Graph-Driven Security: An Interpreter for an Account Access Language with Neo4j Integration

Blair Walker

Master of Science
Cyber Security, Privacy and Trust
School of Informatics
University of Edinburgh
2023

Abstract

User account access graphs are an incredibly effective tool for visualising the connections of accounts in the modern world. Arnaboldi et al. [2] provide theoretical extensions that allow the manipulation of these graphs in a realistic manner. This report provides an interpreter to process a subset of their defined tactic language which automatically updates or produces an account access graph with state using Neo4j. The automatic visualisation provided by said graph database management software eliminates the greatest barrier to entry for utilising the aforementioned graphs - manually producing them. The interpreter is shown to be effective at visualising situations as diverse as a phone theft, leading to Twitter account compromise, to illustrating the dangers associated with password reuse. These contributions offer a significant advancement in the field of account access graphs, providing ease of use and groundwork that future research can build upon.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Blair Walker*)

Acknowledgements

- Firstly, I would like to thank my supervisor Dr David Aspinall for all his help and advice and for providing me with such an interesting and rewarding topic to pursue.
- Secondly, I would like to thank Sandor Bartha who, despite just returning from his holiday and recovering from an injury, took the time to read through my entire dissertation. His insights and feedback were very valuable.
- Finally, I must thank my fiancée Lizzie who, without her incredible support, I would never have been able to finish the first semester of my degree, let alone complete this dissertation.

Table of Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Aims and Objectives 3

1.3 Project Structure 4

2 Background **5**

2.1 Account Access Graphs 5

2.1.1 Properties of Account Access Graphs with State 6

2.1.2 Operations on Account Access Graphs 7

2.1.3 Tactics 9

2.1.4 Tactics for Account Access Graph Operations [2] 9

2.2 Neo4j 10

3 Interpreter Design **11**

3.1 Changes To the Language 11

3.2 Neo4j Integration 12

3.2.1 Atomic Propositions 13

3.2.2 Operations 14

3.2.3 Python 16

3.3 ANTLR 17

3.4 Tactics Grammar 18

3.5 Visitor 19

3.6 Graph Updates 21

3.7 Neo4j Parser Integration 21

4 Evaluation and Testing **23**

4.1 Evaluation with Case Studies 23

4.1.1 Initial Success 23

4.1.2 Twitter Compromise . . . 24
4.1.3 Banking App Compromise . . . 28
4.2 University of Edinburgh: Multi-Factor Authentication . . . 30
4.3 Testing . . . 32
4.3.1 Testing the Visitor . . . 32
4.3.2 Testing the Graph Updates . . . 36

5 Conclusion **38**
5.1 Conclusion and Critical Evaluation . . . 38
5.2 Challenges Faced in the Project . . . 39
5.3 Limitations and Future Work . . . 39

Bibliography **41**

A Account Access Graph Background **44**
A.1 Account Access Graphs Properties (Validity) [2] . . . 44

B Neo4j Code **45**
B.1 Properties of Account Access Graphs Code . . . 45
B.1.1 is_account(v) . . . 45
B.1.2 has_access_a(v) . . . 45
B.1.3 could_access_a(v) . . . 46
B.1.4 uses_method_l(u,v) . . . 46
B.2 Operations . . . 47
B.2.1 gain_access_a,v . . . 47
B.2.2 disc_access_a,v . . . 47
B.2.3 lose_access_a,v . . . 47
B.2.4 create_account_a,v . . . 48
B.2.5 del_account_a,v . . . 48
B.2.6 add_account_a,{u_1, ..., u_2},v,l . . . 48
B.2.7 rem_access_1_a,v,l . . . 49
B.2.8 rem_access_2_a,u,l . . . 49
B.3 Tactics Grammar . . . 49

Chapter 1

Introduction

1.1 Motivation

The compromise of a journalist's photographs and digital accounts resulting in permanent data loss [12], over 1 million dollars of cryptocurrency stolen [22, 24], and a series of iPhone thefts that led to bank account drain and credit card fraud [25]. The thread that connects each of these exploits is a severe lack of understanding regarding the increasing interconnection of user accounts in the current world.

An individual's digital identity is a web of accounts and access methods. Now, more than ever [3], people rely on these accounts for every aspect of their personal and work life. To take one example, large corporations like Apple may carry out tens of thousands of account recoveries every day [1]. To make this process as painless as possible, every aspect of an iPhone and Apple account can be recovered from just the respective PIN number and device [25]. While this may enable frictionless account recovery, it equally provides an Achilles heel for the user's whole Apple ecosystem.

A malicious actor that is able to steal just a phone and corresponding PIN would have complete control over every aspect of the victim's Apple-related identity. This has led to a growing problem where users have had all personal data wiped and substantial fiscal losses [25]. Apple has complex incentives. While they want their platform to be easier to use than any of their competition, they must also be careful that this does not lead to excessive security vulnerabilities. This instance demonstrates how difficult it is to balance these factors.

The overall state of account connections can be summarised in the following xkcd comic on authorisation from 2013 that has been modified to reflect the current environment:

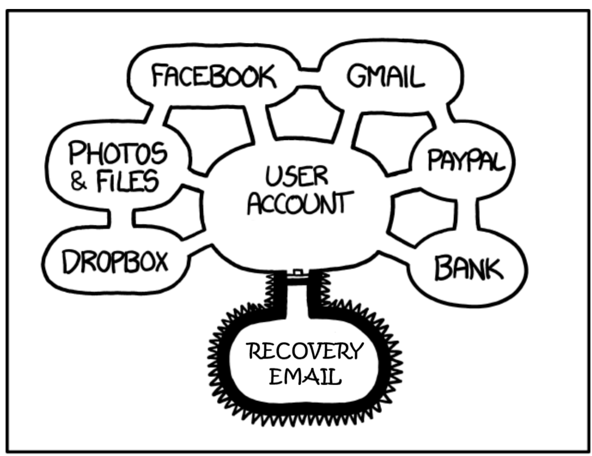

Figure 1.1: Modified xkcd comic (Based on "Authorization" [15])

Fortunately, there is a formal visual representation that can illustrate these risks which may help to understand, and perhaps avoid, them in system design: user account access graphs.

Initially defined by Hammann et al. [11], these graphs represent accounts as vertices and access methods as edges to provide a tool which highlights digital connection. However accounts, by their very nature, are not static. Users can create, modify, and delete their accounts which are requirements for these graphs if they are to accurately represent and identify real-world vulnerabilities. This is the gap in research which a draft paper by Arnaboldi et al. [2] seeks to fill. They extend user account access graphs to include the access participants have to any given account, along with the definition of atomic propositions and operations which are combined to create a language able to describe changes representative of genuine accounts. Their graphs are manipulated by short programs ("tactics") in this language.

This report builds upon their work by visualising the extended user account graphs using the graph database management system Neo4j [17]. To interact with and build these graphs a simplified tactic language, complete with operational semantics, is defined based on the original by Arnaboldi et al.

The overall outcome of this work is an interpreter that translates any tactic written in the simplified tactic language into a Neo4j representation using the Neo4j query language Cypher. The specialised graph visualisation associated with Neo4j is therefore

immediately available after each tactic is carried out. These tactics can be used to build an account access graph from scratch or to edit a pre-existing graph that conforms to the necessary structure.

Ultimately, this line of work could provide benefit to a wide range of people. Large companies like Apple can use it to investigate access methods in their vast ecosystem and determine security trade-offs for individuals' devices and their own systems. Security researchers can make use of it to describe connections and weaknesses in systems of interest and update their models automatically rather than manually recreate them for changes. Finally, individual end-users can enter their own details to discover weaknesses in their personal and work account networks and adjust them to better fit their risk tolerance.

Through this dissertation, the reader will come to understand the design of the interpreter, its integration with Neo4j, and the types of scenarios that can be effectively described. However, a key benefit of the interpreter implementation is that detailed knowledge is not required to use it. A basic understanding of the simplified tactic language (defined at the start of Chapter 3) and an existing Neo4j database instance (Neo4j Desktop and AuraDB both supported) are all that is required to make use of this tool. Hence, the interpreter provides a very low barrier for entry as there is no need for prerequisite knowledge about parsing or graph database management systems.

1.2 Aims and Objectives

The overall aim of this project is to create a system that can interpret tactics written in a simplified language based on the tactic language formalised by Arnaboldi et al. [2], producing or updating a visualisation of the relevant graph. To achieve this aim, the following objectives were completed:

1. The creation of a simplified tactic language with precise definitions through an operational semantics.
2. The development of a parser (including defining precise syntax) to analyse tactics written in the simplified tactic language.
3. The development of appropriate Cypher queries to create, check and update the state of an account access graph.

4. The integration of the Cypher queries with the parser and a graph updates class to interpret the defined tactic language into a Neo4j database with implicit visualisation.

5. Evaluation and testing of objectives 2, 3 and 4 to ensure appropriate functionality and capability of the interpreter.

1.3 Project Structure

This dissertation is divided into the following sections: Chapter 1 has introduced the topic, Chapter 2 shall provide further background on the underlying theory and graph database to be utilised. Chapter 3 shall discuss the design choices and implementation of the Interpreter. Chapter 4 shall go over multiple case studies showing the interpreter's applications and testing to make sure all aspects function as expected. Finally, Chapter 5 shall go over the relevant conclusions and potential future work.

Chapter 2

Background

2.1 Account Access Graphs

As mentioned in the introduction, Arnaboldi et al. [2] extend the account access graphs defined by Hammann et al. [11]:

Let $\mathcal{V}$ be a countably infinite set of vertices, let $\mathcal{L}$ be a countably infinite set of labels, and $\mathcal{A}$ be a set of participants, here $\mathcal{A}$ will contain the user (signified by the icon 👤) and an adversary (signified by 🕵).

Definition 1 (Account Access Graphs with State [2]). *An account access graph with state is a triple $G = (V, E, A)$ where $V \subset \mathcal{V}$ is a finite set of vertices (representing devices, accounts or credentials), $E : (V \times V) \to 2^{\mathcal{L}}$ are edges labelled with finite sets of access method names, and $A : V \to 2^{\mathcal{A}}$ is a map labelling each vertex with a finite set of participants who have access.*

This definition enables the observation of the access held by both users (👤) and adversaries (🕵). This access is illustrated in the graph by mapping a set A(v) to each node v which can contain an individual user, an adversary, a combination of both or can be empty. This mapping is critical as the accounts that a participant can gain access to are directly dependent on which accounts they already have access to.

The term 'account' can lead to some confusion, in the context of user account access graphs, a vertex can be either an account, device or credential. For the sake of consistency with the definitions used by Arnaboldi et al. [2], throughout this report the word 'account' shall be used to refer to any of these three entities.

Below is an example of this type of graph. It illustrates a situation where the adversary has acquired access to the 'Locked phone' and the 'Phone PIN' but only

the user has access to the 'Face' node. The adversary can then use the edges labelled 'PIN' to access the 'Phone' node (i.e. the phone in its unlocked state). This is possible because the adversary controls all nodes that connect edges with that specific label to 'Phone'. However, the adversary cannot access the phone by using 'faceID' as they do not have access to the 'Face' node, despite having access to the 'Locked Phone' node.

