter procedural interaction as closures and some event analysis for NodeRED specific events are covered by this thesis.

This thesis is divided into 5 chapters. This chapter will introduce NodeRED, other used software, the design of the thesis implementation and relates the text to the state of the art of JavaScript static analysis and denial of service detection. Chapter 2 describes the generation of the different graphs making up the Code Property Graph and presents how they are merged together in one database. Then, in chapter 3 the implementation of SYN flood, Slowloris and Slow Post with NodeRED is explained; in addition the choice for these three attacks is motivated. The chapter Denial of Service Detection (4) presents the developed framework to detect denial of service attacks. The last chapter concludes the results of this thesis.
1.1 Related Work

**Static Denial of Service Detection in JavaScript** Static analysis approaches for denial of service attacks are very rare. That is because static analysis approaches need access to the source code of the programs to analyse. But in traditional denial of service attacks the victim or any other party interested in stopping the denial of service attack does not have access to the source code or at least does not know about it. E.g. if the attacker executes the denial of service attack on his own machine, he is not interested in detecting it. In case of a distributed denial of service attack the computer executing the source code is a victim itself, but the source code is already compiled, which makes it harder to detect denial of service attacks in it, and the owner of the computer has no control over it, otherwise he would not participate in a denial of service attack in the first place.

However, recent denial of service attacks abuse the browsers of unwitting users to execute JavaScript. The user visits a web page which includes the JavaScript code for a denial of service attack and the browser automatically executes it. This denial of service attack could become more popular in future since the performance of JavaScript execution in browser increased during the last years and web workers allow multi threading in browsers. The Chinese search engine Baidu, which is the fourth often visited web page of the world according to Alexa [3], hosted JavaScript performing a large and successful denial of service attack in March 2015 [20]. Attacks from a single browser can have the same impact as well known denial of service tools using multi threading on single computer [28]. The approach used in this thesis could be used to scan websites for denial of service attack source code.

**Denial of Service Detection** Two papers gave the author a substantiated overview about the state of art of denial of service detection [15, 18]. Nearly all of them are dynamic. They use signature based and statistical approaches to analyse network traffic and detect ongoing denial of service attacks. Hence, all these approaches are very different from the static approach of this text. Three main differences are:

- They are dynamic.
- Other approaches analyse the network traffic; the approach discussed analyses source code.
- The used techniques differ. On the one hand, there are statistical approaches and signature based detection and on the other hand there are traversals on graphs to identify patterns.

Therefore the approach presented in this thesis is basically different and cannot be compared to the most other detection mechanism. Static approaches have some advantages. They can detect an attack before it is executed. This makes the detection process safer. Static approaches are more scalable than dynamic approaches in most cases. Scalability is an important property in our scenario of a popular and busy public NodeRED server. Despite of the advantages static analysis can be applied to a very limited amount of scenarios compared to dynamic approaches because access to the source code is mandatory as explained above (section 1.1.1).
The Original Code Property Graph This thesis is inspired by a paper using the Code Property Graph to statically detect typically security breaches in C or C++ source code. In a paper from 2014 [31] the Code Property Graph is used the same way as here. It is saved in a graph database and traversals to detect common security problems like Buffer Overflow or Division by Zero are presented. These traversals are to support a code analyst to locate problematic parts in the code. The effectiveness of this approach is demonstrated by analysing the Linux kernel which revealed 18 previously unknown security breaches. This success makes the approach a strong candidate for denial of service detection in the eyes of the author. In this thesis the approach was adapted to JavaScript and the call graph was integrated in the Code Property Graph.

The approach explained in the paragraph before was extended in 2015 to automatically generate traversals to identify vulnerabilities [30]. The motivation to support code auditors locating security breaches stayed the same, but machine learning and graph mining techniques are used to find interesting traversals, rather than crafting them manually.

The paper aforementioned uses the Code Property Graph to identify accidentally introduced vulnerabilities; in contrast, this thesis applies the approach to detect attacks which were written on purpose. This hinders to apply the Code Property Graph to denial of service attack detection because the detection cannot rely on common code patterns since the attacker might use very different code structures to express the same semantics. Instead of discovering common code patterns, denial of service detection has to find invariant code patterns.

Static Analysis of JavaScript JavaScript offers a broad set of dynamic features [25, 27]. The inheritance structure of JavaScript can be change during execution, implicit type conversion and event based code execution are only some examples. This makes static analysis of JavaScript a challenge. Nevertheless, practical applications of static JavaScript analysis exist. JSHint and JSCS support interactive JavaScript linting (syntax checking and more) for multiple IDE’s [7, 6]. JSPrime and ra2-dom-xss detect cross site scripting on web pages [8, 12]. The Google Closure Compiler, which is the basis of the implementation of this thesis, offers minification of JavaScript and is maybe one of the most complex, practical used static analysing tools for JavaScript. So most practical applications of static JavaScript analysis are rather simple and have a clearly defined purpose. General and complex static analysis of JavaScript is still a subject of many recent papers and theses. These texts often deal with a single issue as loop-sensitivity [27], points-to analysis [22] or information flow [24]. Two theses from the last two years introduce more general approaches for static JavaScript analysis [23, 25]. This overview renders this thesis as up-to-date.
1.2 NodeRED

NodeRED is a visual programming interface to connect the Internet of Things. A NodeRED program looks like a flowchart; thus, it is also called flow. It consists of nodes which are connected by so-called wires. Along those wires messages are sent from the output of one node to another nodes input. The number of outputs and inputs of a node is controlled by its type, which also controls its whole behaviour. A NodeRED node type normally has a single, clearly defined purpose.

A sample program sending “Hello World” via TCP to localhost on port 80 is pictured in figure 1.2. Two kinds of nodes are used. The tcp out node is configured to open a TCP connection to localhost on port 80 and sends incoming messages to the other end of the connection. The inject node sends a new, configurable string message to its wired nodes. In this example, it is configured to send the string ‘Hello World’ once when the flow is started.

A list of all used node types in this work can be found below.

Figure 1.2: A NodeRED program sending "hello world" to localhost on port 80.

1.2.1 NodeRED Node Types

All node types used in this work are shortly explained here. Input and output ports are listed in braces behind the name.

**inject** (i: 0; o: 1) - An inject node generates a message and sends it to its successors when clicked on, once at start of the flow or in a configured interval. The payload and topic of the message can be configured, but in this work the content of the messages is mostly unimportant; it is only used to trigger the next node.

**tcp out** (i: 1; o: 0) - Establishes a TCP connection to a configured server. Incoming messages are send to this server.

**tcp request** (i: 1; o: 1) - As tcp out, but it outputs the response of its server.

**function** (i: 1; o: n) - A function node allows to the user to write a JavaScript function to process incoming messages and return zero or more new message objects. Therefore, a function node can have 1 or more outputs. The function is executed by NodeRED in a very restricted context containing only: setTimeout, the Buffer and util NodeJS module, some logging functions and a so called global context. The global context is an empty object, which is shared by all function nodes; so, they can exchange global information with it. In addition, it is possible to configure the NodeRED server to allow access to further NodeJS modules via the global context.

**delay** (i: 1; o: 1): Delays incoming messages for a configurable amount of time.
1.2.2 Custom NodeRED Nodes Types

NodeRED allows the user to specify his own node types. The attacker in the scenario of this thesis is allowed to upload his own nodes. This allows the author to implement different versions of each attack to verify that the detection is working on them. The aforementioned NodeRED servers offered by IBM allow the user to upload his own node types; therefore, the assumption is quite realistic.

Custom NodeRED node types implemented during this work are explained here:

**GreedyTCPNode** (i: 1; o: 0) - A node opening as much as possible TCP connections to a server. Sends incoming messages to all of them. Used by Slowloris and Slow Post.

**raw-socket** (i: 1; o: 0) - Opens a raw socket and sends a SYN request for every incoming message. Used by SYN flood.

**Slow Post** (i: 0, o: 0) - Implements a complete Slow Post attack.

**SYN flood single** (i: 0, o: 0) - Implements a complete SYN flood attack.

1.2.3 NodeRED Functions and Events

Two NodeRED functions are repeatedly mentioned in this text; they are be explained here.

**input event** - The input event is triggered when a node receives a message. The input handler is the function that handles the input event or in other words the function that is executed when a node gets a message. The call to events. EventsEmitter.prototype.emit(’input’) on a NodeRED node will trigger the nodes own input handler.

**RED.Node.prototype.send(msg:Object)** - This method is defined on all NodeRED node types. It sends the message msg to every other node connected to the outports of node it is called on. As result the input handler of these nodes is executed.

1.3 Used Software

This section introduces software used for this thesis. This work is based on the Google Closure Compiler. It is used to compile JavaScript code and generates some intermediate representations of it. The produced Code Property Graphs are saved in a graph database, namely Neo4J. Gremlin is used to query this database and describe traversals on it.

The Google Closure Compiler is a JavaScript compiler. It is developed to minimize the source code and it offers many static analysis techniques for JavaScript. The abstract syntax tree and the control flow graph of the Closure Compiler are used as a fundament for the Code Property Graph generation of this work. The Closure Compiler also greatly supports the data dependence and call graph generation of this thesis.
Neo4J is a well-known, innovative database representing its data as graphs. Therefore, it is perfectly suited to store a Code Property Graph and provides a defined interface to interact with this graph.

Gremlin is a framework to query Neo4J and other graph databases. Due to the use of Gremlin the denial of service detection could also run on other graph databases than Neo4J. Gremlin is used to build parts of the Code Property Graph and to query it for denial of service detection.

1.4 Software Design

The software implemented for this thesis is divided into two modules. JavaScriptToBlueprints generates a Code Property Graph database for JavaScript programs. It uses the Closure Compiler to construct the abstract syntax tree, control flow graph and call graph of the script and produces a program dependence graph itself. This part of the work deals with general, non-detection specific static analysis methods. The package offers a thin interface for reuse ability in further research.

NodeREDToBlueprints encapsulates the NodeRED and detection specific part of this thesis. The denial of service detection is not a package on its own because it is not NodeRED agnostic. Therefore it cannot be modularised cleanly. Anyhow, all listing of denial of service attacks in this text can be detected.

This modularisation has the advantage that the Code Property Graph generation can be easily reused in other projects. In addition it supports further development on this project by dividing fundamental static analysis from detection specific analysis. This will enormously facilitate the decision where to implement subsequent, more precise analysis.

This chapter facilitates the understanding of the implementation by explaining the connection between the classes. A class diagram for every module is presented. Details about algorithms used can be found in later chapters.

1.4.1 JavaScriptToBlueprints

The Compiler is the interface of this package. It uses the Google Closure compiler to generate the abstract syntax tree, the control flow graph, call graph and program dependence graph from a JavaScript program and saves them to a Blueprints database.

The generation of the different program representations are encapsulated in classes used by Compiler. WriteAstToDatabaseCallback is a callback to traverse the abstract syntax tree from the Closure Compiler; visited nodes are written to the database in preorder. The control flow graph is saved by the method writeCFGToDatabase of Compiler. Call graph, data dependence graph and control dependence are stored by the Compiler using CallGraphGenerator, DataDependenceGenerator and ControlDependenceGenerator. DataDependenceGenerator analyses the data-flow with MaybeReachingDefinition and MaybeReachingUses. To generate control dependence of a program a post dominator tree is necessary. This tree is constructed
Classes in the package `com.google.javascript.jscomp` beginning with `Access` expose package local classes of the Closure Compiler to the default package.

Classes in the `db` package contain constants to communicate with the database, e.g. the labels for the nodes and edges or the property names.

---

**Figure 1.3:** UML class diagram for the package `JavaScriptToBlueprint`.

---

### 1.4.2 NodeRedToBlueprints

The `DoSDetectionRunner` is the interface to run detection for SYN flood, Slowloris and Slow Post on flows. It takes the JSON representation of the flows, generates the Code Property Graph databases using `NodeRedProgramDatabaseGenerator` and analyses them with all available `DoSDetectors`. The detectors return `DoSDetectionResults` which are handed to a `DetectionReporter`. Until now the only concrete reporter is the `HTMLReporter` exporting results to a HTML file. `SynFloodDetector` and `SlowHTTPDetector` are two implementations of `DoSDetector`. Both detectors use `AttackComplexes` to analyse the flows. Therefore, they return an `AttackComplexDetectionResult` which contains some information about the located Attack Complexes. All Attack Complexes in a program are discovered and represented with the class `AttackComplexes`. A single Attack Complex is represented by `AttackComplex`. This data structure is comprised of `Triggers`, `CallToSends` and `SocketCreations`.

The generation of a database for a flow is a two step process. First `GenerateNodeRedDatabases` uses the `Compiler` from `JavaScriptToBlueprint` to generate a database containing Code Property Graphs for every NodeRED node type. Then the `NodeRedProgramDatabaseGenerator` copies the graphs from the NodeRED node
type database to a database representing the flow. Additionally `NodeRedProgramDatabaseGenerator` uses the `Compiler` to compile the function nodes of the flow.

`FalsePositiveTesting` runs the detection on false positive flows from the NodeRED webpage. This is not done by the `DoSDetectionRunner` because results are logged in a different way. The results are logged directly after a flow was analysed by using a `DetectionFinishedCallback`. This helps to find false positive flows which contain JSON syntax errors and could cause the detection process to abort.

`CommonQueries` is a utility class containing Gremlin traversals which are used multiple times. `NodeRedNode` accommodates traversals concerning vertices representing a NodeRED node.

`AstTypes`, `DbEdgeLabels` and `DbProperties` contain constants for database querying. `DoSAttacks` and `NodeRedTypes` hold constants for known denial of service attacks and NodeRED node types.

![UML diagram for the package NodeREDToBlueprints.](image)

Figure 1.4: UML diagram for the package NodeREDToBlueprints.
Chapter 2

Code Property Graph

The name Code Property Graph was coined by Yamaguchi, Golde, Arp and Rieck in their paper for the IEEE Symposium 2014 [31]. The Code Property Graph is a graph containing the abstract syntax tree, control flow graph and program dependence graph of a program. These three program representations are merged together to form a powerful combined representation by incorporating syntax, control flow and program dependence information in one graph. This compound offers the possibility to use all these different representations in the same context. In addition to the three aforementioned graphs, this thesis also merges a call graph to the Code Property Graph. As a result, the meaning of the name Code Property Graph in this work is slightly different from the definition by Yamaguchi [31]. It means the combination of abstract syntax tree, control flow graph, program dependence graph and call graph.

The Code Property Graph of this work is stored in a Neo4J database and is traversed with Gremlin traversals. In section 2.1 the format of a Neo4J / Gremlin database is explained. Later sections in this chapter present the database generation process. The generation process is described in steps. Every section adds one of the Code Property Graph sub graphs to the database. The database is built from a single example code listing used in all sections and every section contains a figure of the database after adding its graph.

The generation process of control and data dependence is described in depth since the algorithms used were implemented by the author. The methods to generate the other graphs are provided by the Closure Compiler and are not described here. However, used classes from the Closure Compiler and additional work by the author is mentioned in order to support readers interested in the implementation.

Some special cases and the representation of a NodeRED program in the data base are explained in the last sections of this chapter. JavaScript programs can contain multiple files and use external functions which are not defined by them, but by the runtime environment. Section 2.8 and 2.9 describe how this cases can be accommodated with the database schema of this thesis.

NodeRED programs are not directly written in JavaScript. They are coded by adding nodes to a flowchart and connecting them. NodeRED instantiates a JavaScript object per node and operates on this data structure to execute the flow. Section 2.10 describes how this dynamic execution process can be statically modelled in a database.
2.1 Gremlin Graph Format

Both Neo4J and Gremlin use a directed “property graph” as a basic scheme to save data. Since Gremlin is used later to traverse the graph, its terms will be used to explain the scheme.

A gremlin graph consists of vertices and directed edges connecting them. Both have a unique id and an arbitrary amount of properties which are key value pairs. Every edge has also one label. This is a single string value describing the type of relationship represented by the edge. A simple gremlin graph is shown in figure 2.1. Edge1 is called an ‘out edge’ of vertex1 and ‘in edge’ of vertex2. vertex1 is the ‘out vertex’ and vertex2 is the ‘in edge’ of edge1.

Figure 2.1: Gremlin graph model

2.2 Abstract Syntax Tree

The abstract syntax tree represents a JavaScript program as a tree. Nodes in the tree are representing syntactic parts of the program like constants, variables or operators. An example showing the source code and the generated syntax tree is given in listing 2.1 and figure 2.3. The legend for figures of Code Property Graphs is shown in figure 2.2.

The abstract syntax tree is one of the most basic representations of a program. It will be written before all other representations in an empty database. The root of the abstract syntax tree is marked with the property IS_AST_ROOT. There is exactly one root in the database. The edges of the abstract syntax tree will be labelled AST_PARENT_OF. All vertices of the database are reachable by recursively following the AST_PARENT_OF out edges from the root. A syntax tree vertex has the properties IS_AST_NODE and AST_TYPE. The type of syntax tree vertex is the type of program structure it represents, like WHILE for a while loop or VAR for a variable declaration. The order of children in the syntax tree is important. This order is represented by the AST_CHILD_RANK property on each AST_PARENT_OF edge.

Constant types like STRING or NUMBER or references to a variable can be represented by a string or number. They have the properties STRING_VALUE and NUMBER_VALUE.

Implementation In this thesis the Closure Compiler class Compiler is used to generate syntax trees and the author implemented a class (WriteAstToDatabaseCallback) to save the tree into a database.
Figure 2.2: Legend for figures of Code Property Graphs. Vertices from the different graphs are not displayed differently. Every vertex belongs to the syntax tree, every vertex with a line number belongs to the CFG and some CFG vertices belong to the dependence and/or call graphs.
function helloWorld() {
    alert('hello world');
}
helloWorld();
for(var i = 10; i > 0; i--) {
    if (i % 2 == 0) {
        alert(i);
    }
}

Listing 2.1: The source code of the syntax tree in figure 2.3.

Figure 2.3: Database after writing the AST from listing 2.1. This listing is used as an example through the whole chapter.
2.3 Control Flow Graph

The control flow graph models the control flow of a program. Figure 2.4 shows the control flow graph for the source code of the listing 2.1.

The control flow graph is saved after the abstract syntax tree. For all but its exit vertices it uses the same vertices as the syntax tree. Every function in the source code has its own control flow graph, starting at the syntax tree vertex of type FUNCTION. On this node the property CFG_NODE will be set to CFG_ENTRY_VALUE. On an exit vertex this property will have the value CFG_EXIT_VALUE. On all other control flow vertices it is set to true. A function has exactly one entry and one exit vertex. All return statements of the function are connected to this exit vertex.

The edges in a control flow graph point in the same direction as the control flow. The statement represented by the source vertex of the edge is executed before the statement of the destination vertex. The edges are labelled:

- CFG_UNCONDITION if the statement from the out vertex is always followed by the in vertex statement.
- CFG_TRUE or CFG_FALSE if the execution of the next statement depends on the result of the former statement.
- CFG_ON_EX if the control flows to the in vertex when an exception occurs.

A control flow vertex has at most two outgoing edges plus a CFG_ON_EX edge if an exception handler is attached to it.
If $N$ is the vertex count of the database before adding the control flow graph, then $N + \#Functions$ is the vertex count after adding the control flow graph. Writing the control flow graph adds one exit node per function. Except for these vertices only new edges between existing vertices are added.

**Implementation** The class `ControlFlowAnalysis` is used to generate a `ControlFlowGraph`; both classes are implemented by the Closure Compiler. The class `Compiler` by the author uses them to save the control flow graph into a database.

### 2.4 Program Dependence Graph

The program dependence was first mentioned in a paper by Ferrante, Ottenstein and Warren in the year 1987 [19]. The program dependence graph models control and data dependence of a program and was originally invented to speed up automatic program optimisations by combining these two dependencies in one graph. However, the generation processes of control and data dependence are independent from each other; therefore, this work separates them in different sections. Furthermore, no special section about combining the dependence graphs exists since this process naturally fits into the merging of Code Property Graph. In fact, this work generates, merges and uses the two dependence graphs independent of each other and views them as two separated components of the Code Property Graph.

### 2.5 Control Dependence

Control dependence will exist between two statements $s_1$ and $s_2$, if the execution of $s_2$ depends immediately on $s_1$. A formal definition is given in appendix A. The generated database 2.5 contains several examples:

- The `if` at line 6 is control dependent on the `for` (line 5) since it is only executed if the `for` condition evaluates to `true`.
- The `alert` at line 7 is control dependent on the `if` (line 6) for the same reason. But it is not control dependent on the `for` because it is not immediately controlled by the `for` condition.
- The `for` is control dependent on itself because it might be executed again if executed once.
- The `for` is not control dependent on the call at line 4 because the call is not controlling the execution of it.

The control dependence graph adds only edges to the database. As with the control flow graph they point in the direction of the control flow. They are labelled `CD_ON_TRUE`, `CD_ON_FALSE`, `CD_ON_EX` and `CD_ON_NON_EX`, depending on the type of control dependence between the in and out vertex.
**Implementation**  The Closure Compiler does not support control dependence generation or something similar that could be used to help except for control flow graph generation which is necessary for most control dependence algorithms. Thus, the author decided to use the algorithm which is used by Yamaguchi [31] to stay close to the original Code Property Graph implementation. This algorithm is explained and proved to be correct in [19]. It requires a control flow graph and a post dominator tree for the program. As explained above, the control flow graph can be generated with help of the Closure Compiler, but the author needed to provide a post dominator tree. Several algorithms to generate a post dominator tree are known with asymptotic speeds ranging from $O(n^4)$ to $O(E \cdot N \log \log \log N)$ (with $E$ and $N$ as number of edges or nodes in the control flow graph). Since the algorithm used by the Ferrante’s reference paper was not available to the author, he needed to decide on one algorithm himself. The decision process is described below.

The Lengauer-Tarjan algorithm is the most widely used one to generate post dominator trees[17]. With a timebound from $O(E \cdot \log(N))$ it is one of the fastest known (and $E$ and $N$ are the number of nodes and edges in the control flow graph). The paper [17] compares this algorithm with a simpler iterative one running in $O(N^2)$. They prove that the asymptotic slower algorithm is faster in practice if using a suitable data structure. They measure that control flow graphs with more than 30,000 nodes are necessary before the Lengauer-Tarjan is faster than the simple algorithm. The graph representing a NodeRED program used in this thesis has at most 400 control flow graph nodes. Following the argumentation of the paper mentioned the iterative algorithm was implemented as part of this thesis in the class PostDominatorTree.
The algorithm from [19] is implemented in ControlDependenceGenerator. The algorithm to generate post dominator trees is contained in the class PostDominator Tree.

2.6 Data Dependence

A statement is data dependent on another statement if it needs data from the first statement to be computed. There are four different scenarios where \( s_2 \) depends on \( s_1 \):

- flow dependence: \( s_1 \) writes memory that is read by \( s_2 \)
- anti-dependence: \( s_1 \) reads memory that \( s_2 \) later writes
- output dependence: \( s_1 \) writes memory that is written by \( s_2 \) later
- input dependence: \( s_1 \) and \( s_2 \) read the same memory

To detect DoS attacks flow dependence is the most important. Because it helps to answer questions like: Does the statement \( s_2: \text{so.write('POST to/to HTTP 1.1')} \) write to a tcp socket? This can be described with a flow dependence from \( s_1: \text{var so = new Socket()} \).

In figure 2.6 two flow dependence edges were added to the database. Both edges end at the \texttt{for} at line 5 and start from the statements of the lines 6 and 7 because the \texttt{for} writes the variable \( i \) which is read by the latter statements.

The data dependence graph consists of \texttt{USES} edges. They are pointing from a control flow vertex using a variable to the control flow vertex defining this variable. These edges have the property \texttt{VARIABLE_NAME} set to the name of the variable used. Every control flow vertex could have \( a \in N_0 \) outgoing \texttt{USES} edges with a disjunct set of variable names. Multiple uses of one variable in the same statement will be represented by one edge. Per vertex \( b \in N_0 \) incoming edges can exist, without any constraint on the \texttt{VARIABLE_NAME} property. \texttt{var i=1, j=2} produces two variables with the names \( i \) and \( j \) which can be used more than once. This will result in one edge per use; so, it is possible to have two incoming edges for \( i \) and one for \( j \) if \( i \) is used twice and \( j \) once.
Figure 2.6: Database after writing the AST, CFG, control dependence, data dependence and call graph from listing 2.1. (Call graph and data dependence are added in this figure.)
2.6.1 Data Flow Analysis

To detect data dependence, data flow analysis is performed. Most data flow analyses use “intraprocedural data flow analysis” as a basis, that is flow information for single functions. Intraprocedural analysis can be followed or extended to interprocedural analysis. First the intraprocedural algorithm used in this work will be explained and in chapter 2.6.2 the extension for closures will be presented. This section does not use the listing 2.1 as example because it does not contain a good example for data dependence and would become too complex to show the whole Code Property Graph otherwise.

In this thesis, two data flow analyses are used. The first is the maybe reaching definition analysis which calculates if a definition of a statement could be used by another statement. The second is the maybe reaching use analysis; this analysis determines if a variable use of one statement uses the definition of a former statement.

The listing 2.7 and the table 2.1 give an example of both analyses. First the example of the reaching definition analysis will be explained. The alert (line 10) is reached by the definition of i in line 4 because this definition overwrites the one as function parameter in line 1. It is also reached by both definitions of j (lines 6 and 8) since it is unknown which branch of the if in line 5 will be taken.

To determine which definition reaches a statement, the control flow graph is traversed and the nodes are annotated with reaching definitions. The traversal starts at the entry of the function and walks forward to the return nodes. The direction of the traversal is important, because definition happens before use. While the control flow graph is traversed, every node \( n \) will be annotated with a set of definitions reaching it and a set of definitions available after this node. These sets will be called \( \text{IN} \) and \( \text{OUT} \). \( \text{IN} \) is computed as union of all \( \text{OUTs} \) of the predecessors. \( \text{OUT} \) is determined by adding the generated and removing the killed definitions of \( n \), e.g. the node representing \( i=1; \) (line 4) kills the former definition of \( i \) from line 1 and generates a new definition, which will be propagated. The \( \text{IN} \) and \( \text{OUT} \) sets for the example listing are shown in table 2.2.

The explanation of the maybe reaching use example is quite similar, but the problem is analysed from the uses point of view not from the definitions. The use of \( i \) from line 10 only reaches the assign to \( i \) (line 4), but not the function parameter \( i \) because the latter is overwritten by the assignment. Both definitions of \( j \) in lines 6 and 8 may be used in line 10; as a result, both definitions are reached by the use.

Determining reaching uses works the same way as reaching definitions, but traverses the control flow graph backwards. The traversal starts at the return nodes and traverses to the entry node of the function since uses appear after definition in the control flow graph. Hence, information about uses need to be propagated upwards.
```javascript
function toBeAnalysed(i) {
    // Reaching definition analysis starts here
    alert(i);
    i = 2;
    if (someBool) {
        var j = 1;
    } else {
        var j = 2;
    }
    alert(i + j);
    // Reaching use analysis starts here.
}
```

Figure 2.7: Listing to explain data flow analyses. Control dependence, call graph and some syntax tree edges are omitted in the Code Property Graph for clarity. Reaching definitions and uses are shown per line in the table 2.1.