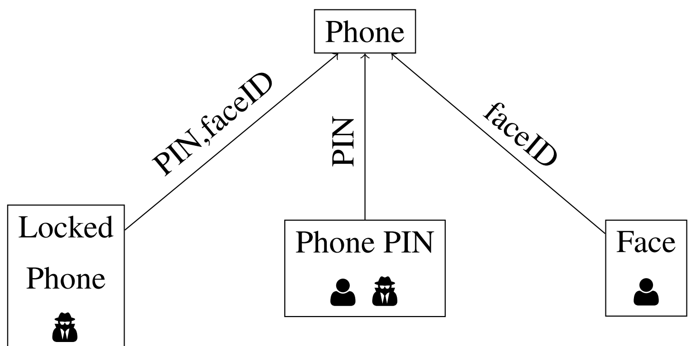

Figure 2.1: Account Access Graph with State [2]

The key takeaway from this example is that edges and their labels allow the expression of both alternative access routes and multi-factor requirements, such as Multi-Factor Authentication (MFA) and Two-Factor Authentication (2FA).

For instance, if there was an additional label on the edge from 'Locked Phone' to 'Phone' called "Swipe" and no other edges match this label, then one could access the phone from just the locked phone using the swipe access method, bypassing the need for 'Phone PIN' or 'Face' access. However, in the example above, access can only be achieved from having access to at least two nodes. This principle can equally be applied to nodes representing two or more devices to visualise multi-factor requirements which shall be seen later.

2.1.1 Properties of Account Access Graphs with State

To help describe operations to manipulate access graphs, formal properties can be used to describe states. A property of an account access graph G is a predicate over (V,E,A). Atomic propositions and combinations are given by the subsequent grammar [2]:

$$
\begin{aligned}
\phi ::= & \\
& \text{is_account}(v) && \textit{account v exists in graph} \\
& |\,\text{has_access}_a(v) && \textit{user a is accessing account v} \\
& |\,\text{could_access}_a(v) && \textit{user a has way to access v} \\
& |\,\text{uses_method}_l(u,v) && \textit{v is accessible from u using label l} \\
& |\,\phi_1 \wedge \phi_2 \,|\, \phi_1 \vee \phi_2 \,|\, \neg\phi \,|\, \mathit{true}
\end{aligned}
$$

The inductive definitions showing validity of assertions can be found in Appendix A.1.

The above definition can be illustrated with the following examples (that all evaluate to true) based on Figure 2.1:

$$
\begin{aligned}
\mathit{is_account}(\text{Locked Phone}) \\
\mathit{has_access}_{🕵}(\text{Locked Phone}) \\
\mathit{could_access}_{🕵}(\text{Phone}) \\
\mathit{uses_method}_{\text{PIN}}(\text{Phone PIN, Phone})
\end{aligned}
$$

2.1.2 Operations on Account Access Graphs

Arnaboldi et al. [2] provide seven operations to update an account access graph. The first three operations (GAIN, DISC, and LOSE) only update the access of participants, modifying the mapping $a \in A(v)$, and thus do not have an effect on the shape of the graph. These three operations represent an actor gaining access to a vertex, an adversary discovering access (without any necessary requirements) and an actor losing access to a vertex, respectively. The following four operations (CREATE, DELETE, ADD, REMOVE1, and REMOVE2) all update the structure of the graph. These operations represent: the creation of a new account, the deletion of an existing account, the addition of a specific account as an access method for another account, removing access for an account by direct access, and removing access to an account through access to one of its access methods. The formal definitions and rules for these operations are as follows:

Definition 2 (**Account Access Graph Operations [2]**). *An account access graph with state can be modified with the following seven operations. We use* α *to range over these operations.*

$$\frac{\langle V,E,A\rangle \models could_access_a(v)}{\langle V,E,A\rangle \overset{gain_access_{a,v}}{\rightsquigarrow} \langle V,E,A[v \mapsto a]\rangle} \tag{GAIN}$$

$$\langle V,E,A\rangle \overset{disc_access_{a,v}}{\rightsquigarrow} \langle V,E,A[v \mapsto a]\rangle \tag{DISC}$$

$$\langle V,E,A\rangle \overset{lose_access_{a,v}}{\rightsquigarrow} \langle V,E,A[v \setminus a]\rangle \tag{LOSE}$$

The notation $A[v \mapsto a]$ means A updated to add a into the access set of v, i.e. the updated map A' given by:

$$A'(x) = \begin{cases} A(v) \cup \{a\} & \text{when } x = v \\ A(x) & \text{otherwise.} \end{cases}$$

Similarly, $A[v \setminus a]$ means A updated to remove a from the set of accesses A(v).

$$\langle V,E,A\rangle \overset{create_account_{a,v}}{\rightsquigarrow} \langle V \uplus \{v\},E,A[v \mapsto a]\rangle \tag{CREATE}$$

$$\frac{\langle V,E,A\rangle \models has_access_a(v)}{\langle V,E,A\rangle \overset{del_account_{a,v}}{\rightsquigarrow} \langle V \setminus \{v\},E \setminus v,A|_{V \setminus \{v\}}\rangle} \tag{DELETE}$$

$$\frac{\langle V,E,A\rangle \models has_access_a(v) \quad E'(x,y) = \begin{cases} \{l\} & \text{if } x \in \{u_1,...,u_n\} \text{ and } y = v \\ \{\} & \text{otherwise.} \end{cases}}{\langle V,E,A\rangle \overset{add_accessa,\{u_1,...,u_n\}v,l}{\rightsquigarrow} \langle V,E \uplus E',A\rangle} \tag{ADD}$$

$$\frac{\langle V,E,A\rangle \models has_access_a(v)}{\langle V,E,A\rangle \overset{rem_access_{a,v,l}}{\rightsquigarrow} \langle V,E[(_,v) \setminus \{l\}],A]\rangle} \tag{REMOVE1}$$

$$\frac{\langle V,E,A\rangle \models has_access_a(v) \quad \langle V,E,A\rangle \models uses_method_l(v,u)}{\langle V,E,A\rangle \overset{rem_access_{a,u,l}}{\rightsquigarrow} \langle V,E[(,u) \setminus \{l\}],A\rangle} \tag{REMOVE2}$$

Here the notation $E \setminus v$ denotes the edge function E updated to remove any edges that have v as a source or target, i.e., E' such that:

$$E'(u_1,u_2) = \begin{cases} \{\} & \text{if } u_1 = v \text{ or } u_2 = v \\ E(u_1,u_2) & \text{otherwise.} \end{cases}$$

Similarly, the notation $E[(_,v) \setminus \{l\}]$ means the update of E to remove the access method l to vertex v from E.

2.1.3 Tactics

The way in which account access graphs are updated is through a concise program known as a 'tactic'. They can be used to model attacks by an adversary, defences by a user and the combination of both sequentially. These tactics make use of the operations defined previously to achieve a specific goal. Tactics can also be comprised of $\top$, the tactic for successful termination, and $\bot$, the tactic for failure. The following grammar defines a syntax for tactics, where α is an operation, $b \in \{\top, \bot\}$ and ϕ represents the atomic propositions also defined previously:

$$t ::= \alpha \mid b \mid t\,;t \mid t||t \mid \text{CHECK}(\phi)$$

2.1.4 Tactics for Account Access Graph Operations [2]

Arnaboldi et al. [2] define a big-step semantics to provide rules for tactic evaluation. Let σ represent the state of an account access graph (V,E,A) and $\langle\sigma\rangle\; t \Downarrow t' \;\langle\sigma\rangle$ be a relation that extends the previous transition relations. The execution of a tactic results in either success ($\top$) or failure ($\bot$) along with the account access graph with updated state. Then, the following list of rules inductively defines tactic evaluation for account access graphs [2]:

$$\frac{\langle\sigma\rangle \stackrel{\alpha}{\rightsquigarrow} \langle\sigma'\rangle}{\langle\sigma\rangle\; \alpha \;\Downarrow\; \top\,\langle\sigma'\rangle} \qquad \text{(AX-T)}$$

$$\langle\sigma\rangle\; b \;\Downarrow\; b\,\langle\sigma'\rangle \qquad \text{(CONST)}$$

$$\frac{\langle\sigma\rangle\; t_1 \;\Downarrow\; \bot\,\langle\sigma'\rangle}{\langle\sigma\rangle\; t_1\,;t_2 \;\Downarrow\; \bot\,\langle\sigma'\rangle} \qquad \text{(SEQ-B)}$$

$$\frac{\langle\sigma\rangle\; t_1 \;\Downarrow\; \top\,\langle\sigma'\rangle \quad \langle\sigma'\rangle\; t_2 \;\Downarrow\; b\,\langle\sigma''\rangle}{\langle\sigma\rangle\; t_1\,;t_2 \;\Downarrow\; b\,\langle\sigma''\rangle} \qquad \text{(SEQ-T)}$$

$$\frac{\langle\sigma\rangle\; t_1 \;\Downarrow\; \bot\,\langle\sigma'\rangle \quad \langle\sigma'\rangle\; t_2 \;\Downarrow\; b\,\langle\sigma''\rangle}{\langle\sigma\rangle\; t_1||t_2 \;\Downarrow\; b\,\langle\sigma''\rangle} \qquad \text{(OR-B)}$$

$$\frac{\langle\sigma\rangle\; t_1 \;\Downarrow\; \top\,\langle\sigma'\rangle}{\langle\sigma\rangle\; t_1||t_2 \;\Downarrow\; \top\,\langle\sigma'\rangle} \qquad \text{(OR-T)}$$

$$\frac{\langle\sigma\rangle \models \phi}{\langle\sigma\rangle\; \text{CHECK}\,\phi \;\Downarrow\; \top\,\langle\sigma\rangle} \qquad \text{(CHECK-T)}$$

$$\frac{\neg\langle\sigma\rangle \models \phi}{\langle\sigma\rangle\; \text{CHECK}\,\phi \;\Downarrow\; \bot\,\langle\sigma\rangle} \qquad \text{(CHECK-F)}$$

The above rules allow for the combination of operations, true or false values and checks on the states of graphs.

2.2 Neo4j

Neo4j is a graph database management system employed by many notable companies, such as NASA [14] and AstraZeneca [23]. It is a native graph database meaning that data is stored as a graph, with nodes and edges (or relationships), which avoids needing to search an index for relationships at query time. This makes it more efficient for path queries than relational databases [28] and an appropriate choice for implementing account access graphs, as the graphs can be stored exactly as they are defined. The use of Neo4j therefore provides an excellent basis to represent large systems of interconnected accounts and devices.

Neo4j supplies an integrated visualisation tool called 'Neo4j Browser' [8]. It is a clear benefit of using the system as it can visualise all of the nodes and edges of a stored graph or a specified subset of the graph. Control over what is visualised and the ability to create and update graphs is provided by the Cypher query language, which has an intuitive visual syntax for matching graph patterns [7]. The Cypher query language shall be utilised extensively for the tactic language, to develop atomic propositions and tactic operations equivalent to those specified by Arnaboldi et al. [2].

The standard syntax for Neo4j is to write labels with CamelCase and relationships as all upper-case [10]. Hence the representations of account access graphs in the rest of this report will follow this convention. There shall be only one diversion; whitespace is perfectly acceptable for properties of nodes according to the Neo4j style guide but the tactic language used in this report shall use underscores instead. This is to aid with parsing which shall be discussed later.

Chapter 3

Interpreter Design

3.1 Changes To the Language

Given the time constraints imposed by the MSc project, it was decided to focus on implementing a subset of the tactic language rather than the entirety of its features and capabilities. Thus, in correspondence with the project supervisor Dr David Aspinall, the design decision was made to simplify the language so that it did not make use of backtracking. A typical problem with database updates, and graph updates in particular, is the choice required for queries that update state continuously. The originally defined tactic language [2] continuously updates state but reverts (backtracks) to a previous point if the current branch is determined not to lead to a solution. To avoid this, the choice was made to only update the state of the graph at the very end of the tactic, once the whole parse tree has been walked. All individual components of the language shall be implemented but, to avoid the backtracking problem, any queries on state will be carried out on the initial state of the graph before the tactic was interpreted, rather than some intermediate state.