<table>
<thead>
<tr>
<th>Line</th>
<th>Reaching Definitions</th>
<th>Reaching Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>i: 3</td>
</tr>
<tr>
<td>3</td>
<td>i: 1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>i: 10</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>j: 10</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>j: 10</td>
</tr>
<tr>
<td>10</td>
<td>i: 4 j: 6, 8</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Shows reaching definitions and uses from the listing 2.7 per variable and line.
Table 2.2: The IN and OUT sets for the listing 2.7 per line. Numbers behind the variablenames are line numbers of the reaching definition or use.

<table>
<thead>
<tr>
<th>Line</th>
<th>Reaching defintion IN</th>
<th>OUT</th>
<th>Reaching use IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{}</td>
<td>{i: 1}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>3</td>
<td>{i: 1}</td>
<td>{i: 1}</td>
<td>{}</td>
<td>{i: 3}</td>
</tr>
<tr>
<td>4</td>
<td>{i: 1}</td>
<td>{i: 4}</td>
<td>{i: 10}</td>
<td>{}</td>
</tr>
<tr>
<td>5</td>
<td>{i: 4}</td>
<td>{i: 4}</td>
<td>{i: 10}</td>
<td>{}</td>
</tr>
<tr>
<td>6</td>
<td>{i: 4}</td>
<td>{i: 4; j: 6}</td>
<td>{i: 10; j: 10}</td>
<td>{i: 10}</td>
</tr>
<tr>
<td>8</td>
<td>{i: 4}</td>
<td>{i: 4; j: 8}</td>
<td>{i: 10; j: 10}</td>
<td>{i: 10}</td>
</tr>
<tr>
<td>10</td>
<td>{i: 4; j: 6, 8}</td>
<td>{i: 4; j: 6, 8}</td>
<td>{}</td>
<td>{i: 10; j: 10}</td>
</tr>
</tbody>
</table>

Listing 2.2: Abstract, iterative data flow analysis algorithm.

```python
worklist = Set(all statments of a function)
for s in worklist:
    In(s) = []
    Out(s) = []
while !worklist.isEmpty():
    s1 = worklist.pop()
    In(s1) = join(Out(predessors(s1) ? forwards : successors(s1)))
    temp = flowthrough(s1, In(s1))
    if temp != Out(s1):
        Out(s1) = temp
        worklist = worklist.union(successors(s1) ? forwards : predecessors(s1))
```

Introducing the Abstract Algorithm  In this work an iterative worklist based algorithm for intraprocedural data flow is used (shown in listing 2.2) and extended to work with closures later. This algorithm is similar to the one mentioned in the Harvard lecture notes by Jeff Foster 2011 [21]. The functions predecessors and successors are given by the control flow graph of the function to analyse. flowthrough is in general implemented as $Gen(s) \cup (In(s) \cap Kill(s))$, with $Gen$ as information generated and $Kill$ information invalidated by s. E.g. the statement $s1: i = 1$; from line 4 will generate “i defined in s1” and kill the information “i defined as function parameter in line 1”.

The implementation of join is driven by the decision whether the result should contain only information that must hold true or information that could be true. The statement from line 10 has two predecessors. These are the two branches from the if in line 5. So by implementing join as an intersection just information that is true on both branches will be propagated. If join is a union, also information which is correct on only one of the two branches will be available in the IN set of line 10. In this work the join is always a union. Because to detect attacks information from both branches should be considered. That kind of analysis is called “Maybe
analyses” according to [21].

As stated above reaching definitions analysis has to traverse the control flow graph forwards, while reaching uses analysis has to traverse backwards. This behaviour is controlled by the boolean forwards. Depending on this property reaching definitions is called a “forward analysis” and reaching uses is called a “backward analysis”.

**Presenting the Concrete Implementation** This algorithm is already implemented by the Closure Compiler [4]. It offers an abstract class DataflowAnalysis with the abstract methods JoinOperation, flowtough and isForward to implement the algorithm 2.2.

In this work there are two subclasses of DataflowAnalysis the MaybeReachingUse Analysis and the MaybeReachingDefinitionAnalysis.

The MaybeReachingUseAnalysis is provided by the Closure Compiler. In this project it is just slightly changed. The implementation by the Closure Compiler ignores all uses from variables not defined in the analysed function. To support closures the analysis now also takes variables from parent functions in account, which are passed in as a parameter.

The Closure Compiler implements a MustReachingDefinitionAnalysis. This is a reaching definition analysis as explained above, but it collects only information that must be true for all paths through the function. The MustReachingDefinitionAnalysis got refactored to the MaybeReachingDefinitionAnalysis of this project, by changing the JoinOperation from an intersection to an union over all predecessors.

These two classes are used by the class DataDependenceGenerator to generate data dependence edges in the Code Property Graph. The DataDependenceGenerator was implemented by the author of this thesis.

**Combining Reaching Definitions and Uses** The DataDependenceGenerator traverses the abstract syntax tree and has two functions enterScope and exitScope which are called, when the traversal enters or leaves a function - scope and functions are the same in JavaScript. Every time a function is entered the MaybeReachingDefinitionAnalysis is used to generate reaching definition annotations and when the traversal leaves a function MaybeReachingUseAnalysis is used to get reaching uses annotations about all statements of the function. These annotations are saved and processed later in generateDataDependence to generate data dependence edges. Listing 2.3 shows the pseudo code of enterScope, leaveScope and generateDataDependence.
```java
class DataDependenceGenerator {
    Map<Node, ReachingDefinitions> mDef; // Node is an ast and cfg node
    Map<Node, ReachingUses> mUse; // ReachingUses and
    ReachingDefinition have no direct correspondent in the real code.

    void enterScope(Node scopeRoot) {
        cfg = new ControlFlowGraph(scopeRoot);
        rda = (new ReachingDefinitionAnalysis()).analyse(cfg);
        mDef.putAll(rda.getNodeAnnotationMap()); // A node appears only
        in one scope, so putAll never overwrites values.
    }

    void leaveScope(Node scopeRoot) {
        cfg = new ControlFlowGraph(scopeRoot);
        rda = (new ReachingUsesAnalysis()).analyse(cfg);
        mUse.putAll(rda.getNodeAnnotationMap());
    }

    void generateDataDependence() {
        for (Node n : mDef) { // for every node
            for (ReachingDef def : mDef.get(node)) { // for every
                definition reaching node n
                Node nDef = def.getDefiningNode();
                ReachingUses uses = mUse.get(nDef);
                if (uses.containsNode(n)) { // if node nDef is reached by the
                    use from node n
                    // add use edge from n to nDef to database.
                }
            }
        }
    }

    Listing 2.3: Generate data dependence by combining reaching use and reaching
    definition information.
}
```
2.6.2 Data Flow Analysis in Presence of Closures

Closures are used very often in JavaScript and NodeRED. Basically, closures are nested functions allowing the usage of variables defined in the outer function in the inner function. In general, data can flow from the outer function to the inner function and vice versa; so to speak, in both directions. This is demonstrated in listing 2.4 and table 2.8.

```javascript
function outerFunction() {
    var outerVar = 42;
    function innerFunction() {
        alert(outerVar);
        outerVar = 23;
    }
    innerFunction();
    alert(outerVar);
}
```

Listing 2.4: Example of a Closure. The outer function is data dependent on the inner function and vice versa. This is presented in table 2.8.

<table>
<thead>
<tr>
<th>Line</th>
<th>OuterVar value</th>
<th>Used definition (line)</th>
<th>Dataflow direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>42</td>
<td>2</td>
<td>outer ⇒ inner</td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td>5</td>
<td>inner ⇒ outer</td>
</tr>
</tbody>
</table>

Figure 2.8: Demonstrates direction of dataflow in listing 2.4.

Closures in NodeRED are typically used as event handlers (as shown in listing 2.5). If the event input occurs the innerFunction will be executed. NodeRED uses this pattern to handle incoming messages to nodes. The use as event handler is a special case of closure use. Since JavaScript is a single threaded language, it is impossible for the event handler to be executed before the outer function was completely processed. This prevents usage of definitions from the inner function in the outer function (demonstrated in table 2.9). Hence, data can flow only in one direction - from the outer to the inner function. Furthermore, it guarantees that only the last definition of a variable from the outer function is used in the inner function. These two constraints are not guaranteed for closures in general as example 2.4 shows. Thus, analysing dataflow of event handlers is easier than the analysis of closures in general. The closure data flow analysis of this thesis excludes cases, where an inner function is called from an outer function. This offers the possibility to concentrate on closure used most often in NodeRED.

With the constraints discussed above in mind, uni directional data flow and using only the last definitions of the outer function, only slight changes from the algorithm shown in 2.3 are necessary to extend the data flow analysis to handle closures.

The definitions reaching the last node of the outer function will be added to the IN of the entry node of the inner function. So they are treated as all other reaching
definitions in the inner function. Listing 2.6 shows changes necessary to the initial phase of the used algorithm 2.2. The rest stays unchanged.

```javascript
worklist = Set(all statments of a function)
for s in worklist:
  if s.isEntryNode:
    In(s) = Out(returnNodeOfOuterFunction)
  else:
    In(s) = []
    Out(s) = []
# Algorithm as above in listing 2.2.
```

Listing 2.6: Changes to the initial phase of the algorithm from 2.2 to support closures.

The definitions reaching the last node of the outer functions are available when the reaching definition analysis of the inner function starts. That is because the analysis takes place when the scope of the function is entered and the outer scope is always entered before the inner scope.

Uses of the inner function can be propagated in a similar way.

```javascript
function outerFunction() {
  var outerVar = 42;
  function innerFunction() {
    alert(outerVar);
    outerVar = 23; \ Never used
  }
  this.on('input', innerFunction);
  alert(outerVar);
  outerVar = 0;
}
```

Listing 2.5: Example of closure used as event handler. Dataflow is unidirectional.

<table>
<thead>
<tr>
<th>Line</th>
<th>OuterVar value</th>
<th>Used definition (line)</th>
<th>Dataflow direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>9</td>
<td>outer ⇒outer</td>
</tr>
<tr>
<td>8</td>
<td>42</td>
<td>2</td>
<td>outer ⇒inner</td>
</tr>
</tbody>
</table>

Figure 2.9: Demonstrates direction of dataflow in listing 2.5.
2.7 Call Graph

A call graph consists of nodes representing calls and functions. The edges connect calls to their possible targets. The call graph is generated by the class Callgraph from the Closure Compiler. The author added the class CallgraphGenerator writing the results from the Closure Compiler to the database and adding edges for external functions (functions not defined in the input files, see also 2.9), which are not linked to their call sites by the Closure Compiler. Figure 2.6 shows the call graph for the listing 2.1.

While writing the call graph into the database, edges between syntax tree vertices of type CALL and vertices of type FUNCTION are created. Every call and every function vertex can have $n \in \mathbb{N}_0$ incoming and $m \in \mathbb{N}_0$ outgoing edges because a function can be called more than once and for a call multiple target functions could be found. If a call has multiple targets the call graph construction cannot clearly determine the target and each linked function vertex may get called.

2.8 Programs with Multiple Files

One file has one syntax tree, but a JavaScript program might include multiple files. This needs to be expressed in the database.

If more than one file is parsed into a database, they have the same syntax tree root. The root has a one child of type BLOCK. This block has one child of type SCRIPT for each file. So there is one syntax tree for all files. This representation is adopted from the Closure Compiler. It implies that all scripts have the same global name space. This is the case if they are used together in a website or by NodeJS, making a useful and realistic implication in most cases.

In figure 2.10 a database for the example listing and one unknown second input file is shown.

2.9 External Functions and Definitions

Most JavaScript programs will use some external functions not contained in their source code, like console.log or setTimeout. These functions are implemented by the browser, NodeJS or in general by the environment the program runs in. Since the source code of these functions is not always available, their interfaces will be handed to the Closure Compiler as so-called externs. These interfaces are described in JavaScript files containing definitions and comments declaring parameters and return types.

For externs only the abstract syntax tree is generated and saved in the database. This abstract syntax tree contains only the definitions of the externs, but not their implementation. The other graphs of the Code Property Graph are not generated for externs. The syntax tree of external files is added as first child of the root (shown in 2.10). Call edges as described in section 2.7 connect them to the matching CALL
vertices of the represented program.

Figure 2.10: Database with two input files and two external files. One external file contains a definition for the function `alert` and call graph edges are shown.
2.10 NodeRED Flows

A NodeRED program consists of one or more nodes. These nodes are connected by wires. Along those wires messages can be sent.

NodeRED executes a flow by generating a JavaScript object of the correct type for each node in the flow and propagating messages via events. It is very hard to model this dynamic behaviour directly in the database. Therefore the structure of the flow is explicitly mirrored in the database. The database for the flow 1.2 is shown in graph 2.11. This flow is comprised of one inject node wired to a tcp out node.

Figure 2.11: Database representing the flow 1.2. This flow contains one inject and one tcp out node.

For every node in the flow, a vertex with the property IS_RED_NODE = true exists. This vertex has the properties RED_ID and RED_TYPE set to the id and the type of its node. Each of these vertices has an incoming edge labelled CONFIG_OF connecting it to a RED_CONFIG vertex, which is modelling the configuration of the node. An edge labelled AST_OF points from the syntax tree root of the Code Property Graph representing the JavaScript object of the node to the NodeRED node vertex. Nodes which are wired to each other are connected with WIRES_TO edges. Calls to RED.Node.prototype.send in the first node will trigger the input handler of the second node; therefore, these calls will be linked with SENDS_TO edges to the input handler of the receiving node. This edges are not shown in the example graph.
2.11 Automatic Code Refactors

The Closure Compiler offers a wide range of refactors called compiler passes. Some of them help to analyse the program by changing its syntax, but not its semantics. This results in an abstract syntax tree representing another syntax as the input source code.