This simplification was accomplished by splitting the interpreter into two stages: checks and updates. The checks stage does not update the state of the graph but rather carries out the logic of the language in relation to $b \in \{\top, \bot\}$, checking the atomic propositions (which result in either $\top$ or $\bot$) and recording all the valid α (operations) in relation to these in a list.

The updates stage then carries out the appropriate operations α in the list returned by the checks stage, provided the premises for the operations pass.

Therefore, to represent this simplified tactic language, big-step semantics were defined for the two stages. For the big step semantics, σ is a Neo4j account access graph

(V,L,E,A) (defined in the next section) and the relation $t \Downarrow l,\ t'$ represents the execution of a tactic which returns a list and another tactic. Executing a tactic initially results in a list and $\top$ (success) or $\bot$ (failure).

$$\langle\sigma\rangle \vdash \alpha \ \Downarrow\ [\alpha],\ \top \tag{AX-T}$$

$$\langle\sigma\rangle \vdash b \ \Downarrow\ [\,],\ b \tag{CONST}$$

$$\frac{\langle\sigma\rangle \vdash t_1 \ \Downarrow\ [\,],\ \bot}{\langle\sigma\rangle \vdash t_1\,;t_2 \ \Downarrow\ [\,],\ \bot} \tag{SEQ-B}$$

$$\frac{\langle\sigma\rangle \vdash t_1 \ \Downarrow\ l_1,\ \top \quad \langle\sigma\rangle \vdash t_2 \ \Downarrow\ l_2,\ b}{\langle\sigma\rangle \vdash t_1\,;t_2 \ \Downarrow\ l_1 +\!+ l_2,\ b} \tag{SEQ-T}$$

$$\frac{\langle\sigma\rangle \vdash t_1 \ \Downarrow\ [\,],\ \bot \quad \langle\sigma\rangle \vdash t_2 \ \Downarrow\ l_2,\ b}{\langle\sigma\rangle \vdash t_1\,||\,t_2 \ \Downarrow\ l_2,\ b} \tag{OR-B}$$

$$\frac{\langle\sigma\rangle \vdash t_1 \ \Downarrow\ l_1,\ \top}{\langle\sigma\rangle \vdash t_1\,||\,t_2 \ \Downarrow\ l_1,\ \top} \tag{OR-T}$$

$$\frac{\langle\sigma\rangle \models \phi}{\langle\sigma\rangle \vdash \mathrm{CHECK}\,\phi \ \Downarrow\ [\,],\ \top} \tag{CHECK-T}$$

$$\frac{\neg\langle\sigma\rangle \models \phi}{\langle\sigma\rangle \vdash \mathrm{CHECK}\,\phi \ \Downarrow\ [\,],\ \bot} \tag{CHECK-F}$$

For the second big-step semantics, the state of the graph is updated. The relation $\langle\sigma\rangle\ \alpha :: l \Downarrow l\ \langle\sigma'\rangle$ represents the execution of one operation in the list and that operation being removed from the list, along with the updated state of the graph.

$$\langle\sigma\rangle\ [\,] \Downarrow \langle\sigma\rangle \tag{LIST-E}$$

$$\frac{\langle\sigma\rangle \overset{\alpha}{\rightsquigarrow} \langle\sigma'\rangle}{\langle\sigma\rangle\ \alpha :: l \ \Downarrow\ l\ \langle\sigma'\rangle} \tag{LIST-A}$$

3.2 Neo4j Integration

The only deviation from the definitions set forth by Arnaboldi et al. [2] for this system is the use of multiple edges rather than a single edge with multiple property labels. This is because Neo4j is optimised for relationship based queries which are more efficient than searching through a list of properties of a single relationship [16]. This deficiency would be especially evident for a single account that acts as a joint access method for many ways of entering another account. For instance, a phone node that can be used as

part of a 2FA scheme with many other accounts. Properties require loading the entire list to then filter out the irrelevant entries, whereas individual relationships automatically filter out the irrelevant data with appropriate Cypher queries. Thus, the definition from Chapter 2 can be slightly modified into one that shall be adhered to throughout the rest of this project:

Definition 3 (Neo4j Account Access Graphs). *A Neo4j account access graph is a quadruple* $G = (V, L, E, A)$ *where* $V \subset \mathcal{V}$ *is a finite set of vertices,* L *is a finite set of access methods,* $E \subseteq V \times V \times L$ *are edges labelled with access methods, and* $A : V \to 2^{\mathcal{A}}$ *is a map labelling vertices with a finite set of participants.*

This is implemented in Neo4j as follows: each vertex has an 'Account' label, this is required as vertices must have at least one label. They may also have a User and/or Adversary label equivalent to the mapping of $a \in A(v)$. Additionally, vertices are assigned a name property equivalent to the name of the account and relationships are equivalent to edges.

The example from Chapter 2, now adhering to the above definition, can be seen represented in Neo4j below:

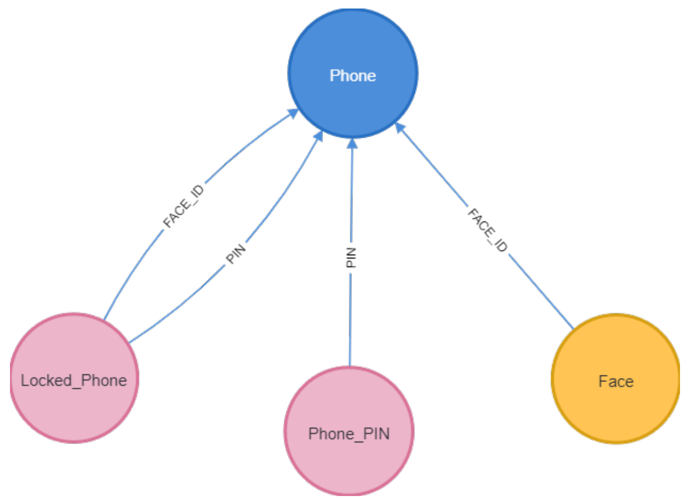

Figure 3.1: Neo4j Implementation of an Account Access Graph

3.2.1 Atomic Propositions

All of the atomic propositions from Arnaboldi et al. [2] were defined using the Cypher query language. These can be seen in Appendix B.1. Here, only a single proposition shall be covered to give an idea of the implementation.

uses_method_l(u,v):

```
:params {l: 'label', u: 'vertex_u', v: 'vertex_v'}

OPTIONAL MATCH (u)-[l]->(v)
WHERE type(l) = $l AND u.name = $u AND v.name = $v
RETURN
CASE
    WHEN l IS NOT NULL THEN true
    ELSE false
END AS `uses_method_l(u,v)`
```

For this proposition, optional match is used to allow the chance of failure. If a pattern exists which matches a vertex 'u' connected to a vertex 'v' by an edge 'l' then that relationship is stored in the variable 'l'. The case statement then simply returns true if such a relationship exists and false if it does not. This should provide the reader with an understanding of the intuitive nature of the Cypher query language.

Parameterisation was used as this shall be important when implementing the interpreter.

3.2.2 Operations

Similarly to the atomic propositions, all of the operations from Chapter 2 were implemented as Cypher queries. The premises for the operational semantics were all handled separately from the conclusions.

3.2.2.1 Operational Semantics - Premises

Fortunately, the premises for the majority of the operations use the atomic propositions that were already defined in the previous section. This means that, in most cases, said propositions can be called to ensure that the operations are valid to be carried out. There are two situations where this is not the case, a part of the premises for ADD and the premise for REMOVE2.

Part of the condition for ADD specifies the way in which the additional edges must connect the input vertices. This is already a requirement for the add_access operation, as it is defined in Neo4j, so this section of the rule does not need a specific Cypher query.

While the premise for REMOVE2 uses both $has_access_a(v)$ and $uses_method_l(v,u)$, the variable ‘v’ is not passed in through the $rem_access_{a,u,l}$ operation. Since ‘v’ is not provided then the current implementation of the relevant propositions are not viable for checking the condition to be satisfied. Thus a new Cypher query was developed specifically for the REMOVE2 premise:

```
:params {a: 'User/Adversary', u: 'uVertex', l: 'lEdge'}

OPTIONAL MATCH (v)-[l]->(u)
WHERE $a in labels(v) AND type(l) = $l AND u.name = $u
//only require one node to match
WITH v
LIMIT 1
RETURN
CASE
    WHEN v IS NOT NULL THEN true
    ELSE false
END AS `rem_2_rule_check`
```

This query matches any node for which the relevant $a \in A(v)$ is a label and that connects to vertex ‘u’ by edge ‘l’. There only has to be one such vertex for the premise to be satisfied, so the variable ‘v’ with a limit of 1 is carried over for the final part of the query. Then, if ‘v’ has a value, i.e. there is at least one ‘v’ that matches the pattern, true is returned and the operation can be carried out. If ‘v’ has a null value then there is no vertex that matches the pattern, so false is returned and the condition fails.

3.2.2.2 Operational Semantics - Conclusions

The Cypher queries for all of the operations can be found in Appendix B.2. As an example, seen below is the code for lose_access_a,v, i.e. actor ‘a’ losing access to an account ‘v’:

```
:params {v: 'vertex', a: 'User/Adversary'}

MATCH (v)
WHERE v.name = $v AND $a in labels(v)
CALL apoc.create.removeLabels(v, [$a])
YIELD node
```

```
RETURN node
```

The above code matches a vertex with name ‘v’ that actor ‘a’ has access to and then removes the label ‘a’ from that vertex. As with many of the operations, the APOC (Awesome Procedures on Cypher) Core library [9] is utilised. The APOC Core library is an extension of the Cypher query language that is also officially supported by Neo4j. While not a requirement to carry out these operations, this library allows for further parameterisation of the variables which would otherwise have to be entered into the query manually. APOC was not initially a part of the operations, however when implementing the interpreter it was found to be much more convenient if every variable was parameterised so the design decision was made to go back and refactor the Neo4j operations so that parameterisation was used.

3.2.3 Python

To incorporate the aforementioned queries into the interpreter the decision was made to use the Python programming language with the Neo4j Python Driver. Java was initially considered as Neo4j is a JVM-based database [29] and has wide support for Java. However, the design choice of Python over Java was made because development on Java required the use of the SDKMan package manager which did not run seamlessly on Windows. Thus, the use of Python ensures future development is straightforward and platform independent, and it is also one of the 5 officially supported Neo4j drivers.

The Neo4j Python Driver is quite straightforward to use. Instantiating the driver is expensive so it is only connected to once per python file [13]. In contrast, sessions using the driver are resource efficient and provide implicit error handling so these were used for every query on the Neo4j database. Read and write sessions were both used, atomic propositions and operational semantics premises are checked with read sessions and operation conclusions are carried out with write sessions. The error handling is a strong benefit of using multiple sessions as they will try to reconnect to the database if their requests do not go through and they will provide details should the session fail for some reason.

It was possible to reuse code to a greater extent but, where practical, each operation and account property was given its own specific function in python. This improves code readability and also ensures that future extensions to any of the pre-existing functions do not require excessive refactoring of the code base.

3.3 ANTLR

The previous section covered the creation of individual components in Neo4j but to develop an interpreter these must be combined to process the tactic language.

The parser generator ANTLR (ANother Tool for Language Recognition) [20] was used to produce a parser, lexer and visitor for the purpose of processing the tactic language. A lexer splits input up into tokens and a parser processes combined tokens into a parse tree. This can be demonstrated by the following example of lexing and parsing the sum “437 + 734” from the ANTLR Mega Tutorial [26]:

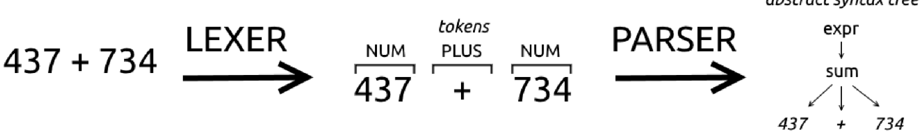

Figure 3.2: Lexing and Parsing Visualisation [26]

ANTLR has two different methods of walking a parse tree that is produced: visitors and listeners. Visitors are more complex but provide control over which branches of the parse tree that they walk and can return values, whereas listeners are simpler to implement but always walk every branch of the parse tree and cannot return values. The choice of which was used for this interpreter shall be discussed in a subsequent section.

ANTLR is another tool that was originally created for Java but has a supported Python implementation [20]. Similarly to Neo4j, the choice was made to implement all of the ANTLR functionality using Python as it improves readability and compatibility if the parse tree code and code for interacting with the database are both written in Python. As it was originally Java based, all of the relevant keywords and functions in ANTLR are written in CamelCase or camelCase. Therefore, to ensure code conformity, all of the novel parsing and interpreting syntax written used this standard. The main exception to this will be during the testing phase which makes use of pytest. Pytest uses underscores for default features so this style was followed for the test scripts.

Originally, it was planned to have every aspect of the tactic language carried out on a single walk of a parse tree for the language. However, this was decided to be inefficient. The reason why can be seen with a simple example: if a full left branch of the parse tree contained many functions and then at the very top there was “ ; $\bot$” then, by the defined rules for the simplified tactic language, none of those operations should be carried out and any processing of them would have been for nought. Therefore, it

makes more sense to keep a list of operations that need to be carried out which can then be processed individually at the end of a parse tree walk, if it is not null. The solution decided upon was to create an ANTLR grammar for the checks stage (which was then used to created a parser and lexer) and a separate python script for the updates stage. These two parts match the two big-step semantics defined previously. ANTLR files create and then walk the parse tree provided by the tactic language, recording the operations from the correct branches by $\top$ and $\bot$ tactic rules. The updates file is then run using the list of appropriate operations that was recorded in the prior stage.

3.4 Tactics Grammar

The tactics grammar draws extensively from the grammar provided by Arnaboldi et al. [2] for the original tactic language. AND is the lexing rule for ';', OR is the lexing rule for '||', and t_single contains the rules for the rest of the grammar. The only notable addition is brackets to ensure that a user can specify the order of operations.

```
tactic: t EOF;

t:
    t AND t
    | t OR t
    | '(' t ')'
    | t_single;

t_single: alpha | b | CHECK '(' phi ')';
```

These are the topmost parser rules for the grammar, the rest of the parser and lexer rules can be found in Appendix B.3. When defining the grammar, a design decision had to be made about precedence. Arnaboldi et al. [2] are very abstract in their paper and do not make a clear choice about the order of precedence. Therefore the choice was made to interpret the language with ';' (AND) binding more strongly than '||' (OR). This is similar to conjunction and disjunction which are already aspects of the language due to the atomic propositions so it may cause confusion if these were defined otherwise. Precedence is very straightforward to implement in ANTLR, it is decided by the order in which rules are defined in the grammar. As AND is defined before OR above, then it has higher precedence. Brackets still bind more strongly than AND and OR because

the parser will match the open bracket before matching ‘t’.

The grammar is defined in a left recursive manner as the first non-terminal variable is always on the left. This would not have been possible with ANTLR v3 but fortunately ANTLR v4 handles this well. Left recursion is transformed and optimised behind the scenes but the produced parse tree is still visualised as one would expect the left recursive grammar to look [19].

Even though operations are not carried out by the parse tree that is produced by this grammar there are still relevant lexer and parsing rules for them. This allows the content to be easily recorded in a list that contains tuples of function names and inputs cleanly separated for the updates to follow. This also provides a basis for future work if it is decided to refactor the code so that everything is carried out using a single parse tree.

Using the defined grammar, parse trees can now be produced from any input that conforms to the rules of the tactic language. For example the tactic:

$$disc_access_(\text{Adversary,V_Locked_Phone}); disc_access_(\text{Adversary,V_Phone}); CHECK(\text{is_account(V_Phone)}) || TOP$$

Has the corresponding parse tree:

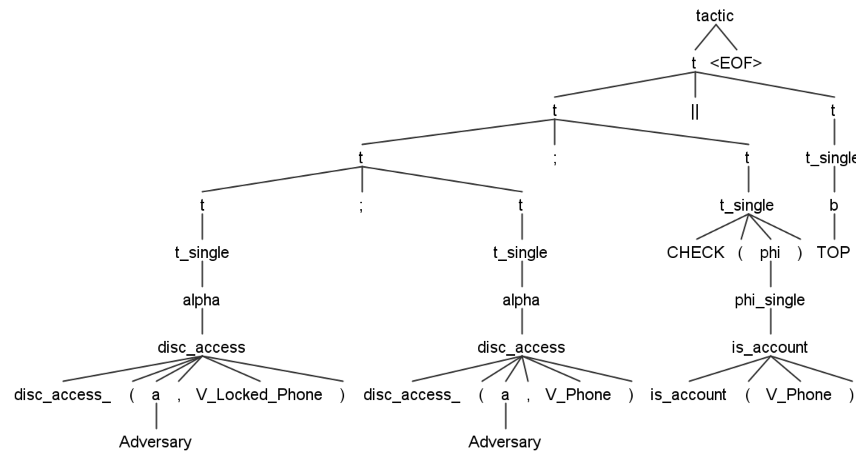

Figure 3.3: Example Tactic Parse Tree

3.5 Visitor

As mentioned, due to the structure of the simplified language, it is not possible to know what operations should be carried out until one has reached the top of the parse tree. This creates the interesting problem of how to store the operations found in different

branches of the parse tree while walking it. Take the following example, say t_1, t_2, t_3 and t_4, are all α with relevant operations to carry out. Then if t_5 returns $\bot$ this requires the removal of the operations t_3 and t_4 from the data structure without removing the operations t_1 and t_2:

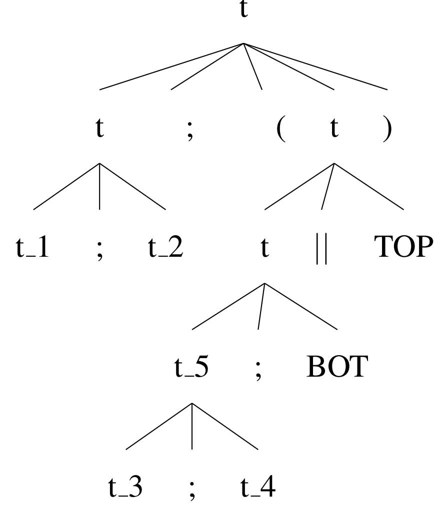

Figure 3.4: Operations Storage Problem

Initially a stack was considered so that items could be popped as necessary, however it is difficult (though not impossible) to process how many items should be removed dependent on the depth of the current branch of the tree. The simpler solution that was decided upon was to have each instance of a node in the parse tree return a function list and a Boolean value. The function list stores the relevant operations from that branch of the tree (often empty) and the Boolean value represents the overall tactic value of said branch of the tree, be that “True” corresponding with $\top$ or “False” corresponding with $\bot$. This means that each node has a list of functions and a Boolean, then based on its sibling nodes there is logic to delete the list of functions or return it to the parent node. This design choice is simpler than a global stack and allows for easier maintenance as any bugs can be narrowed down much more easily with many individual return statements than with a global stack that is constantly updated.

For this reason, and the following, a visitor was used for walking the parse tree, despite being more complex to implement than a listener. Since the simplified tactic language contains clauses that may be irrelevant depending on other parts of the tactic, it is not necessary to visit every branch of the parse tree. Therefore, the efficiency of the parse tree is greatly increased by using a visitor that includes logic to check whether a branch of the tree needs to be visited before visiting it.

The following simple example highlights this efficiency. One can see that multiple large branches of the tree are not visited because it is known, based on the visited nodes, that no values from those branches will be used.

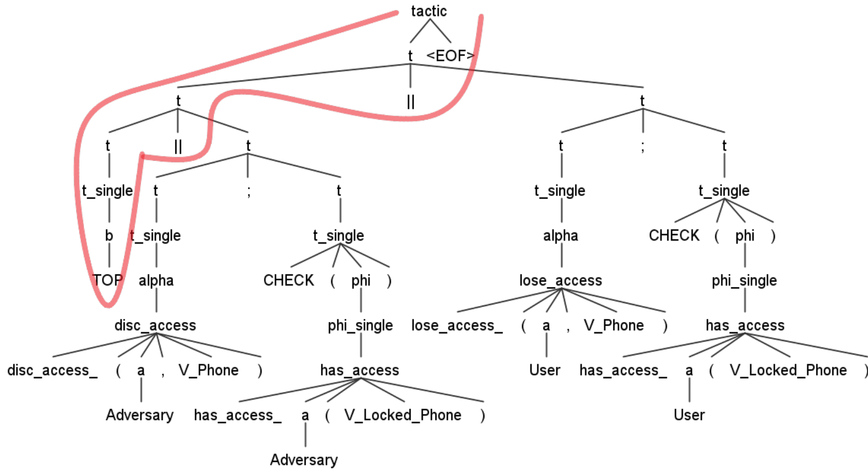

Figure 3.5: Visitor Branching Example

3.6 Graph Updates

After completing traversal of the parse tree, the visitor returns a list of functions (or operations) and a Boolean value. The graph updates class is only instantiated if the functions list is non-empty and the returned Boolean value is true. This design decision increases efficiency because always instantiating the class would cause the interpreter to create an instance of the Neo4j Driver to attempt to update the graph in unnecessary situations, which would be a waste of resources. When the function list is not empty and the returned value is true, then the graph updates class is instantiated, with the relevant database details, to execute the operations. The functions list is then passed to a particular method within the produced graph updates object which iterates over each tuple in the list invoking the stored function with the matching inputs.

3.7 Neo4j Parser Integration

Neo4j is integrated into both the visitor and graph updates class that have just been covered. Both files start with opening a Neo4j driver which connects to the relevant database. The visitor uses a read session for every visit to a 'CHECK' node and connects using the python function which contains the relevant Cypher query. This query is

carried out and a Boolean value is returned equivalent to the result. This value is then used appropriately as the tactic value $\top$ or $\bot$ for that node of the parse tree.

For the updates file, a read session is opened for checking any premises for the operational semantics. If the premises for an operation fail then an error is printed and the operation is not carried out, the program is not interrupted and moves on to the next operation in the list. This is, again, another design decision that has been made to avoid backtracking. Once the Neo4j database is written to, there is no inherent way to revert updates so it was decided that greater functionality was provided by checking premises individually and failing on a by-operation basis than failing the whole tactic. If the operation premises pass, or there is not a relevant premise, then a write session is opened and the operation is carried out. This is accomplished by calling a python function containing a Cypher query that interacts with the Neo4j database.