The default configuration of the compiler from this work does not apply any Closure Compiler passes to the input. However, it offers the possibility to configure the internally used Closure Compiler to do refactors. This will change the generated syntax tree and is on own responsibility.

To simplify denial of service detection, Closure Compiler passes to inline variables and functions are applied to analysed NodeRED code (for further information see section 4.2.2).
Chapter 3

Denial of Service Attacks

To demonstrate denial of service attack detection Slowloris, Slow Post and SYN flood were chosen, because the first two attacks are implementable with nodes available in the standard NodeRED distribution and SYN flood might be the widest known denial of service attack. Since the idea behind of all three of them is undemanding to understand and the implementation is straightforward, they are especially dangerous in combination with visual programming. This combination enables people to launch a denial of service attack devoid of expert knowledge and without deep research.

3.1 Slow HTTP Denial of Service Attacks

Both Slowloris and Slow Post are using weaknesses in the HTTP protocol; therefore, they use normal, unmanipulated TCP connections which will be more difficult to differentiate from the ones used by real users, than TCP connections used by a TCP SYN flood attack.

3.1.1 Slowloris

Slowloris was first mentioned by Micheal Zalewski and Adrian Ilarion Ciobanu in the year 2007 and was implemented by Rsnake in 2009[16]. Slowloris establishes HTTP connections with the server until the victim will not open any more and tries to keep them alive as long as possible; as result, no other client will be able to connect anymore. Slowloris achieves through low bandwidth and processor time usage on the attacking machine by starting a HTTP GET request and sending endlessly HTTP headers with a very low frequency. In consequence the server is forced to wait for the completion of the request and cannot use this connection for a legitimate user. Listing 3.1 presents the pseudo code of Slowloris; in addition, the reference implementation used by the author can be found at [14].
function SlowLoris() {
    var a = [];
    while(victimHasFreeConnections()) {
        var s = new net.Socket('192.42.42.42');
        s.write('POST p.html HTTP/1.1\n');
        a.push(s);
    }

    setInterval(function sendHeader() {
        for (s in a) {
            s.write('x-header: value\n');
        }, 1000);
    }
}

Listing 3.1: Pseudo implementation of Slowloris

3.1.2 Slow Post

Slow Post was discovered in 2009 by Wong Onn Chee and his team[16]. It is very similar to Slowloris, but it uses POST requests and completes the headers as a normal request; after that, it sends the body with low frequency. On the one hand this is harder to detect. Since the header part is completed as in normal requests. On the other hand the server needs to accept POST requests; on consequence, there needs to be a form on the requested site, because some server will not expect POST requests otherwise.

3.1.3 Implementation in NodeRED

Different versions of Slowloris and Slow Post were implemented for this work; to prove that the detection works for more than a single form of implementation. One of the implementations demonstrates the possibility to program an attack in native NodeRED, without adding custom nodes. All possible implementation in pure NodeRED will be easy to detect; so, another version uses custom nodes. A third implementation variant uses a single custom node containing the attack to test the detection on this border case where a flow contains only one node. The three versions will be presented in the order as introduced above. Their JSON representation can be found in the NodeRedDoS folder. All implementation of Slowloris could be used to perform a Slow Post attack by applying minor changes and vice versa. Thus both attacks have been implemented in at least one version, but not every version was implemented for both of them.

Figure 3.1 shows the author’s native Slowloris implementation. The program starts at the Start inject node. The node is configured to send one message when the flow is started. This message triggers the Start request function node to send a message with the payload "GET index.html HTTP/1.0\n". The payload is send to
the attacked server by a TCP request node. It is necessary to use TCP nodes; since, both attacks use the HTTP protocol, but do not forge complete requests. These incomplete requests are not a legitimate use of the protocol; therefore, they are not supported by the HTTP nodes of NodeRED. The flow continues at the Trigger inject node. It is firing a message every 5 seconds. This messages activate the Next header function node. This node will send the payload "xheader: value\n" to the TCP request node. To conclude the Trigger and the Next header node repeatedly send new HTTP headers to the victim. After 10 minutes some attacked servers might time-out the HTTP request. If this happens it sends a HTTP response with the status code 408. The HTTP request nodes wraps this response into a message object and sends it to the Restart attack function node. This node will deactivate Next header node and propagate the message to the delay node. This node will delay propagation of the message for one second. This is enough time for the TCP request node to reconnect to the victim. After the Start request node is triggered to start a new request and enable the Next header node again.

Figure 3.1: Native NodeRED Slowloris implementation. This structure must be at least replicated 150 times to perform an attack.

Since a Slowloris needs many TCP connections, but a TCP request node offers only one and none of the native NodeRED nodes offer more. Therefore the structure presented above and shown in figure 3.1 must be copied to form a working attack. It needs to be replicated around 160 times to successfully block a Apache server with default configuration and more often to attack a real life server. This massive use of TCP request or TCP out nodes causes this implementation to be easily detected. In addition the author assumes that few legitimate applications will need that many TCP connections; so, detecting by counting used TCP nodes with the same destination address will be nearly false-positive free. Consequently a less obvious implementation is needed; however, this will need a NodeRED node which is openings multiple TCP connection at the same time. No native NodeRED node is supportings more than one connection; so, a custom node is necessary.

As result the GreedyTcpNode got implemented. This node type is openings connections until the victim will not open any more new connections and it is sendings incoming messages to all of them. The GreedyTcpNode can replace the TCP request node in figure 3.1; after, the attack is implemented with 7 nodes and replication is not necessary any more. The GreedyTcpNode internally uses a recursive function to instantiate multiple connections and a for loop to send messages to them. Thiese code sttures are much harder to detect, than replication.
The drawback of this implementation is that if one connection is timed out by the attacked server all of them must closed and reconnect since a fresh connection would be in another state than an old connection. To be more precise on a fresh connection the victim expects a payload like \texttt{GET index.html HTTP/1.0\n}, while an old connection is expected to send a header. But an incoming message is send to all connections. This can be improved by deleting the \texttt{Start} and \texttt{Start request} node and writing \texttt{GET index.html HTTP/1.0} to every connection directly after it is created. This improvement was not implemented.

As stated above the implementation in a single custom node is an edge case and as such interesting, but it has some more advantages. This version can be used to detect limitations of the detection easier than the former versions; since, the attacker fully controls the code. This control allows to fine-tune the complexity of the code to detect on. Slow Post was implemented as a single node.

This approach does not use special NodeRED features like messages in between nodes; consequently, it will make detection less dependent on NodeRED specifics and make it easier to reuse on JavaScript code of other applications.

### 3.2 SYN flood

The SYN attack is exploiting the TCP three way handshake. The attacker sends many TCP SYN packages to the victim, but never answers the TCP ACK replies from the victim. Therefore, the victim is forced to allocate resources for the half opened connections and cannot use this resources for other applications. The SYN flood may be the most commonly known denial of service attack; thus, it is very likely to be used; therefore, it is covered by this work. A pseudo implementation of it is shown in listing 3.2. The author used this python implementation as orientation [13].

#### 3.2.1 Implementation in NodeRED

To implement a SYN flood a raw socket is needed because a normal TCP socket always completes the three way handshake. Therefore, the NodeJS package ‘raw-socket’ was installed. Two versions of the SYN flood attack were implemented in NodeRED by the author. As for Slow Post a single custom node performing a SYN flood attack was created. The second version uses two nodes. An inject node sends one message per second to a \texttt{rawsocket} node. Every time the \texttt{rawsocket} node receives a message, it generates a SYN package and delivers it to the victim.

Unfortunately it was impossible to measure the effect of the implementations on an Apache server because although, the \texttt{rawsocket} library signalled that the SYN packages were send successfully, they were not send. They were also not listed in Wireshark. Due to the simplicity of the attack the author believes that no fundamental bugs were introduced in the flows; therefore, the implementations can bye legitimately used for the purpose of detection. An unproportional amount of time would have been necessary to indentify the problem, which is most likely a format mistake of the generated SYN packages or a misconfiguration.
var rawSocket = require('raw-socket');

var options = {
    addressFamily: rawSocket.AddressFamily.IPv4,
    protocol: rawSocket.Protocol.TCP,
}

var socket = rawSocket.createSocket(options);

socket.setOption(rawSocket.SocketLevel.IPPROTO_IP, rawSocket.SocketOption.IP_HDRINCL, 1);

var packet = generateTcpSynPacket();

setInterval(function sendPacket() {
    socket.send(packet);
}, 100);

Listing 3.2: Pseudo implementation of a SYN flood attack.

3.3 Denial of Service Attacks in MQTT

Since MQTT is a famous protocol in the Internet of Things, it was goal of this thesis to invent a denial of service attack using the MQTT protocol to attack a MQTT server. This goal could not be reached. Although, the author tried to find a denial of service attack, no attack could be developed. One concept for an attack was implemented and tested on four different MQTT servers.

The MQTT is a publish/subscribe messaging transport. Every message has a topic. A client can ‘publish’ messages to a server. The server publishes this messages to all other clients interested in its topic. The protocol describes a retain flag. If this flag is setted on a message, the server has to retain the message until another message with the same topic arrives. If no message with the same topic arrives, it must be saved forever. One message can carry 256 megabytes payload, which need to be cached by the server. Therefore, the author had the idea that a relatively small amount of messages (with the retain flag set) would be enough to fill up all available RAM storage of the server. A prototype of this attack was implemented and tested on Mosquitto [11], Moquette [9], emqtt [5] and Mosca [10] MQTT servers. When the attack was started, the RAM of the servers filled up, but even before all messages were completely send, it became free again. Some research showed that all servers use databases to cache retain messages. Therefore, the attack is really hard to successfully perform since the whole hard disk needs to be filled and this is difficult to achieve without using distributed denial of service at least. The attack would be easily detected by an intrusion detection system because most MQTT messages are much smaller than 256 megabytes. Thus, the caused network traffic by this attack looks highly suspicious.
Chapter 4

Denial of Service Attack Detection

The detection should prove that the Code Property Graph is a powerful static analysis tool to distinguish denial of service attacks from a big pool of legitimate NodeRED flows. Furthermore some similarity should be discovered in the detection of SYN flood, Slowloris and Slow Post because common ways to detect different attacks will promote code generalisation and generalised code can be reused later to detect more attacks.

The creation of sockets, writing on sockets and a trigger, to call the write statement repeatedly, can be found in all attacks discussed in this work. Therefore, a data structure containing information about occurrences of these code structures builds the basis for detection. This code structure is named Attack Complex by the author. It is explained in section 4.1.

The result of the detection should be reported in a user friendly way and help him concentrate on reviewing suspicious programs with a high probability of containing a denial of service attack. Therefore the analysed programs will be divided into harmless flows and flows which might implement an attack. In addition the second category will be rated on a score from one to ten with regard to the suspected probability of being an attack. Section 4.4 explains the reporting in detail.

4.1 Attack Complexes

As noted above creation of sockets, writing on sockets and triggers executing a write are common in all denial of service attacks of this thesis. Therefore, locating them and their connections in flows is a good starting point for denial of service detection. Furthermore these structures are not only common to the attacks; they also need to occur in the source code of an attack which makes them invariants. Speaking of a connected creation and write means that the write is maybe called on the created socket object; a trigger is connected to a write, if the write is executed, when the trigger fires. One or more creations and triggers connected to the same set of writes form an Attack Complex. One creation, write or trigger could be contained in more than one Attack Complex and a flow can contain multiple Attack Complexes.
If at least one Attack Complex is found in a flow, it is rated suspicious and additional detection takes place. Further detection benefits from the information gathered about the attack structure in the Attack Complex data structure.

The process of pinpointing Attack Complexes in a flow is explained using the example of a Slowloris attack (listing 3.1). A partial Code Property Graph of this listing is shown in figure 4.1.

The first section of this chapter explains how Gremlin queries work because Gremlin queries are used later to traverse the Code Property Graph and locate Attack Complexes. This part uses the same Code Property Graph for its examples.

### 4.1.1 Gremlin Queries

A Gremlin query starts on a set of input vertices and subsequently performs operations on them called steps. These steps work like maps from map/reduce: They perform the same operation on every input vertex independently. To put it in other words, they take the first input vertex and perform their operation; then, they take the second and perform the same operation and so on. Per vertex a step might produce zero or more vertices as result of its operation. Every step takes the output from its predecessor as input. A simple step is the `out` step. It follows all outgoing edges from its input. Gremlin queries are also called pipelines or traversals.

In the sample Code Property Graph the query `for.out` starting at the for loop vertex (down, middle) would produce `expr_result` from line 11 and `var` from line 2. This step produced a two vertices output from a one vertex input. The query `for.out.out` would produce `call` and `while`. In particular, the query `for.out` was explained before and the last `out` step takes the output from the first `out` (`expr_result` and `var`) and follows the out edges again producing `call` and `while`. The `out` step has one optional parameter called `label` to limit the edges followed, e.g. `for.out("CFG_UNCOND")`.

Figure 4.1: Simplified Code Property Graph of the listing 3.1.
has(propertyName, propertyValue) is another Gremlin step. It filters out all elements which do not have the property propertyName set to the correct value. So the traversal for.out.has("SV", "name").out outputs only the while vertex (all states are shown in 4.1). Some other common Gremlin steps are introduced below.

<table>
<thead>
<tr>
<th>Step</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>for (10)</td>
</tr>
<tr>
<td>out</td>
<td>expr_result (11), var (2)</td>
</tr>
<tr>
<td>has</td>
<td>var (2)</td>
</tr>
<tr>
<td>out</td>
<td>while (3)</td>
</tr>
</tbody>
</table>

Table 4.1: States of the pipeline for.out.has("SV", "name").out used on the graph 4.1. The table shows the output vertices after the step named on the left.

**Common Gremlin Steps**

- **out[label]**: Starts at a vertex returns all vertices that can be reached directly via outgoing edges labelled label.
- **in[label]**: Like out but following incoming edges.
- **both[label]**: Follows edges in both directions.
- **has(name, [value])**: Filters vertices with the property name with correct value.
- **dedup**: Removes duplicates from the pipeline. Often used to avoid infinity loops.
- **retain(list)**: Retains only objects which are contained in list.

**Custom steps**

- **walkAstFromStatement**: Traverses all AST children of the statement.
- **getNodeRedNode**: Starts at a statement and returns the containing NodeRED node vertex.
- **getOnInputHandler**: Starts at a NodeRED node vertex and returns the input handler.

**Loop pattern** The loop step loop(step:string, loopFunction:object−→boolean, outputFunction:object−→boolean) steps back to the step named step and starts the query from there with all output vertices for which the function loopFunction returned true. It returns all object generated in any iteration for which outputFunction was true. E.g. var.as("x").out("CFG_UNCOND", "CFG_TRUE", "CFG_FALSE").dedup.loop("x", o−→true, o−→true) returns all vertices reachable via control flow from the upper left var vertex in graph 4.1. States shown on the next page.
<table>
<thead>
<tr>
<th>Step</th>
<th>Output</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>var (2)</td>
<td></td>
</tr>
<tr>
<td>as</td>
<td>var (2)</td>
<td></td>
</tr>
<tr>
<td>iteration 1</td>
<td>var (2)</td>
<td>Output for first iteration</td>
</tr>
<tr>
<td>iteration 2</td>
<td>while (3)</td>
<td>Input for second iteration</td>
</tr>
<tr>
<td>iteration 3</td>
<td>var (4), expr_result (9)</td>
<td></td>
</tr>
<tr>
<td>iteration 4</td>
<td>expr_result (5)</td>
<td></td>
</tr>
<tr>
<td>iteration 5</td>
<td>expr_result (6)</td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>var (4), expr_result (9)</td>
<td></td>
</tr>
<tr>
<td>dedup</td>
<td>var (4), expr_result (9)</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>var (4), expr_result (9)</td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>expr_result (5)</td>
<td></td>
</tr>
<tr>
<td>dedup</td>
<td>expr_result (5)</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>expr_result (5)</td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>expr_result (6)</td>
<td></td>
</tr>
<tr>
<td>dedup</td>
<td>expr_result (6)</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>expr_result (6)</td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>while (3)</td>
<td></td>
</tr>
<tr>
<td>dedup</td>
<td>∅</td>
<td>while is duplicate, pipeline is empty and ends</td>
</tr>
</tbody>
</table>

**Complete loop output** while, 2 * var, 3 * expr_result

The states of the pipeline

1. var.as("x").out("CFG_UNCOND", "CFG_TRUE", "CFG_FALSE").
2. loop("x", o -> true if AST_TYPE != expr_result,
3. o -> true only for AST_TYPE == var)
4.

are shown below. This loop stops at expr_result and returns only var statements. To put it in other words, the loop does not use expr_result as input for further iterations and only var statements are contained in the output. Thus, the output after the first iteration is empty.
<table>
<thead>
<tr>
<th>Step</th>
<th>Output</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>var (2)</td>
<td></td>
</tr>
<tr>
<td>as</td>
<td>var (2)</td>
<td></td>
</tr>
<tr>
<td>iteration 1 (input)</td>
<td>var (2)</td>
<td>iteration 1 empty output because only vars are output</td>
</tr>
<tr>
<td>out</td>
<td>while (3)</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>iteration 2 (input)</td>
<td>while (3)</td>
<td>iteration 2 empty output because only vars are output</td>
</tr>
<tr>
<td>out</td>
<td>var (4), expr_result (9)</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>var (4)</td>
<td></td>
</tr>
<tr>
<td>iteration 3 (input)</td>
<td>var (4)</td>
<td>iteration 3 does not continue on expr_results</td>
</tr>
<tr>
<td>out</td>
<td>expr_result (5)</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>iteration 4 (input)</td>
<td>∅</td>
<td>iteration 4 empty input, pipeline ends</td>
</tr>
</tbody>
</table>

**Backtrack pattern** The step \texttt{back(step)} steps back to the step \texttt{step}. It can be used to validate if a statement calls an external function and if so step back to it. E.g. in the graph 4.1 the pipeline

```
1 start.as("x").out("CALLS").has("QUALIFIED_NAME", "net.Socket").
2 back("x").in("AST_PARENT_OF")
```

will return \texttt{name SV=s} if \texttt{start} is the call to \texttt{net.Socket} (line 4), but if \texttt{start} is the call to \texttt{setInterval} (line 9) the output will be empty. States are shown below:

<table>
<thead>
<tr>
<th>Step</th>
<th>Output</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>call (line 4)</td>
<td></td>
</tr>
<tr>
<td>as(&quot;x&quot;)</td>
<td>call (4)</td>
<td></td>
</tr>
<tr>
<td>out(&quot;CALLS&quot;)</td>
<td>name</td>
<td>net.Socket</td>
</tr>
<tr>
<td>has(...)</td>
<td>name</td>
<td>net.Socket</td>
</tr>
<tr>
<td>back(&quot;x&quot;)</td>
<td>call (4)</td>
<td></td>
</tr>
<tr>
<td>in(&quot;AST_PARENT_OF&quot;)</td>
<td>name (4)</td>
<td></td>
</tr>
<tr>
<td>start</td>
<td>call (9)</td>
<td></td>
</tr>
<tr>
<td>as(&quot;x&quot;)</td>
<td>call (9)</td>
<td></td>
</tr>
<tr>
<td>out(&quot;CALLS&quot;)</td>
<td>name (setInterval)</td>
<td></td>
</tr>
<tr>
<td>has(...)</td>
<td>∅</td>
<td>pipeline is empty and ends</td>
</tr>
</tbody>
</table>
The back step cannot be replaced by the step \texttt{in("CALLS")}. This step looks like the inversion of \texttt{out("CALLS")}, but it will return all callers of the \texttt{net.Socket}; so, it might introduce many new vertices to the pipeline. The back step will never output vertices that have not been visited in the traversal before. So, the back step allows backtracking in Gremlin and its use is often called backtrack pattern.

### 4.1.2 Locating Attack Complexes on Slowloris

The attacker will be allowed to upload his own nodes and can use function nodes to execute nearly arbitrary JavaScript code. This gives a lot of control to the attacker and puts the detector in a bad position. Therefore invariant code structures must be found which cannot be changed by the attacker. In this section the pseudo code of a Slowloris attack is used as an example (see 3.1). Slow Post is very familiar to Slowloris; thus, it can be detected in the same way as described here.

A TCP socket is needed to send incomplete HTTP requests to the victim; therefore, the creation of a socket is an invariant code structure. In the example the socket is created at line 4. The call to the \texttt{write} function of the socket (line 11) is another invariant statement which is necessary to send a HTTP request. The call at line 5 is not part of the Attack Complex, because it is not reached by the trigger from line 9. To add new headers to the request frequently a \texttt{setTimeout, setInterval} or a similar function is needed to trigger the call to \texttt{write} multiple times.

**Pinpointing Code Invariants** The three invariant statements can be found by scanning the call graph for calls to \texttt{net.Socket, net.connect, setTimeout} or \texttt{net.Socket.prototype.write}. The scan for the external function \texttt{f} can be implemented with the gremlin query from listing 4.1.

```java
1 g.V.has("IS_EXTERN", true).has("QUALIFIED_NAME", nameOfF).in(CALLS)
2
```

Listing 4.1: Gremlin query to find calls to the external function with the name \texttt{nameOfF}.

The traversal starts on all vertices of the Code Property Graph; then it filters for external vertices with a qualified name of \texttt{f}. After the filter only the vertex representing the function scanned for is left in the pipeline. From this vertex the incoming edges of type call are followed to the callers.

The concrete states of the pipeline for our example and \texttt{nameOfF = net.Socket.prototype.write} is shown in table 4.2.

During the detection this query runs for every external function for which calls should be found. In case of Slowloris it is executed five times for the functions \texttt{net.Socket, net.connect, setTimeout, setInterval} and \texttt{net.Socket.prototype.write}. In the example (3.1) the calls at line 4, 5, 9 and 11 will be located. After these vertices have been identified; it must be confirmed that they are connected.
Linking Socket Creations to Socket Writes To find a link between socket creation and socket write data dependence edges are followed. If a call to write uses a socket, their vertices are linked by data dependence edges labelled USES. A traversal from a socket creation to a socket write can be formulated in gremlin as shown in listing 4.2.

```
1 socketCreationVertex.as("loop").both("USES").dedup.
2  loop("loop", true, true).retain(foundSocketWrites)
```

Listing 4.2: Traversal to connect socket creations and writes.

The traversal starts at the socket creation vertex and repeatedly follows USES edges. At the end only calls to write are retained; these calls can be found using the call graph as explained before (section 4.1.2). Socket creations are located the same way.

This traversal walks over all USES edges in the Code Property Graph 4.1 to traverse from the socket creation (var (4)) to the write (expr_result (11)). The pipeline is shown in table 4.3.

<table>
<thead>
<tr>
<th>Step</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>g.V has(&quot;IS_EXTERN&quot;, true) has(QUALIFIED...) in(&quot;CALLS&quot;)</td>
<td>All vertices The four external vertices The net.Socket.prototype.write external vertex Two call vertices from line 5 and 11</td>
</tr>
</tbody>
</table>

Table 4.2: The pipeline states for cpg.V.has("IS_EXTERN", true).has("QUALIFIED_NAME", "net.Socket.prototype.write").in("CALLS") for the Code Property Graph 4.1.
<table>
<thead>
<tr>
<th>Step</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>var (4)</td>
</tr>
<tr>
<td>as</td>
<td>var (4)</td>
</tr>
<tr>
<td>iteration 1 (input)</td>
<td>var (4)</td>
</tr>
<tr>
<td>both(&quot;USES&quot;)</td>
<td>expr_result (6)</td>
</tr>
<tr>
<td>loop</td>
<td>expr_result (6)</td>
</tr>
<tr>
<td>iteration 2 (input)</td>
<td>expr_result (6)</td>
</tr>
<tr>
<td>both(&quot;USES&quot;)</td>
<td>var (2)</td>
</tr>
<tr>
<td>loop</td>
<td>var (2)</td>
</tr>
<tr>
<td>iteration 3 (input)</td>
<td>var (2)</td>
</tr>
<tr>
<td>both(&quot;USES&quot;)</td>
<td>for (10)</td>
</tr>
<tr>
<td>loop</td>
<td>for (10)</td>
</tr>
<tr>
<td>iteration 4 (input)</td>
<td>for (10)</td>
</tr>
<tr>
<td>both(&quot;USES&quot;)</td>
<td>expr_result (11)</td>
</tr>
<tr>
<td>loop</td>
<td>expr_result (11)</td>
</tr>
<tr>
<td>iteration 5 (input)</td>
<td>expr_result (11)</td>
</tr>
<tr>
<td>both(&quot;USES&quot;)</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Complete loop output</td>
<td>var (4), expr_result (6), var (2), for (10), expr_result (11)</td>
</tr>
<tr>
<td>retain(list of write statements)</td>
<td>expr_result (11)</td>
</tr>
</tbody>
</table>

Table 4.3: The states of the traversal 4.2 on the graph 4.1 from the socket creation at line 4 to the socket write at line 11.
Explaining Limitation and Improvements of Dataflow Traversals

The traversal connecting socket creations and socket writes is limited to the case that socket creation and socket write happen in the same node; that is, because all native NodeRED nodes and all custom nodes used in this work fulfil this limitation. The author knows two ways to communicate between distinct nodes in NodeRED; that is, via calls to `RED.Node.prototype.send` or via the `context.global` object in `function` nodes. Both cases could be covered without losing much precision by modifying the NodeRED flow database generation process and naturally integrating the global context and messages in the existing data flow model.

In order to cover the `context.global` object a vertex for the global object could be introduced and connected with incoming USES edges to all usages of the global object in functions. The vertex for `context.global` would represent the actual definition of the global context object in the `function` node source code; in other words, this would be a natural extension of the database. Graph 4.2 illustrates the database extension.

Figure 4.2: Simplified Code Property Graph for two `function` nodes communicating the value Hello David via the `context.global` object. The global context object vertices in dark red or purple are database extensions. The upper, dark red extension is based on a field sensitive data flow analysis. The purple solution below is field insensitive. The source code of the function is a property of the function nodes vertices.

Inter node communication via messaging could be handled by adding a USES edges from the definition of the message object in the sending node to the input handler of the receiving node. The relationships between sending and receiving nodes are static and they can be read from the JSON file of the flow. To locate the message object definition in the sending node a traversal from the first argument of the call to `RED.Node.send` to its definition could be designed or the name of the message object could be restricted (e.g. to `msg`) and found by it. Graph 4.3 illustrates the
The traversal connecting the argument of `send` to its definition in this example is trivial since it has to follow only one data flow edge, but this traversal becomes more complex if an object is used as message or if the attacker obfuscates his code. In most real NodeRED nodes an object is used as message. Therefore, no solution for this traversal is explained here. The `SENS_TO` edge already represents the relationship between the sending and receiving node and will help to design an algorithm to add data flow relations between wired nodes.

Figure 4.3: Simplified Code Property Graph for two NodeRED nodes. A message object definition and a call to `RED.Node.prototype.send` is modelled on the left. The input handler from a node wired is shown on the right. The dashed data flow edge is the database extension explained in the paragraphs before.