With the integration of Neo4j, this report has now covered the complete implementation of the interpreter. It returns the tactic for success or failure ($b \in \{\top, \bot\}$) in terms of the entire tactic along with the list of operations that it tried to apply. Any errors caused by the operational semantics premises failing are returned and it is specified which operations were not carried out because of this. Feedback is also automatically provided by ANTLR if the tactic input fails to be parsed in any way. Also, due to the Neo4j sessions, feedback is provided for any database connection issues. These factors ensure that a user is provided with detailed guidance should they encounter any issues when trying to use the interpreter.

Chapter 4

Evaluation and Testing

4.1 Evaluation with Case Studies

The following case studies demonstrate that the interpreter, even using the simplified tactic language, is effective at modelling a range of attacks. The studies provide formal descriptions, using the tactic language, of the relevant attacks and defences but they also provide intuitive visuals. These visuals effectively communicate the core concepts in a simple manner, making them accessible even to a non-technical audience. All of the graph visualisations (including colouring) were created automatically by Neo4j, the sole manual part was rearranging the nodes to increase clarity. This is far less cumbersome than the only prior option, creating the graphs by hand for every example. This provides researchers, whose time is very limited, with a more efficient process.

4.1.1 Initial Success

During initial testing of the system, the examples provided by Arnaboldi et al. [2] were processed to ensure correct functionality. However, upon carrying out the operations to transition from Fig 2 (b) to Fig 2 (c) in their paper, the system repeatedly returned false for the satisfaction of the operational semantics premises.

The tactic in question (translated into the syntax for the interpreter) is:

$$rem_access_1_{\text{Adversary,V_SIM,E_PHYS}} ; rem_access_2_{\text{Adversary,U_SMS,E_DISPLAY}} ; add_access_{\text{Adversary,\{U_Second_Phone\},V_SIM,E_PHYS}} ; add_access_{\text{Adversary,\{U_Second_Phone,U_SIM\},V_SMS,E_DISPLAY}}$$

Entering the above tactic into the interpreter (and connecting to a Neo4j instance

containing the relevant graph) provides the output:

"add_access_(Adversary, U_Second_Phone,U_SIM, V_SMS, E_DISPLAY) premise rule check failed, function not carried out."

Upon close inspection it was discovered that the operational semantics premise of the adversary having access to the SMS vertex was not satisfied and so the relevant *add_access* operation could not be carried out. This highlights the utility of this interpreter and how it can aid researchers. Future novel tactics can be processed using this system to ensure that they are valid and that no rules are accidentally broken.

4.1.2 Twitter Compromise

(Twitter has recently re-branded to 'X' [5] but shall be referred to as Twitter throughout this report as that was how it was initially written).

In the Mat Honan hack [12] multiple accounts were compromised solely for the purpose of obtaining access to a Twitter account. This shows how valuable social media access, and especially Twitter, can be to certain motivated adversaries. As an example that is applicable to the majority of social media then, this report shall use account access graphs to visualise some of the ways in which a Twitter account could be compromised.

Seen below is an initial account access graph representing a Twitter user with the Twitter app installed on their phone. The state of the graph is immediately after the user has had their PIN observed and their locked phone stolen:

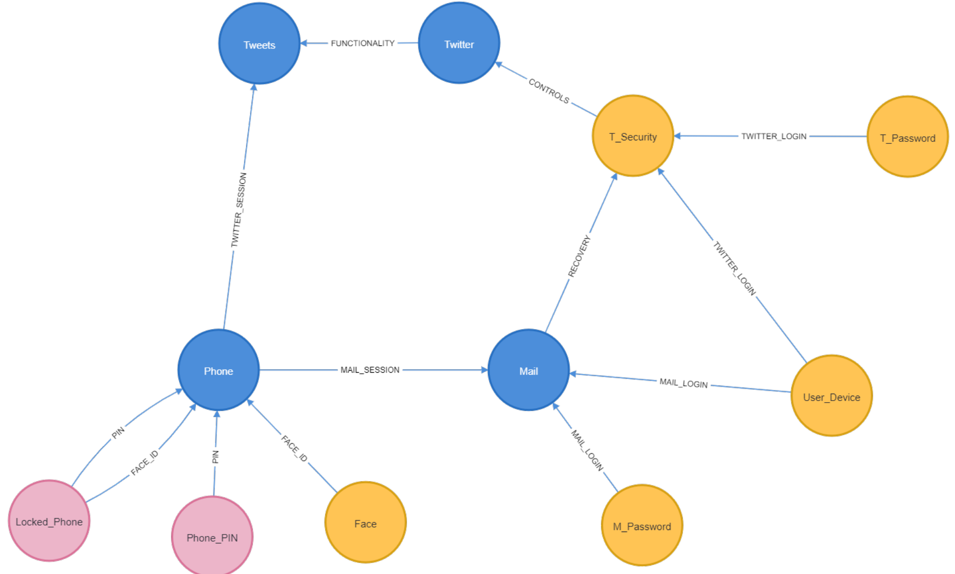

Figure 4.1: Stolen Phone with Twitter Account

Similarly to the iPhone attacks [25], it was assumed that the adversary has been close by to the user, observed them type in their PIN number and has subsequently stolen their phone.

The adversary can now access the unlocked phone via the 'PIN' access method and they are able to get into the Twitter app via the session access method. The 'SESSION' access methods (for Tweets and Mail) represent the fact that many applications do not sign the user out automatically so the user can avoid the inconvenience of entering login details. The following tactic represents this series of events:

$$gain_access_{\text{Adversary,V_Phone}} \ ; \ gain_access_{\text{Adversary,V_Tweets}}$$

Entering the above tactic into the interpreter and then using Neo4j visualisation we can see the updated state of the graph:

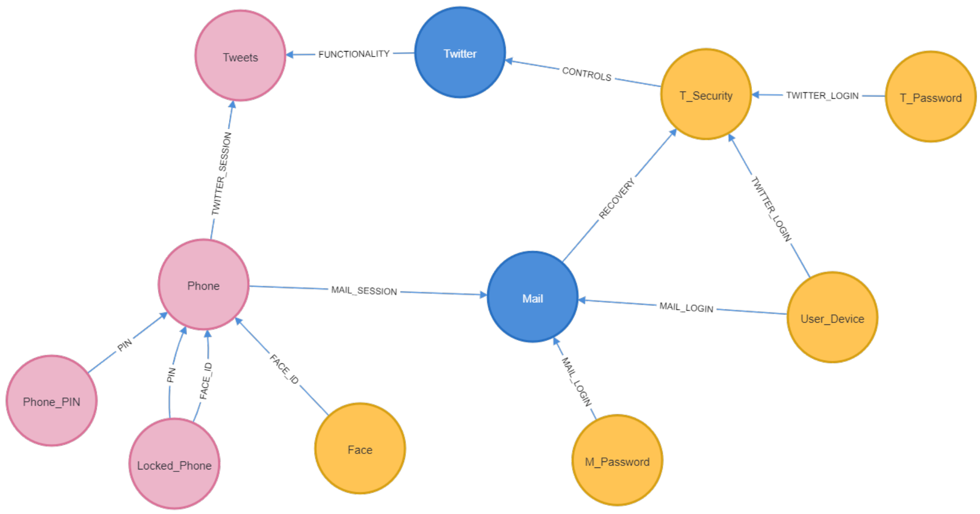

Figure 4.2: Adversary Gained Access to Tweets Functionality

Now the adversary has the ability to send Tweets but they can't yet alter security settings, represented in this graph by 'T_Security'. At the current moment, the victim could utilise another device, represented in the graph by 'User_Device' and access the security settings in order to remove the adversary from the Twitter account. However, the assumption is made that the phone the adversary has stolen has a mail app connected to the Twitter recovery email account. This can be used to recover the password associated with the Twitter account, which then gives the malicious actor access to the security settings. Security settings access can be used to modify the password and remove the access of the old password. This is accomplished by the creation of a new password account, the creation of the adversaries' own device, the removal of the access

previously held by the user's password and device, and then finally adding the access to the adversaries' password and device. All of these steps can be described by the following tactic:

$$\begin{aligned} gain_access_{\text{Adversary,V_Mail}} \ ; \ gain_access_{\text{Adversary,V_T_Security}} \ ; \\ create_account_{\text{Adversary,V_Adv_Password}} \ ; \ create_account_{\text{Adversary,V_Adv_Device}} \ ; \\ rem_access_1_{\text{Adversary,V_T_Security,E_TWITTER_LOGIN}} \ ; \\ add_access_{\text{Adversary,\{U_Adv_Password,U_Adv_Device\},V_T_Security,E_TWITTER_LOGIN}} \end{aligned}$$

This was then entered into the interpreter, which produced no errors and updated the Neo4j account access graph to the following:

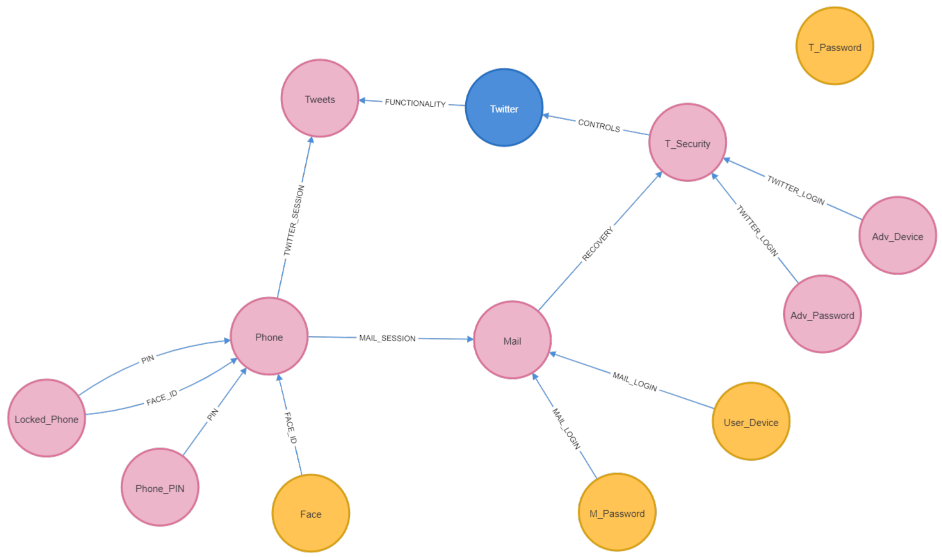

Figure 4.3: Adversary Replaced Official Access Methods with Their Own

The malicious actor has now obtained complete control of the user's Twitter account security and its access methods. The only hope the user has of saving their account is by signing into their recovery email. The user still has the appropriate access methods to gain access to the relevant vertex (the password and device can be seen in the bottom right of the graph). However, even this might not be enough, as the adversary has full control of the Twitter account security settings. They have the ability to change the email address (remove access to the user's email) which would leave the user with no access methods remaining and the adversary with uncontested control of the Twitter account.