Both suggested solutions add `USES` edges for object definitions; in consequence, the approaches could gain precision if the data flow analysis used was field sensitive. But field sensitivity is not offered by the Closure Compiler. Therefore, it was not implemented in this thesis. Graph 4.2 compares a database generated by field sensitive or field insensitive data flow analysis.

The traversal connecting socket creations with socket writes is quite imprecise. In following example 4.3 a write to a buffer will be mistaken as a write to a socket. Two reasons for this will be explained and solutions suggested. The example is not taken from any existing source code, but it would be a running JavaScript program and could be used in one NodeRED node.
function ImpreciseExample(message) {
    var buffer = new Buffer();
    var socket = new Socket(message.address);

    buffer.write(message.payload); // Will be recognized as a write on a socket.
    socket.write("Something");
}

Listing 4.3: This listing demonstrates the impreciseness of the used data flow traversal. A write to a buffer is mistaken for a write to a socket.

Figure 4.4: Simplified Code Property Graph for listing 4.3. The dashed CALLS edges would not exist with a more precise call graph generation.

In listing 4.3 the buffer.write is detected as a write to a socket, although it is not. The listing contains also a real socket write. Most discussion ahead deals only with the buffer.write. However, the socket.write is included to show a correctly detected write and enable the reader to compare the situations. The buffer.write is mistaken for a socket write because the call graph offered by the Closure Compiler does not use type inference and since the Closure Compiler uses a very simple approach to deal with properties of objects; ‘[it] treats all objects as capable of having the same set of properties’ (cited from the JavaDocs of SimpleDefinitionFinder used for call graph construction [4]). Therefore the call from line 5 will be marked as a call to buffer.Buffer.prototype.write and net.Socket.prototype.write since no more detailed information are available. As a consequence the gremlin query to locate calls to net.Socket.prototype.write (4.1) will return also the call in line 5. Later during the detection process it will be connected to the socket creation because the connecting traversal is imprecise. This matter is addressed latter in this section. The problem concerning the call graph generation could be fixed by implementing a more precise call graph generation.
Using the type inference offered by the Closure Compiler is an approach to refine the call graph generation. The classes offered by the Closure Compiler are TypeInferencePass and TypeInference. With the knowledge that the buffer variable is of type Buffer, the target of the write call in line 5 could be unambiguously determined.

The Closure Compiler provides type inference and call graph generation, but is not combining information from both sources. Therefore, some work will be necessary to retrieve type information and combine it to the call graph information already available in CallGraphGenerator. Specifically, the combination of both information sources could be a non-trivial task.

The imprecise call graph generation is a problem with the fundamentals provided by the Closure Compiler. The second reason for which the call to buffer.write is mistaken for a socket write arises from the used data flow query 4.2 and as such can be solved without changing the graph generation. The data flow query 4.2 follows all incoming and outgoing USED edges. In the example it starts at the socket creation (line 3). This statement uses the message object; so, the outgoing USES edge to the parameter message (line 2) is followed and from there it walks along the data dependence edge to line 5; this line also uses the message object. Finally the socket creation and the buffer write are connected by this traversal, not because the write happens on the socket created, but because the same variable is used in both statements.

This behaviour could be improved by restricting the data flow query to USES edges with the correct VARIABLE_NAME property; this is a property on USES edges. In this example only data flow edges from the socket creation with the variable name socket should be followed.

The implementation of Slowloris 4.4 will serve as an example to illustrate which names should be followed for each statement. The same source code was already used as an example before and its Code Property Graph is presented in graph 4.1.

In order to restrict the data flow query to the correct variable names, it will get more complex because it needs to be aware of its context and path taken. E.g. in line 6 it needs to recognize the call to push and in line 2 it has to infer that the defined array might contain the formerly pushed socket. This context sensitivity needs to be implemented on a case-by-case basis and might limit the query; so it will not perceive connections which were detected before. In other words the improvement trades generality for precision. However, the context sensitivity has also an advantage. For now the query follows USES edges in both directions; as stated above this is needed to handle cases as such a socket in an array, but with context sensitivity the traversal could be restricted to the incoming or outgoing direction depending on the context.

Both solutions could be implemented independently and both could solve the problem without the other; to put in other words, if one of them is implemented only one socket write will be detected in listing 4.3 which is correct. The first solution making the call graph more precise would produce a database without the dashed call edges from graph 4.4. Therefore, the buffer.write at line 5 would not be identified as a call to net.Socket.prototype.write; thus, it would not be a valid end point for the data flow traversal (4.2). The second solution following only the USES edge
with correct names would only follow the one USES edge with the name `socket` from the socket creation at line 3. Hence, the socket creation and the `buffer.write` will not be connected; even though, the `buffer.write` would be recognized as a call to `net.Socket.prototype.write`.

```javascript
function SlowLoris(msg) {
    var a = []; // 3 follow 'a'
    while(victimHasFreeConnections()) {
        var s = new net.Socket(); //Start here. Definition -> follow 's'
        s.write('POST p.html HTTP/1.1\n');
        a.push(s); //Followed 's' to here, push to array -> now follow 'a'
    }

    setInterval(function sendHeader() {
        for (s in a) { //Followed 'a' -> array unpacking -> follow 's'.
            s.write('x-header: value\n'); // End here.
        }, 1000);
    }
}
```

Listing 4.4: Demonstration which variable use should be followed. The comments explain which name to follow and give a hint why.

**Connecting Triggers to Socket writes** A `setTimeout` or `setInterval` triggers a `write`, if the `write` is reached by the control flow of the time-out handler. All control flow reached statements are found with this Gremlin query 4.5.

```javascript
cfgEntryNodeOfHandler.as("loop").out("CFG_UNCOND", "CFG_TRUE",
   "CFG_FALSE", "CFG_EX").loop("x", true, true)
```

Listing 4.5: Gremlin query to follow control flow

In the listing 3.1 of a Slowloris attack the traversal starts at the handler of the `setInterval` in line 9 which is the second syntax tree child of the located trigger statement. Then it follows two `CFG_UNCOND` edges to `for (var s in a)` (line 10) and from there to `s.write()` (line 11). After that it reaches the `CFG_RETURN` node of the handler and stops since the return node has no outgoing control flow edges.

NodeRED massively uses event handling to propagate messages from node to node. This also carries the control flow from one node to another; therefore, the NodeRED specific event handlers `input` and `send` need to be taken into account to find connections between `writes` and triggers on distinct nodes. A flow with an `inject` node connected to a `tcp out` node should illustrate the use of both events to trigger a `write` with a `setInterval`. Listing 4.6 shows the important parts of code from the `inject` and `tcp out` node. The control flow in this example starts in line 4 in the handler of `setInterval` and triggers the event `input` on the `inject` node in line 5; then the event handler of `input` in line 7 is executed and `this.send` is called.

54
in line 8. This call will trigger the input event on all connected nodes; so the input event handler of the tcp out node in line 15 and the call to write (line 16) will be executed.

To implement the traversal from the trigger in the inject node to the write in the tcp out node as in the example above, three different Gremlin queries are used. One of these gremlin queries is known from the first example and defined in the listing 4.5; it follows the control flow edges. Both other traversals start at a statement and end at control flow entries for triggered input or send handlers. These two queries are defined and explained; then the algorithm using these three queries is described.

The traversal 4.7 checks a statement for calls to \texttt{events.EventEmitter.prototype.emit("input")} triggering the input event on a NodeRED node and traverses to the control flow entry of the input handler of the same node.

```javascript
1 statement.walkAstFromStatement.as("x").out("CALLS").
2 has("IS_EXTERN", true).
3 has("QUALIFIED_NAME", "events.EventEmitter.prototype.emit").
4 back("x").getSecondAstChild.has("STRING_VALUE", "input").
5 getNodeREDNode.getOnInputHandler
6
Listing 4.7: Traversal to scan a statement for triggers of the onInput handler of a NodeRED node.
```

It walks the sub syntax tree of the starting statement with \texttt{walkAstFromStatement} and names this step \texttt{x} to use the backtrack pattern later. If one of the expressions calls \texttt{events.EventEmitter.prototype.emit} with the string "input" as first argument, the traversal walks to the NodeRED node vertex belonging to the starting statement and from there with the step \texttt{getOnInputHandler} to the control flow entry of the input handler of this node.

In the graph 4.5 the traversal 4.7 starts at the \texttt{expr_result} from line 5. This starting vertex is chosen because this is the vertex that query will produce results on later in the discussion. The output of all steps is shown in table 4.4.

The third used traversal 4.8 handles the NodeRED specific event of sending messages from one node to another. If \texttt{RED.Node.send} is called, the input handler of all connected nodes will be triggered. This control flow must be followed to connect triggers like setTimeout to writes.

```javascript
1 statement.walkAstFromStatement.out("SENDS_TO")
2
Listing 4.8: Traversal to scan a statement for triggers of the onInput handler of a NodeRED node.
```

The traversal scans its starting statement for calls to \texttt{RED.Node.send} and returns control flow entries of input handlers triggered. Calls to \texttt{RED.Node.send} are linked to the input handler of other nodes activated by them with \texttt{SENDS_TO} edges during
Listing 4.6: Use of input and send event to trigger a write. The source code is taken from the NodeRED nodes.

```javascript
// Inject Node
function InjectNode() {
    RED.nodes.createNode(this, n);
    setInterval(function() {
        node.emit("input", {});
    }, this.repeat);
    this.on("input", function onInput(msg) {
        this.send(msg);
    });
}

// TCP Node
function TCPOutNode() {
    RED.nodes.createNode(this, n);
    this.on("input", function onInput(msg) {
        socket.write(msg);
    });
}
```

Figure 4.5: Simplified Code Property Graph for listing 4.6. Data dependence and control dependence are completely omitted to avoid cluttering the graph. The two custom steps `getNodeREDNode` and `getOnInputHandler` are included as dark red edges; these are no database edges. The purple edge is NodeRED specific and is labelled with `SENDS_TO`. 
Table 4.4: State of the query 4.7 in the graph 4.5 starting on `expr_result` at line 5.

database generation. The traversal follows outgoing `SENDTS_TO` edges from expressions within the scanned statement and walks to the control flow entry of the reached `input` handlers.

The pipeline 4.8 starting at `expr_result` (line 8) in the graph 4.5 is illustrated below:

<table>
<thead>
<tr>
<th>Step</th>
<th>Output</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td><code>expr_result</code> (8)</td>
<td></td>
</tr>
<tr>
<td>walkAstFromStatement</td>
<td><code>call, getProp, this, 2 * name</code></td>
<td></td>
</tr>
<tr>
<td>out(&quot;SENDTS_TO&quot;)</td>
<td><code>function (15)</code></td>
<td></td>
</tr>
</tbody>
</table>

The three traversals `controlFlowReached` (listing 4.5), `onInputReached` (listing 4.7) and `onSendReached` (listing 4.8) are combined in a worklist based algorithm to find connections between triggers and `writes` even if they are in different nodes. The algorithm is presented in 4.9.

The worklist is initialised with the control flow entry handler from the trigger. For all statements from the worklist every statement reachable with control flow edges is added to the set `cfgReached`; after all control flow reached statements are scanned for triggering the `input` or `send` event of NodeRED with the traversals `onInputReached` (4.7 and `onSendReached` (4.8). As stated above, this queries return the control flow entry nodes of eventually triggered handlers. These control flow entries are added to the `onInputOrSendReached` set. All reached statements are collected in `allReached`. For all new vertices that were discovered after an iteration, the loop stops. If no new vertices were discovered after an iteration, the loop stops and all statements reached by the trigger are intersected with all found calls to `write` in the program; this intersection equals all `writes` which could be executed by the
trigger. During detection this algorithm is computed once per located trigger.

On the graph 4.5 three while-loop iterations are necessary. The contents of the different lists and outputs of the traversals are shown per iteration in table 4.5.

```java
// @param handlerCfgEntry The control flow graph entry of the handler of the trigger.
// @param callsToWrite All previous found calls to write in the program.
void findAllReachedWrites(Vertex handlerCfgEntry, Set<Vertex> callsToWrite) {
    Set allReachedStatements = new Set();

    Set worklist = new Set(triggeredCFGEntry);
    while (!worklist.isEmpty) {
        Set cfgReached = new Set();
        for (Vertex v : worklist) {
            cfgReached.add(v.controlFlowReached());
        }
        Set onInputOrSendReached = new Set();
        for (Vertex v : cfgReached) {
            onInputOrSendReached.addAll(v.onInputReached());
            onInputOrSendReached.addAll(v.onSendReached());
        }
        allReached.addAll(cfgReached);
        allReached.addAll(onInputOrSendReached);
        worklist.addAll(onInputOrSendReached);
    }
    return callsToWrite.retainAll(allReachedStatement); // Intersect writes and reached statements.
}
```

Listing 4.9: Algorithm to connect a trigger to a subset of all found sockets write of the program.
<table>
<thead>
<tr>
<th>Traversal / Set</th>
<th>Output / Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>worklist</td>
<td>function (4) (trigger handler)</td>
</tr>
</tbody>
</table>

**iteration 1**
- `cfgReached`: `expr_result (5)`
- `onInputReached`: `function (7)`
- `onSendReached`: `∅`
- `allReached`: `expr_result (5), function (7)`
- `worklist`: `function (7)`

**iteration 2**
- `cfgReached`: `expr_result (8)`
- `onInputReached`: `∅`
- `onSendReached`: `function (15)`
- `allReached`: `expr_result (5), function (7), expr_result (8), function (15)`
- `worklist`: `function (15)`

**iteration 3**
- `cfgReached`: `expr_result (16)`
- `onInputReached`: `∅`
- `onSendReached`: `∅`
- `allReached`: `expr_result (5), function (7), expr_result (8), function (15), expr_result (16)`
- `worklist`: `∅`

Intersection with writes: `expr_result (16)`

Table 4.5: This table shows the content or output from sets or traversals per iteration from the algorithm 4.9 on graph 4.1 starting at the located trigger. This is not a table illustrating a single Gremlin traversal.