In contrast, if the user responds with haste from the beginning then they would be able to prevent this account takeover. Consider the initial graph again, Figure 4.1, the

user simply has to access the 'Mail' node with their backup device and mail password (by gain_access). They can then remove access for the phone node which protects their mail and their Twitter security settings. This can be done by the following tactic:

$$gain_access_{\text{User,V_Mail}} \; ; \; rem_access_1_{\text{User,V_Mail,E_MAIL_SESSION}}$$

Which entering into the interpreter acting on Figure 4.1 gives:

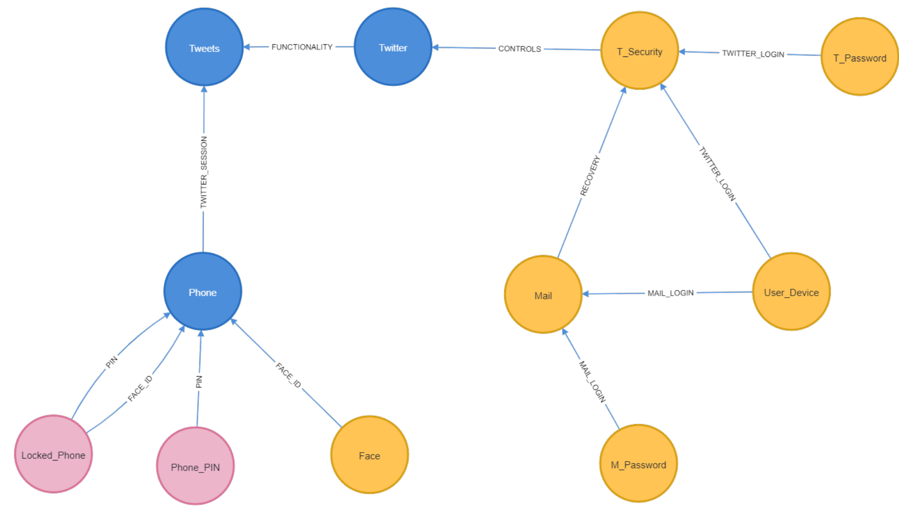

Figure 4.4: User Maintains Control of Mail

The adversary is now unable to access the Mail vertex. As expected, if the tactic $gain_access_{\text{Adversary,V_Mail}}$ from above is entered then the interpreter produces the following output: *"gain_access_(Adversary, V_Mail) premise check failed, function not carried out."*

The Twitter exploit covered above is equally valid for the vast majority of social media applications. Even the addition of two factor authentication, either email or SMS, would not be beneficial because both accounts are already in control of the adversary.

The underlying problem here is not limited to phone and PIN theft. There is growing criticism that social media apps provide too much control to devices that are already signed in to accounts. Take Linus Tech Tips, a YouTube channel with over 10 million subscribers who had the session cookie for their account compromised earlier this year. This led to a fraudulent cryptocurrency scheme being live-streamed and advertised to their sizeable audience [21]. The relation is similar to the usability versus security problem Apple is facing, mentioned in the introduction. All of these applications want to provide as enjoyable a user experience as possible, but this comes with considerable security risks that must be balanced.

This is made exceptionally clear by the number of possibly open accounts on a single device, especially considering this account access graph only investigated the damage caused to a single account. A similar attack could be launched on multiple social media applications simultaneously as soon as the phone is stolen. The critical point highlighted by this case study is that not just a single account session/token must be considered, but that the combination of sessions and tokens that a single compromised device has can lead to immense control and exploitation.

Another critical takeaway from the above examples is that even if Apple resolve the overwhelming power that an iPhone and PIN combination have on the Apple ecosystem, the “improved” state shall still have some serious security concerns.

4.1.3 Banking App Compromise

While the banking app exploitation covered by the Wall Street Journal involved access to Apple’s password management system [25], there is an equally dangerous problem that is vastly more difficult to patch: human nature. Password reuse is incredibly common [6], thus it is likely that a number of users make use of the same PIN for their phone as they do for their banking application. Two banking apps were tested by this report and they made use of 4 digit and 5 digit PIN protection respectively [27, 4]. The entry for banking PINs used a very similar user interface to entering a phone PIN, reinforcing the idea that password sharing is likely. This weakness can be modelled using account access graphs:

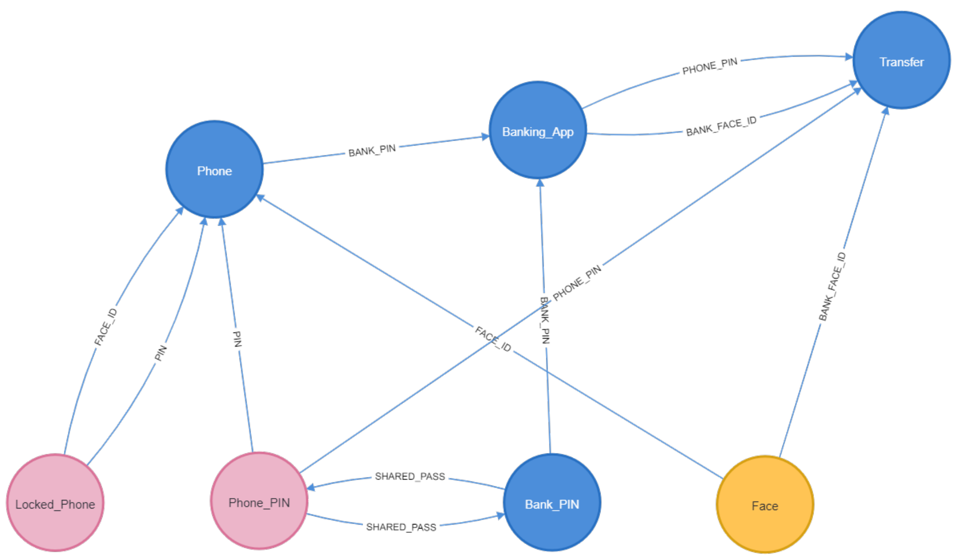

Figure 4.5: Shared Password Mobile Banking

Again, this graph has a similar setup as previously, in terms of the three access methods for the phone and that the adversary has acquired the locked phone and the user's PIN. The distinction in this scenario is that the Banking App should be safe from the attacker because it has a separate PIN from the phone. However, as can be seen in the graph, the user of this device has used the same PIN for both the phone and the bank. This is represented by a back and forth relationship with no joint access methods, which means that if either PIN is compromised the other can be immediately gained access to as well.

As before, the adversary can gain access to the phone, following which they can easily gain access to the bank PIN due to the 'SHARED_PASS' relationship. Using the control of these two accounts, the adversary can then gain access to the mobile banking application, this series of steps can be written as the following tactic:

$$gain_access_{\text{Adversary,V_Phone}} \ ; \ gain_access_{\text{Adversary,V_Bank_PIN}} \ ; \ gain_access_{\text{Adversary,V_Banking_App}}$$

Interpreting this tactic provides the following Neo4j output:

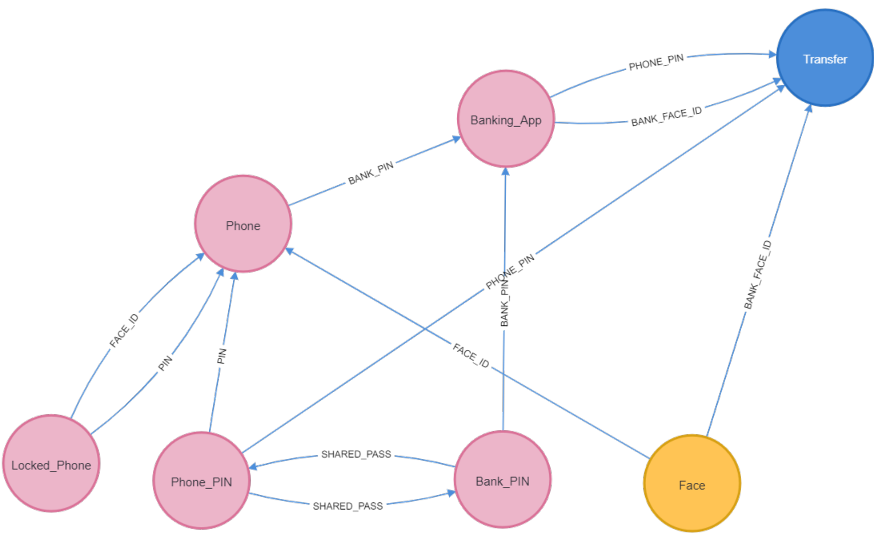

Figure 4.6: Adversarial Access to Banking Application

Ironically, the safety mechanism intended to provide further security against unwanted transfers of money simply asks one to enter the relevant phone PIN or use biometrics/faceID. However, it can be seen from the graph that this extra step provides no further protection against an adversary that has been able to compromise the banking app. This could be a case of security theatre where the bank app is trying to convince users of extra security that is, in reality, meaningless. Designers of bank apps should

be careful however, that they are not causing disruption to the user without purpose as this may make users less likely to choose said app. If the use of biometrics is for safety, rather than security, perhaps a warning and tick box would be more appropriate.

4.2 University of Edinburgh: Multi-Factor Authentication

The University of Edinburgh provides managed laptops to staff and students that are accessed by the federated single sign-on service MyEd. An example setup can be represented with the following account access graph:

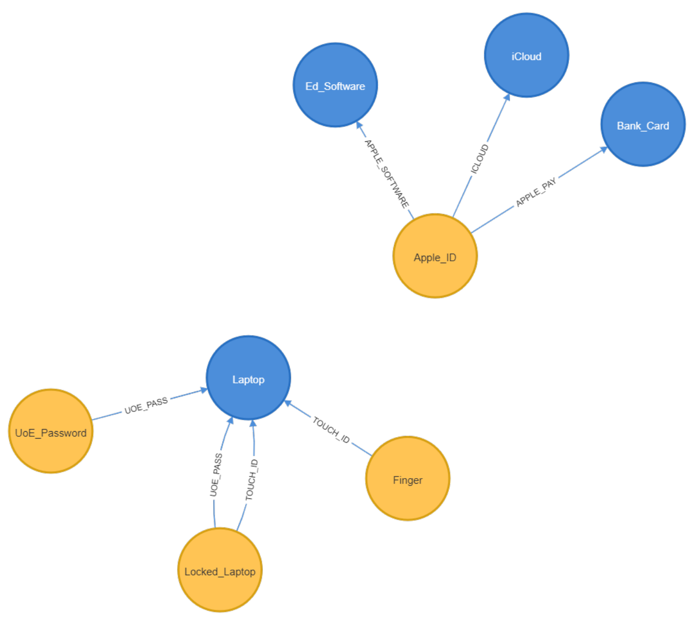

Figure 4.7: UoE Managed Laptop Setup

If the user needs to be able to access the education software ('Ed_Software' in the above graph) to continue their research then this encourages the connection of their Apple ID to their university laptop. Since the user already has access to their Apple ID, they can add an access method and then gain access to their required software through their university managed laptop:

$$gain_access_{\text{User, V_Laptop}} \; ; \; add_access_{\text{User, \{U_Laptop\},V_Apple_ID,E_LAPTOP_TO_APPLE}} \; ; \; gain_access_{\text{User, V_Ed_Software}}$$

Entering this into the interpreter provides the visual representation:

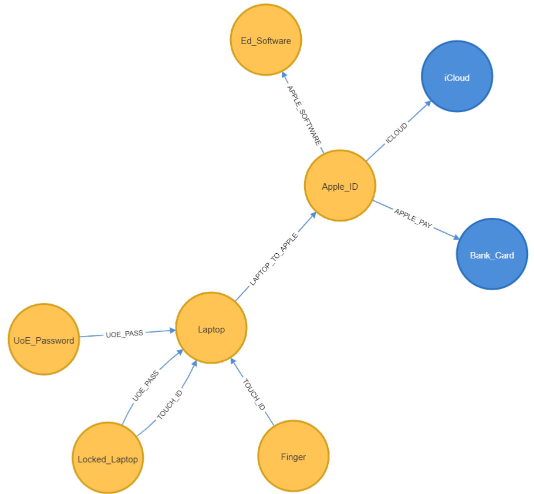

Figure 4.8: UoE Managed Laptop Apple ID Access

However, a critical vulnerability of this setup is if an adversary discovers access to the locked laptop and the user's UoE password then they can gain access to the AppleID and thus the 'Bank_Card' and 'iCloud' vertices through the following tactic:

$$disc_access_{\text{Adversary, V_UoE_Password}} \; ; \; disc_access_{\text{Adversary, V_Locked_Laptop}} \; ; \; gain_access_{\text{Adversary, V_Apple_ID}} \; ; \; gain_access_{\text{Adversary, V_iCloud}} \; ; \; gain_access_{\text{Adversary, V_Bank_Card}}$$

Even if there are security measures to prevent the comprise of iCloud in this method, if the laptop has a signed in email address then a similar take over method to the Twitter example could be possible.