**Improving Trigger Detection** This section will explain how to implement two concrete improvements for the trigger detection.

For now all usages of `setTimeout` and `setInterval` are treated as triggers even though they will have a very long time-out or a low frequency respectively. But a trigger for a Slowloris attack should have a time-out no longer than 5 minutes and for SYN flood the time-out probably should be shorter than one second. The time-out or interval of both functions is the second argument and as such can be determined easily, if the value is given as a literal. The value is not given as a literal in the native NodeRED `inject` node; however, it is written as a literal in the configuration in this case and can be read from the `config` vertex of this node.

In general it is a hard problem to detect the value of a time-out or `setInterval` if is variable. Points-to analysis as discussed later in section 4.3 could help here.

When the time-out or interval value is known only triggers with a low enough value could be taken into account to build Attack Complexes.

In this work the name of the `input` handler of nodes is restricted to `onInput` and the
setTimeout or setInterval calls are required to get their handler directly defined in their first argument (it is not possible to hand in a function pointer); so the handlers can be found easily. This is a limitation of the thesis itself. In addition it prevents another enhancement to refine the detection by filtering the triggers. This improvement will be explained in the next paragraph.

Currently every timeout is accepted as a trigger in the detection process, but a timeout needs to set itself again or else its handler will be executed only once. A setTimeout that sets itself again is demonstrated in listing 4.10. Line 3 sets a timeout called in its own handler handler and in line 5 the handler is called directly.

In general, a timeout will set itself again, if it lies within the execution path of its handler. In all implemented attacks this is done by setting the time-out inside of its handler and calling the handler directly once.

```javascript
1 function handler() {
2   doSomething();
3   setTimeout(handler, 1000);
4 }
5 handler();
```

Listing 4.10: Example of a setTimeout that sets itself again.

Such a timeout can be detected by getting the function it is set in and compare it to the handler. If they are equal, the timeout sets itself again. The function the time-out is set within or in other words called in can be determined with the help of the Gremlin query `setTimeoutStatement.walkUpAst(AstType.Function)`. The function of interest is the first result of this traversal. In graph 4.6 this traversal is only one edge long (from the `expr_result` to the `function` vertex). The handler can be identified by following the USES edges from the time-out. No traversal to do this is presented here because it is easy in the example, but hard to find a sound traversal in general since an attacker could easily hide handler by putting it in an object (`setInterval(obj.x.y, 100)` or use multiple aliases. Field sensitive Points-to analysis could help to soundly find the handler in such cases (see section 4.3).
4.1.3 Putting the Parts Together: Building Attack Complexes

After all invariants have been found and socket creation and triggers are connected to writes, they need to be connected to Attack Complexes. Every socket creation and trigger can be connected to multiple writes. E.g. a trigger has multiple writes if it reaches more than one. Every write can be connected to multiple socket creations or triggers. The combination from one socket creation and one or more writes will be called creation part. A trigger and the writes connected to it will be called trigger part. A creation part and a trigger part could form an Attack Complex if they are connected to the same write statements. Therefore, several ways to construct Attack Complexes are possible. A creation part and a trigger part could form an Attack Complex if their sets of writes are equal or they could form several Attack Complexes for every subset of equal writes. To keep it simple and to cover every possible Attack Complex the author decided to build all combinations possible. Every socket creation and trigger connected to a single same write build an Attack Complex. To put it a different way, socket creations and triggers are joined if they are connected to at least one equal write. The Slow Loris attack from listing 3.1 contains one Attack Complex. As a more complex example imagine a program with two socket creation $s_1, s_2$, two triggers $t_1, t_2$ and two writes $w_1, w_2$. $s_1$ is connected to $w_1$ and $w_2$. No write belongs to $s_2$. $t_1$ and $w_1$ are connected. $t_2$ reaches $w_1$ and $w_2$. The program contains three Attack Complexes: $\{s_1, w_1, t_1\}, \{s_1, w_1, t_2\}, \{s_1, w_2, t_2\}$.

4.1.4 Applying Attack Complexes to SYN flood Attacks

As for Slowloris the socket creation, socket writes and trigger are invariants of a SYN flood attack all of them are needed to send SYN packages to the victim. The invariants can be found in lines 7, 13 and 12 of the listing 3.2. They can be connected with the same traversals as explained above for Slowloris. The socket write is data dependent on the socket creation and the write is control flow reachable from the handler of the trigger.

One difference in constructing an Attack Complex for a SYN flood attack is the names of external functions to seek for. The externals used for a SYN flood are from the package rawsocket, because a raw socket is needed instead of a TCP socket. The names of these functions are rawSocket.createSocket and rawSocket.Socket.prototype.send. To locate the triggers no changes are necessary at all.

Another difference is that no timeout or setInterval is necessary to perform a SYN flood attack because the SYN packages can be sent as fast as possible. Therefore the program execution does not have to be delayed as for a Slowloris or Slow Post attack. This allows the attacker to use another trigger. The function send of a raw socket offers to call a function after a package is completely transmitted. This function can be used to send another SYN package. Source code of a SYN flood attack using this trigger is shown in listing 4.11. This code is also used in the single node implementation of SYN flood. A sound SYN flood detection should detect for this kind of triggers as well. The Attack Complex approach is flexible enough to be easily extended for this new situation. As explained in section 4.1.2 triggers are
var rawSocket = require('raw-socket');
var socket = rawSocket.createSocket(options);
var packet = generateTcpSynPacket();

function sendPacket() {
  socket.send(packet, sendPacket); // Second parameter is a function
called after the package has been sent.
}

Listing 4.11: This listing demonstrates the use of the `afterSend` callback from `rawSocket` as trigger for SYN flood.

located by finding calls to external functions in the Code Property Graph. This trigger can be found by scanning for `rawSocket.Socket.prototype.send`. After a trigger is pinpointed in the graph, a traversal following control flow and some NodeRED specific triggers is used to connect triggers with located `writes` (see 4.1.2). This traversal starts at the entry point of the triggered function and identifies all statements that will be executed by the trigger. If one of these statements is write to a socket, trigger and write are connected and could be part of an Attack Complex. This traversal can be used without any changes to connect the new trigger to writes. It starts at the entry point of the callback (line 5 in the example) and follows the control flow to the socket write at line 6. The simple integration of this new kind of trigger in the Attack Complex approach show cases its flexibility.

4.1.5 Conclusion

The Attack Complex data structure is a central result of this work. Since a program with an Attack Complex might perform a denial of service attack, it is reported as suspicious with the lowest score. Based on the Attack Complex found or other program properties further detection is applied to filter out false positives. Two kinds of further detection were implemented to make the detection of Slowloris and Slow Post more precise and are explained in section 4.2. Additional ideas for further detection and Attack Complex building refinement are explained above. This proves that the Attack Complex can be used for more precise detection, but also that the traversals used to build the Attack Complex will need some refinements before used in practice. In addition some fundamental limitations of this work proved to be an obstacle for more specific results. Especially field sensitive points-to analysis to handle aliasing, objects, arrays and get closer to the value of used used variables would be of great help.

The Attack Complex was discovered to be a common structure in all denial of service attacks in the scope of this work. The code to locate an Attack Complex was re-used for all attacks; therefore, the goal to identify similarities in detection for different attacks could be achieved.

To conclude the Attack Complex structure locates the cornerstones of different denial of service attacks; so, further detection can be based on it, but enhancement will be necessary to use this approach in real-world applications. Starting points and ideas
for enhancement are provided by this work.

4.2 Detection Refinements

The detection approach of this work is based on Attack Complexes. If a flow contains an Attack Complex, it is mostly marked as suspicious. However, the detection is further refined after an Attack Complex was identified. E.g. by filtering out Attack Complexes that are not actually suitable to perform an attack (as explained in section 4.2.1) or by locating code structures typical for an attack independently of the Attack Complexes like in section 4.2.2.

4.2.1 Multi Targeted Socket Writes

Slowloris and Slow Post try to block all available HTTP connections to their victim (see sattackedection explaining 3.1 those). As a result they need more than 100 simultaneously opened TCP connections to the server attacked and they need to write to all of them frequently. Therefore, references to all connections opened need to be stored. The most common way to do so is to use an array to keep the references and writing on them by iterating the array with a loop. In this case a single statement calling `write`, within the loop, will be executed for each reference; in other words, it is executed on multiple targets.

This section explains how to filter out located Attack Complexes with a socket write to only a single socket reference, which do not use an array or object to store multiple references. These Attack Complexes are false positive in terms of Slowloris and Slow Post since they may frequently write to a TCP connection, but not to hundreds of them. Hence, Attack Complexes with a single socket instance as target to their write should not mark a flow suspicious.

To demonstrate the problem and to prove it can be solved, a flow with only a single TCP socket instance was implemented. Before the refinements from this section, it was falsely detected. The flow consists out of an `inject` wired to a `tcp out` node (shown in figure 1.2). Its implementation can be found in `NodeRedDoS/FalsePositive/inject.tcpout.json`. This flow contains a detectable Attack Complex. The `tcp out` node creates a socket and writes to it in its input handler. This call to `write` is reached by the `setTimeout` and `setInterval` inside the `inject` node. This Attack Complex was recognized and the flow was marked suspicious by the first detector implementation of this work.

The flow explained above can be distinguished from a flow using an array to store different socket references by the data flow traversal between its socket creation and socket write (see section 4.1.2 for detailed description of the traversal used). For both cases the source code is shown in the listing 4.12 and the traversals from creation to write are shown in figure 4.7.

There is one elementary difference in the traversals shown in figure 4.7. The traversal for a single targeted write only uses incoming edges, but the multi-target traversal needs edges of both directions to connect creation and write. This is the case because
function TCPOutNode() {
    RED.nodes.createNode(this,n);
    var socket = new Socket();
    this.on("input", function onInput(msg) {
        socket.write(msg);
    });
}

function GreedyTCPNode() {
    RED.nodes.createNode(this,n);
    var a = [];
    while(victimHasFreeConnections()) {
        var s = new net.Socket('192.42.42.42');
        s.write('POST p.html HTTP/1.1\n');
        a.push(s);
    }
    this.on("input", function onInput(msg) {
        for (s in a) {
            s.write('x-header: value\n');
        }
    });
}

Listing 4.12: Source code from the native tcp out node with a single socket instance, below source code from the Greedy TCP node using multiple sockets.

both the creation site of the traversal (lines 16, 18) and the write site (lines 22, 23) use the array definition (line 14); so both of them have data dependence edges pointing toward the array definition.

To conclude, the traversal connecting socket creation to a single targeted write often uses edges of only one direction, but to connect a multi-targeted write to its creation sites the traversal contains edges of both directions. This is fundamentally true for all cases using an array, object or any kind of data structure to iterate over because the creation site and the write site have to use the same definition of the data structure and at this definition the edge direction followed will change. The difference described is used by the detection of this thesis to eliminate falsely positive Slowloris and Slow Post results.
4.2.2 Suspicious Strings

Slowloris and Slow Post attacks need to manually establish a HTTP connection to their victim. Therefore they often contain a string like ‘POST path.html HTTP/1.1 n’. Strings like this are not often used in legitimate NodeRED flows because NodeRED offers native HTTP nodes which can be used instead of writing the request on your own. These native nodes do not contain such strings since they use an external library. Consequently a string to start a HTTP request is suspicious and locating it leads to a higher score in the detection result (indicating higher probability of being an attack).

All string literals in a program are represented in abstract syntax tree vertices of type `STRING` and the used literal is held by the property `STRING_VALUE`. Thus, suspicious strings starting a HTTP request can be located by matching the regex `'(GET|POST)\s+\s+\s+\s+HTTP/1.[01]\n/i'` against all `STRING_VALUE` properties of `STRING` vertices of the flow. This regex matches all strings that start with ‘GET’ or ‘POST’, contain one or more white spaces, a requested path of ambiguous length, one or more white space, a HTTP version specifier for the versions 1.0 or 1.1 and
end with a new line character. The /i makes the regex case insensitive.