The University of Edinburgh is introducing Multi-Factor Authentication [18] with personal devices which should be able to prevent the above attack. The laptop connected to the Apple ID would now have an extra device edge that shares a label with the UoE password. The creation of this device and addition of this edge can be represented with the tactic:

$$create_account_{\text{User, V_Auth_Device}} \; ; \; add_access_{\text{User,\{U_Auth_Device\},V_Laptop,E_UOE_PASS}}$$

This way, if the adversary tries to perform the exact same tactic to compromise the bank card and iCloud as above, then the interpreter outputs:

gain_access_(Adversary, V_Apple_ID) premise check failed, function not carried out. gain_access_(Adversary, V_iCloud) premise check failed, function not carried out. gain_access_(Adversary, V_Bank_Card) premise check failed, function not carried out.

The extra authentication method is preventing the adversary from gaining access to the laptop so they can't use that vertex to gain access to any further accounts. This stresses the effectiveness of account access graphs. They are most powerful when describing the weaknesses that come from a chain of access methods that could easily be overlooked.

These case studies have demonstrated the strength of the interpreter's ability to process tactics and update graphs in different situations. They have also displayed the strong advantage given by the automatic visualisation provided by Neo4j which allows for an understanding of complex access method relationships at a glance.

4.3 Testing

Testing was carried out using the pytest framework to ensure the system functioned as expected in a diverse range of situations. The visitor and graph updates were tested individually to allow for modular testing and verify that each file operates independently. All tests were carried out using the tactic language; this supported testing the parser's ability to process syntax while simultaneously checking the Neo4j component, all within the same Python script.

4.3.1 Testing the Visitor

A visitor test script was created and partitioned into different sections to carry out different tests for each aspect of the language.

First, the environment for running the tests had to be initialised. To ensure appropriate Neo4j query execution, the setup for the test module involves wiping and creating a new Neo4j database from scratch, every time tests are run. The basis for this database is a simple account access graph defined in the setup portion of the test script. The database must be wiped regardless of which test or tests are run. This is to ensure the correct conditions for any test, or combination of tests, being run in the module. This creates the interesting quandary of how to guarantee that a test database only contains a simple user account access graph, no matter which test is run but also avoid having to wipe and build such a database for every test when running the full module. Fortunately, pytest fixture functions contain an optional setting to set the scope to be module wide. This results in a function that will only be run once per module but can be called many times. Once it is run, it saves state and does not run again but provides the expected

output to any functions that call it. Therefore, each individual test can call this function but the driver will only be created once for each module run. This greatly increases performance, as shall be shown subsequently.

Each test constituted setting up the lexer, parser, visitor and environment in the same manner as they would be within the interpreter. A helper function was defined that accomplished this. Other than calling this helper function, all that comprised each test was using the produced visitor on the produced tree, both from the helper function, and then verifying the results that were produced by the visitor. To verify the results of many different inputs, the parameterisation features of pytest were heavily utilised for these tests. For each test, a series of inputs were run which were then compared to the expected outputs. Pseudocode for the test file can be seen below (the test_inputs were handled using pytest parameterisation rather than a loop):

```
# graph setup is run once per module,
# despite being called by every function
setup_graph:
    Neo4j write query to wipe database instance and
    setup a test graph

setup_visitor(details):
    Create visitor based on input details

test_inputs:
    list of test inputs and expected outputs
test_atomics:
    setup_graph
    for test_input in test_inputs:
        tree, visitor = setup_visitor(details)
        result, functions = visitor.visit(tree)
        assert result = expected_result
        assert functions = expected_functions
...
# multiple test_inputs and tests, as above and below
...
test_inputs:
    list of test inputs and expected outputs
```

```
test_all_combined:
    # as for test_atomics
```

The outputs that were tested, equivalent to that of the visitor in the interpreter, were a Boolean value and a list of functions. The outputs were compared to the list of expected output Booleans and function lists which were associated with the relevant input (all contained within the pytest parameterisation) to decide whether a test passed.

While the test functions are, in reality, identical and it would have been possible to have all the tests called by a single parameterised list on a single function, this would make it extremely complex to debug if there was an issue. Instead, as tests were segmented into specific subsets of the language, if tests were to fail, the person debugging could easily tell exactly what aspect of the language was causing the problem.

The first aspect to be tested were the atomic propositions, these were wrapped in the CHECK() syntax of the tactic language. Each proposition was ran in a situation where it should return true and a situation where it should return false. It was also confirmed that the visitor returned an empty list of functions for each of these checks.

Then the atomic propositions were combined with the AND and OR features of the tactic language along with $b \in \{\top, \bot\}$ to confirm successful tactic combinations.

The list of α (operations) of the language were tested in a similar manner but with a non-empty expected output function list, unless functions should be cancelled out in the tactic. Edge cases, including the one visualised in Figure 3.4, were tested to ensure that the appropriate functions were always returned. Tests that incorporated all tactic aspects: atomic propositions, $b \in \{\top, \bot\}$, and α, all combined using AND and OR, were also carried out to conclude the testing.

As covered in Chapter 3, the visitor does not implement any of the operations in the graph but rather returns the relevant list of functions for the updates phase. Consequently, no updates to the graph were tested in this module, which helps the separation and maintainability of the code as the testing has modularity built in.

Overall, 74 individual tactics were written to test normal functionality and edge cases for the visitor alone, all of which passed.

As previously mentioned, instantiating Neo4j Driver instances is known to be resource intensive [13]. This can be measured by the fixture function scope optimisation addressed in the beginning of this section. If the part of code that changes the scope is removed and the module of tests is run, it takes 482 seconds or 8:02 minutes to complete. Whereas with the specification in place, running the entire test module takes 181 seconds or 3:01 minutes. This is a 62.45% performance increase. This highlights

the importance of the effort made to minimise new driver sessions throughout the interpreter and the efficiency obtained by doing so.

The visitor testing verifies that the interpreter is fully functional at parsing the tactic language and that the AND and OR logic is correct. It also shows that the Neo4j language can be interacted with successfully with read queries while walking the parse tree to return a Boolean which allows for an accurate interpretation. Additionally, it has been verified that the correct list of functions is returned to the system based on the parse tree. The following section shall finalise the testing by confirming the write operations are functional.

The overall visitor aspects tested are represented in the following table: (A few additional test cases did not fit so were not included in the tables in this chapter but all of which passed and every tick (check mark) in the table represents a genuine successful test).

Aspect of Language	Subsection	Tests			
		1	2	3	4
Atomic Propositions	is_account(v)	✔	✔		
	has_access_a(v)	✔	✔	✔	
	could_access_a(v)	✔	✔		
	uses_method_l(u,v)	✔	✔	✔	
	$\phi_1 \wedge \phi_2$	✔	✔	✔	✔
	$\phi_1 \vee \phi_2$	✔	✔	✔	✔
	$\neg\phi$	✔	✔	✔	✔
	Mixed Atomics	✔	✔	✔	✔
$b \in \{\top, \bot\}$	TOP	✔			
	BOT	✔			
	Mixed	✔	✔	✔	✔
	b with Atomics	✔	✔	✔	✔
α List	Operations (1-4)	✔	✔	✔	✔
	Operations (5-8)	✔	✔	✔	✔
	Multiple Operations	✔	✔	✔	✔
	Operations with Atomics	✔	✔	✔	✔
	Operations with $b \in \{\top, \bot\}$	✔	✔	✔	✔
	Operations with Atomics and b	✔	✔	✔	✔
		✔	✔	✔	✔

4.3.2 Testing the Graph Updates

The graph updates testing followed a similar principle to the visitor testing but the implementation had some major differences.

The critical tasks for the graph updates testing involve updating the access of participants and the shape of an account access graph. To confirm that this was successful, these factors require checking in some way. To accomplish this, the CHECK() aspect of the tactic language was used. This means that for the graph updates testing to be valid, all of the tests for the visitor must first succeed.

Since the state of the graph was not updated in the visitor testing, the pytest script only had to write to the Neo4j database once, at the very start of testing. This is not the case for the graph updates, as to allow for functions to be called independently it is essential that there is a standard tactic graph for each test. This was accomplished by two pytest fixture functions: the first is run once per module whereas the second is run before every function. Both are defined to run automatically. The first function defines the Neo4j driver instance which remains open while the module testing is happening and is closed after all testing is finished. The second function uses the driver that was initiated by the prior function to open an (efficient) Neo4j write session which wipes the test database and then creates a new account access graph before each test is run.

There were less variations of combination for this testing as only tuples containing functions and their inputs are contained in the functions list and they are separated by commas, not logical operators like AND and OR. This meant that only two tests were written, one to test each operation individually and another to test longer lists of functions. As with the visitor testing, pytest's parameterisation features were heavily utilised here. For each of the two tests a series of different inputs and expected outputs were entered, covering all possible operations and a wide range of possible combinations, as can be seen in the table below.

Initially, it was considered to import just the functions that check atomic propositions rather than using the visitor. However, this would have led to another series of tests to ensure that these functions were operating as expected. Additionally, use of the visitor allows for the testing to be carried out using the tactic language which maintains the consistency of the code. For these reasons it was decided to create an instance of the visitor and use the tactic language for testing. This requires further instances of the driver for testing than the alternative, creating further overhead for the tests. If future work requires the tests to be carried out as efficiently as possible then this is an

important area to be modified. However, when designing the tests the choice was made that greater benefit would be found through the use of the visitor to check the state of the graph. Thus, for each graph update function test, a visitor is also created to check it.

Overall, 37 lists of functions were run through the tests and all of them passed. The first set of inputs tested that all of the operations update state successfully when the operational semantics premises are satisfied and that they do not update state when these conditions are not satisfied. The second set of inputs tested the same points but additionally that the graph updates class can handle multiple operations being called consecutively. All of the relevant tests passed, the results and breakdown can be seen in the table below: (As remove_access_2 has the most complex premise to be checked, it underwent the most individual tests to ensure it was failing and passing appropriately.)

Operations	Subsection	Tests			
		1	2	3	4
Individual Operations	$gain_access_{a,v}$	✔	✔		
	$disc_access_{a,v}$	✔			
	$loses_access_{a,v}$	✔			
	$create_account_{a,v}$	✔			
	$del_account_{a,v}$	✔	✔		
	$add_access_{a,\{u_1,\ldots,u_n\},v,l}$	✔	✔		
	$rem_access_1_{a,v,l}$	✔	✔		
	$rem_access_2_{a,u,l}$	✔	✔	✔	✔
Multiple Operations	(GAIN, DISC, LOSE) Mixed Testing	✔	✔	✔	✔
	(CREATE, DEL) Mixed Testing	✔	✔	✔	
	(ADD) Mixed Testing	✔	✔	✔	
	(REMOVE1, REMOVE2) Mixed Testing	✔	✔	✔	✔
	All Operations Mixed Testing	✔	✔	✔	✔
		✔	✔	✔	

This chapter has shown that the interpreter is functional for a wide range of possible tactics using the simplified tactic language. It is able to parse the relevant grammar in normal circumstances and edge cases. It is also able to update the graph with all of the operations defined by Arnaboldi et al. [2] while adhering to their premises.