For the attacker it is straightforward to disguise usage of these suspicious strings. He could split them into two or more parts and concatenate them later. For simple cases, which are also the most obvious ones to use, this disguise can be uncovered with the help of the Closure Compiler. The Closure Compiler offers compiler passes to fold constants (PeepholeFoldConstants), inline variables (InlineVariables, FlowSensitiveInlineVariables) and functions (InlineFunctions). The inlining passes will inline variables and function return values that are known to be constant in every program run. The pass PeepholeFoldConstants will solve simple operators like + (string concatenation or addition) for known constant values. Some rather simple attempts to hide suspicious strings, which can be uncovered using the aforementioned compiler passes, are shown in listing 4.13.

```javascript
1 // Simple variable inlining + constant folding.
2 var append = 'POST index.html ';
3 append = append + 'HTTP/1.0\n';

4 // Flow sensitive inlining and constant folding.
5 var a = 'POST index.html ';
6 var b = a + 'HTTP/1.0\n';
7 if (false) {
8     b += '1';
9 }
10
11 var a = 'P';
12 var b = true;
13 if (b) {
14     a += 'OST index.html HTTP/1.0\n';
15 }
16
17 // Function Inlining and constant folding.
18 function constantReturn() {
19     return 'POST index.html ';
20 }
21 var a = constantReturn() + 'HTTP/1.0\n';
22
Listing 4.13: Examples of cloaked suspicious strings which can be uncovered with Closure Compiler passes. Passes necessary for each example are mentioned in comments.

This approach to uncover suspicious strings does not work for loops iterating over an array of strings and appending them (an example is shown in listing 4.14.). It does not work either on encoded strings.

To conclude, the detection process uses a regex to find strings starting an HTTP request, which are suspicious because there is no common use for them in legitimate flows. Simple methods to disguise these strings can be uncovered with the help of the Closure Compiler. However, more complex disguise methods as encoding or looping will allow the attacker to use these strings without being detected.
var array = ['P', 'O', 'S', 'T', '', 'index.html', 'HTTP/1.0\n']

var suspiciousString = ''

for (var c in array) {
    suspiciousString += c;
}

Listing 4.14: Example of disguised suspicious string which will not be detected.

4.3 Further Work to Fine-Tune Code Property Graph for Denial of Service Detection

The former chapter explains Attack Complexes. This is an general and promising approach to use a Code Property Graph to detect denial of service attacks in NodeRED flows. Limitations of this approach and next steps to improve it are presented. This section discusses further work about the Code Property Graph to make it a even more powerful basis for detection. The discussion focuses on two central points. The limitation sections of the former chapters reveal that improving the data dependence graph preciseness would be of great help. Specifically helpful would be points-to analysis. The data dependence is often discussed in the former sections; whereas control dependence is never mentioned. Control dependence was not very useful for detection until now; some reasons are explained in this paragraphs.

During the work on this thesis it became obvious that points-to analysis would be a better approach than data dependence to analyse the data flow. Data dependence focuses on definitions; whereas points to analysis concentrates on references. Some examples showcasing the difference between data dependence and points to analysis are shown in figure 4.8.

The example demonstrates two scenarios where the data flow based traversal connecting socket creations to socket writes could be simplified and more precise with a good points to analysis because only a single edge must be followed. Points to analysis as shown in the examples is possible for JavaScript and was developed in [29] or [22]. Nevertheless points to analysis is more complex than data dependence and not supported by the Closure Compiler. Furthermore points to analysis information are not integrated in the Code Property Graph yet. This three reasons made data dependence clearly the correct choice within the framework of this thesis.

Different data dependence and points to analysis implementations are categorised in so called sensitivities. To be sensitive means to be precise with regard to a certain aspect. E.g. the data dependence analysis of this thesis is flow sensitive. This means during analysis it takes the execution order of the statements into account. Therefore, it correctly determines that in var s = 1; s = 2; alert(s); the alert call uses the second definition of s instead of the first. A flow insensitive analysis could not decide which definition is used in cases like this.

The discussion about limitation and improvements in the further sections revealed that an attacker could 'hide' used variables in objects very easily because the used data flow analysis cannot handle object use very well. The used data flow analysis
Figure 4.8: Two source code listings and data flow graphs demonstrating the difference between points to and data dependence analysis. The upper graph is the data dependence graph, the lower is produced by points to analysis.
var o = {};
o.interval = 100;
setInterval(function() {socket.write();}, o.interval);

Figure 4.9: Example comparing a field insensitive (left site) to a field sensitive dataflow analysis on the right. On the right side it is easier to track the value used in `setInterval` trigger.

is not able to distinguish between different fields of an object; this property is called ‘field insensitive’. The graph 4.9 demonstrates this problem and compares the used field insensitive solution to a field sensitive solution. The `USES` edge in the example points to the object definition in 1. Hence, it is more difficult to get information about the interval value of the `setTimeout` call. The data dependence edge in the right graph produced by a field sensitive analysis points to the correct definition of `o.interval` at line 2; this makes it easier to determine the value used as interval.

To conclude, adding field sensitivity to the data flow analysis of this thesis would help to deal with objects and arrays in a proper way.

For now control dependence could not be used for detection purposes. No detailed control flow information is used for different reasons. The detection is based on locating Attack Complexes in flows. Some of the traversals used to construct these are not using control related information at all; as the query to connect socket creations to writes (listing 4.2). The traversal connecting triggers and writes (listing 4.9) follows the control flow of the program, but considers all paths possible without analysing if the path is possible during execution. Control dependence could become useful for improvements of this traversal. However, sound handling of function pointers, which is a data flow based problem, is a basic limitation of this traversal (see section 4.1.2). Therefore the author believes that a more precise dataflow analysis is of more importance because it causes more significant weaknesses in both mentioned traversals.
4.4 Detection Reports

The main goal of the reports is to categorise and prioritise the analysed flows for a human reviewer to help her or him to focus on the most suspicious programs. Furthermore, hints where to start the reviewing process should be included, e.g. information in which node the different parts of an Attack Complex are located.

The analysed programs will be classified as ‘Suspicious’, ‘Harmless’ and ‘Did not compile’. ‘Suspicious’ flows could perform a denial of service attack and are prioritised by a scoring system explained below. Programs categorised as ‘Harmless’ are believed not to contain an attack. Some flows cannot be compiled because of uncompileable function node source code or an unknown type of node used in them; they will be classified as ‘Did not compile’. If a lot of programs did not compile, this could indicate a bug in the detection software. To support debugging, information about the problem is added to the report.

The ‘Suspicious’ flows are scored from 1 to 10. A program with an Attack Complex inside is scored with 1. If detection refinements could find further dubious code structures, the score will increase. To be more precise, the detection of a suitable Attack Complex is the base line and any further dubious detection results raise the score of the flow.

The two detection refinements implemented during this work (‘Suspicious Strings’ and ‘Multiple Target Writes’) present, how categorising and prioritising can depend on detection. The category of a program depends on the existence of an Attack Complex in it; if an Attack Complex is detected, it is ‘Suspicious’. ‘Multiple Target Writes’ filters out flows with an unsuitable Attack Complex for Slowloris and Slow Post from this category and classifies them as ‘Harmless’. If a suitable Attack Complex was found, the discovery of dubious strings increases the score. Table 4.6 illustrates categorising and scoring implemented.

<table>
<thead>
<tr>
<th>SYN flood AC</th>
<th>Slow HTTP AC</th>
<th>Multiple Targets</th>
<th>SS</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>0</td>
</tr>
<tr>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>0</td>
</tr>
<tr>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>1</td>
</tr>
<tr>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>5</td>
</tr>
<tr>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>1</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>max(1, 5)</td>
</tr>
</tbody>
</table>

Table 4.6: Categorising and scoring implemented by this work. A score of 0 represents the category ‘Harmless’. AC = Attack Complex, SS = Suspicious String

A screenshot of the HTML report from this work is shown in 4.10.

As one can see, names and ids of the NodeRED nodes containing the Attack Complex are included to support the reviewer. Scores for each denial of service attack are displayed in the upper right corner of every suspicious flow reported. The score for the whole flow is the maximum of the scores for different attacks since scores for different attacks are independent; therefore, they should not be summarised.
Figure 4.10: Report for 9 analysed flows.
4.5 False Positive Tests

The flows from the NodeRED webpage were chosen as false positive test suite. False positive testing was planned to show which parts of the detection needs to be refined. This objective was hindered because no false positives could be found. For this reason NodeJS programs were discussed as an alternative.

The exposé of this work set the goal to run false positive testing on around 150 different flows from NodeRED webpage. For this purpose the flows were downloaded as JSON files by a Python script implemented by the author and were analysed with the FalsePositiveTesting main class, which adds some reporting and error handling to deal with the high amount of input files. E.g. if one input file leads to a program aborting error the analysing work done before should not get lost.

It was known before the tests started that most likely no spurious results could be found for the SYN flood detection since it uses the rawsocket package not included in native NodeRED. But the author was convinced to find some cases which would help to improve the Slowloris and Slow HTTP detection. Surprisingly no such flows were discovered. The reason for this is that none of the scanned programs used a TCP related node. To sum up the test suite could not produce any false positive because TCP sockets and raw sockets are not used in the public NodeRED programs.

Even though the false positive testing did not help to make the detection process more precise, it still improved the robustness of the detection process. The detection ran on many different real-world programs; as a result, error handling for unknown NodeRED node types and compile errors in faulty function nodes have been added. In addition logging was implemented to locate erroneous inputs and recover from program crashed without losing all progress.

Alternatively NodeJS programs were considered for false positive testing, but to run the detection on the mentioned NodeRED flow took a lot of time scheduled for false positive testing and some additional work would have been necessary to adjust the process for NodeJS. Therefore, NodeJS false positive finding remains a starting point for further work. If this approach is to be tried, fundamental restrictions of this work may need to be reconsidered. E.g. inter procedural analysis or prototype inheritance is not covered in this work because it is assumed to be unnecessary to analyse NodeRED flows. However, this assumption is possibly incorrect for NodeJS programs. Furthermore, NodeJS perhaps has some special cases like message sending in NodeRED which have to be considered for a meaningful reasoning about denial of service attacks.

To conclude false positive testing stays a weak point of this thesis. The public NodeRED flows tested could not produce any spurious results. Due to time constraints no alternatives could be investigated. Nevertheless, the false positive testing greatly improved stability and robustness of the implemented program.
Chapter 5

Conclusion

As stated in the introduction the author believes that especially denial of service attacks could be dangerous in connection with NodeRED because they are easy to implement and understand. This assumption was verified by this thesis; it is straightforward to implement SYN flood, Slowloris and Slow Post with NodeRED. A flow performing a Slowloris attack is not complex since it contains only 7 nodes. The rich functionality of native NodeRED nodes could allow persons who do not even understand basic program structures like if statements or loops to program a denial of service attacks.

This thesis is motivated by the scenario that an owner of a public NodeRED server could be liable for denial of service attacks launched from his server. The software implemented during this thesis could be installed on the public server and a trigger could be configured to generate reports for all flows uploaded. By reviewing only flows which are reported to be suspicious the server owner could avoid analysing most likely harmless flows and save a lot of time. False positive tests showed that TCP related nodes are not used very often in real-world flows. As a consequence not many programs should be reported to be suspicious. However, most likely the detection is not sophisticated and precise enough to be practically used.

The detection mechanisms developed are still very fundamental and imprecise, but they are also generally usable for all denial of service attacks considered and further, more precise detection can be based on them. Some reasons for the detection mechanism being imprecise are explained in the paragraph below.

No previous work about static denial of service detection in JavaScript could be found for orientation. This is no surprise since the scenario of being the one to perform and at the same time being the one to prevent the attack is not a common scenario. Therefore, the author used the Code Property Graph by Yamaguchi as basis to build his own detection approach from scratch. Starting at the very beginning is naturally a long and time consuming process; in addition, the author hoped for more support by the Closure Compiler. The Closure Compiler does not offer methods to generate the control dependence part of the Code Property Graph; thus, the algorithm to generate a post dominator tree and the control dependence were researched and implemented by the author. Furthermore, no data dependence graphs can be produced by the Closure Compiler; hence, the author had to understand, rewrite and combine two data flow analyses from the Closure Compiler. Thus, the
generation of both dependence graphs was estimated to need less time than actually used. The extension of the Code Property Graph by the call graph was another part of the work not scheduled. However, the call graph adds value to the Code Property Graph generation, which can be reused in further research. During the later parts of this project, it became obvious that precise denial of service detection needs a more detailed data dependence, but a less precise control dependence analysis than Yamaguchi’s project which these thesis is inspired from (section 4.3 explains this fact in depth). The author regrets that no further work for a more precise detection could be done within the framework of a bachelor thesis; however, the second goal of generalising the denial of service detection could be fully met. The idea of Attack Complexes storing all socket creations, socket writes and triggers in a flow was introduced. These three code structures are common to SYN flood, Slowloris and Slow Post. They were found in the suggested MQTT attack of this thesis as well. Therefore the author is optimistic that the Attack Complex approach can be applied to other kinds of denial of service attacks easily.

During his presentation the author was asked, how useful the Code Property Graph was to detect denial of service attacks. This question cannot be fully answered within the framework of this thesis because no alternative approaches were tried. However, it can be concluded that a promising and general detection mechanism could be found by using the Code Property Graph.

Furthermore the Code Property Graph generation for JavaScript is a considerable contribution of this thesis. The reuse of the graph in advanced static analysis of JavaScript was planned for during the implementation of this project and the modularisation will support other researchers to profit from it.
Appendix A  Control Dependence Definition

The formal definition of control dependence used in this thesis is given below. It is taken from [19].

**Definition 1**  A control flow graph is a directed graph $G$ with a unique ENTRY and EXIT node. Every node in this graph has at most three successors. For any node $N$ in $G$ exists a path from ENTRY to $N$ and from $N$ to EXIT. The control flow graph of this thesis fulfils this definition.

This definition was slightly changed to accommodate control flow edges for exception handling. Therefore, every node in the control flow graph might have three successors; one more than in the original definition.

**Definition 2**  A node $N$ is post-dominated by a node $M$ if every path from $N$ to EXIT (not including $N$) contains $M$.

**Definition 3**  Let $G$ be a control flow graph and $X$ and $Y$ be nodes of $G$. $Y$ is control dependent on $X$ if

1. there exists a path $P$ from $X$ to $Y$ with every $Z$ in $P$ (excluding $X$ and $Y$) post-dominated by $Y$ (this condition might be satisfied by a single edge path; then no $Z$ is existing) and

2. $X$ is not post-dominated by $Y$. 
Bibliography


[29] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7313 LNCS:
435–458, 2012. ISSN 03029743. doi: 10.1007/978-3-642-31057-7-20. URL
http://www.researchgate.net/publication/262290966_Correlation_tracking_for_points-to_analysis

http://ediss.uni-goettingen.de/handle/11858/00-1735-0000-0023-9682-0.

http://user.informatik.uni-goettingen.de/~krieck/docs/2014-ieeesp.pdf.
Ich bestätige, dass ich diese Arbeit eigenständig geschrieben und angefertigt habe.
Weiterhin bestätige ich, dass die Arbeit nur am Lehrstuhl von Professor Posegga an der Universität Passau abgegeben habe.
Mit freundlichen Grüßen,
Per Fuchs