The defined tests provide an advantage for future work and extensions. As, now that the unit tests have been created, they can be run any time the source code for the interpreter is modified to ensure that no bugs have been introduced.

Chapter 5

Conclusion

5.1 Conclusion and Critical Evaluation

Overall, this project has effectively designed and implemented an interpreter capable of interpreting a simplified tactic language. This interpreter carries out short programs, referred to as ‘tactics’, in this language on user account access graphs. It subsequently checks, produces, or updates a visualisation of the relevant graph using Neo4j. This was accomplished through:

1. The creation of a simplified tactic language with precise definitions through an operational semantics.
2. The development of an ANTLR grammar (including defining precise syntax) for the simplified tactic language.
3. The development of appropriate Cypher queries to create, check, and update the state of an account access graph.
4. The implementation of an ANTLR visitor and a graph updates class to carry out the ‘checks’ and ‘updates’ stage respectively of the simplified tactics operational semantics.
5. The integration of the Cypher queries with the visitor and the graph updates class to interpret the defined tactic language into a Neo4j database with implicit visualisation.
6. Evaluating the interpreter through numerous case studies.
7. Writing a comprehensive test suite for the interpreter.

The testing showed that the interpreter can process all of the components of the simplified tactic language, individually and combined, without any errors. Additionally, the interpreter effectively used the simplified tactic language to: catch errors in graphs, model attacks on account networks and model potential defences against attacks. In summation, the above achievements are clear indicators of the success of the project.

5.2 Challenges Faced in the Project

The completion of this project required overcoming multiple challenges, predominantly stemming from areas that I was previously unfamiliar with. These areas included:

- Graph databases and databases in general
- Query languages
- Logic and operational semantics
- Parsing and parser generators
- The Python programming language
- Pytest and software testing
- Interpreters

Despite my initial lack of experience in these domains, I was motivated to select this project by the current relevance of account security and a strong passion for security overall. Tackling the steep learning curve, I defined an operational semantics for a simplified tactic language, devised a way to parse it, provided integration with a graph database using the Cypher query language, and combined these aspects to create an interpreter that was implemented and tested in Python. The success of this project proves that the choice of topic was appropriate and that the many challenges were effectively overcome.

5.3 Limitations and Future Work

One limitation that time constraints prevented this report for incorporating is shorthand notation for the tactic interpreter. Arnaboldi et al. [2] suggest letting **IF** ϕ **THEN** t_1 **ELSE** t_2 stand for $(\text{CHECK}\phi \,;\, t_1) \,||\, (\text{CHECK}\neg\phi \,;\, t_2)$ along with other useful shorthand notation.

Due to time constraints and the fact that the language is able to address all relevant graphs without this shorthand, it was omitted. Future work could extend the interpreter to provide this enhanced usability.

Another limitation of the work is the speed of the tactic processing. Despite the optimisations of minimising the creation of a new Neo4j Driver, the speed of communication with the database is still slow. Future work could investigate this and see if implementation in another programming language, using one of the other officially supported languages, leads to an increase in performance.

The creation of a compiler which automatically compiles a tactic into Cypher code is another avenue that future work could investigate. This report initially considered compiling instead of interpreting but it was decided that the use of a higher level language, like Python or Java, that runs multiple individual queries would be simpler and more readable than compiling a tactic into one very long Cypher query. Error handling and testing is also much simpler to implement in an interpreter than a compiler which is a strong benefit of this implementation. Additionally, this interpreter could be used in conjunction with any future work to create a compiler as it provides a basis for the parsing and error handling. It may also be the case that compiling is not a useful extension for this domain as the main resource cost is associated with establishing the connection to the Neo4j database, something a compiler would also have to do. In addition, interpreters not only allow for simpler testing but also provide increased usability compared to compilers, a very important aspect considering one of the goals of this project was to reduce the effort involved with producing graphs.

Finally, a key limitation is the fact that only a simplified tactic language was implemented for this interpreter. Future work could build upon this and investigate ways to implement backtracking so that this interpreter could be extended to interpret the full language defined by Arnaboldi et al. [2]. The atomic propositions as well as the individual functions to carry out each operation (all of the Cypher queries) can be taken and are fully equivalent to the original language defined by Arnaboldi et al. [2].

This means that, while this work does not provide an interpreter for the full language, it does provide much of the work required to develop one, and thus provides a strong foundation to build this in the future. Furthermore, the case studies have proven that the simplified tactic language is already powerful enough to describe a range of attacks (as not all attacks rely on backtracking) showing that the interpreter is already useful in its current state.

Bibliography

[1] Ross Anderson. *Security Engineering: A Guide to Buidling Dependable Distributed Systems, Third Edition, Chapter 2*. Wiley, 2021.

[2] Luca Arnaboldi, David Aspinall, Christina Kolb, and Saša Radomirović. Tactics for account access graphs. Submitted for Publication, 2023.

[3] Brooke Auxier and Monica Anderson. Social Media Use in 2021. *Pew Research Center*, 2021.

[4] Starling Bank. Starling Banking App, 2023. URL `https://www.starlingbank.com/download/`. Mobile Application.

[5] Kate Conger. So What Do We Call Twitter Now Anyway? *The New York Times*, 2023.

[6] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The Tangled Web of Password Reuse. *Network and Distributed System Security (NDSS)*, 2014.

[7] Neo4j Documentation. Query a Neo4j database using Cypher, Accessed on: 2023-06-23. URL `https://neo4j.com/docs/getting-started/cypher-intro/`.

[8] Neo4j Documentation. Neo4j Browser, Accessed on: 2023-07-09. URL `https://neo4j.com/docs/browser-manual/current/`.

[9] Neo4j Documentation. APOC user guide for Neo4j v5, Accessed on: 2023-07-13. URL `https://neo4j.com/docs/apoc/current/`.

[10] Neo4j Developer Guides. Cypher Style Guide, Accessed on: 2023-07-03. URL `https://neo4j.com/developer/cypher/style-guide/`.

[11] Sven Hammann, Saša Radomirović, Ralf Sasse, and David Basin. User Account Access Graphs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1405–1422, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3354193. URL https://doi.org/10.1145/3319535.3354193.

[12] Mat Honan. How Apple and Amazon Security Flaws Led to My Epic Hacking. *Wired*, 2012.

[13] Chris Leishman. 8 Solid Tips for Succeeding with Neo4j, 2016. URL https://neo4j.com/blog/8-tips-succeeding-with-neo4j/.

[14] David Meza. How NASA Finds Critical Data through a Knowledge Graph, 2017. URL https://neo4j.com/blog/nasa-critical-data-knowledge-graph/.

[15] Randall Munroe. Authorization, 2013. URL https://xkcd.com/1200.

[16] Mark Needham. Neo4j: Generic/Vague relationship names, 2014. URL https://neo4j.com/blog/neo4j-genericvague-relationship-names/?_ga=2.143116725.1062534417.1691180028-578698118.1691180028.

[17] Neo4j, Inc. Neo4j, 2023. URL https://neo4j.com/. Graph Database Management System.

[18] University of Edinburgh. The Rollout Explained, Accessed: 2023-08-13. URL https://www.ed.ac.uk/information-services/computing/computing-infrastructure/authentication-authorisation/multi-factor-authentication/rollout-explained.

[19] Terence Parr. *The Definitive ANTLR 4 Reference, Chapter 5, Page 72*. The Pragmatic Bookshelf, 2013.

[20] Terence Parr. ANTLR (ANother Tool for Language Recognition), 2023. URL http://www.antlr.org. Computer Software.

[21] Jay Peters. How hackers took over Linus Tech Tips. *The Verge*, 2023.

[22] Nathaniel Popper. Identity Thieves Hijack Cellphone Accounts to Go After Virtual Currency. *The New York Times*, 2017.

[23] Joe Roemer. Improving Patient Outcomes with Graph Algorithms, 2020. URL https://neo4j.com/blog/improving-patient-outcomes-algorithms-graphconnect/.

[24] Kate Rooney. Hacker lifts $1 million in cryptocurrency using San Francisco man's phone number, prosecutors say. *CNBC*, 2018.

[25] Joanna Stern and Nicole Nguyen. Apple's iPhone Passcode Problem: Thieves Can Ruin Your Entire Digital Life in Minutes. *The Wall Street Journal*, 2023. URL https://www.youtube.com/watch?v=QUYODQB_2wQ.

[26] Gabriele Tomassetti. The ANTLR Mega Tutorial. URL https://tomassetti.me/antlr-mega-tutorial/. Accessed on: 2023-07-09.

[27] Santander UK. Santander Banking App, 2023. URL https://www.santander.co.uk/personal/support/ways-to-bank/on-your-mobile. Mobile Application.

[28] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Neo4j in Action, Chapter 1*. Manning, 2014.

[29] Jim Webber. A Programmatic Introduction to Neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 217–218, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315630. doi: 10.1145/2384716.2384777. URL https://doi.org/10.1145/2384716.2384777.

Appendix A

Account Access Graph Background

A.1 Account Access Graphs Properties (Validity) [2]

The validity of assertions is defined inductively, starting from atomic propositions [2]:

$-\langle V,E,A\rangle \models \text{is_account}(v)$ if $v \in V$.

$-\langle V,E,A\rangle \models \text{has_access}_a(v)$ if $v \in V$ and $a \in A(v)$

$-\langle V,E,A\rangle \models \text{could_access}_a(v)\ if\ v \in V\ and\ \exists l,u\ .\ l \in E(u,v) \wedge \forall x \in V\ .\ l \in E(x,v) \implies a \in A(x)$

$-\langle V,E,A\rangle \models \text{uses_method}_l(u,v)\ if\ u,v \in V\ and\ l \in E(u,v)$

$-\langle V,E,A\rangle \models \phi_1 \wedge \phi_2\ if\ \langle V,E,A\rangle \models \phi_1\ and\ \langle V,E,A\rangle \models \phi_2.$

$-\langle V,E,A\rangle \models \phi_1 \vee \phi_2\ if\ \langle V,E,A\rangle \models \phi_1\ or\ \langle V,E,A\rangle \models \phi_2.$

$-\langle V,E,A\rangle \models \neg\phi\ if\ it\ does\ not\ hold\ that\ \langle V,E,A\rangle \models \phi.$

$-\langle V,E,A\rangle \models true\ always.$

Appendix B

Neo4j Code

B.1 Properties of Account Access Graphs Code

B.1.1 is_account(v)

```
:param v: 'vertex'

OPTIONAL MATCH (v:Account)
WHERE v.name = $v
RETURN
CASE
    WHEN v IS NOT NULL THEN true
    ELSE false
END AS `is_account(v)`
```

B.1.2 has_access_a(v)

```
:params {a: 'User/Adversary', v: 'vertex_v'}

OPTIONAL MATCH (v)
WHERE v.name = $v AND $a IN labels(v)
RETURN
CASE
    WHEN v IS NOT NULL THEN true
    ELSE false
```

```
END AS `has_access_a(v)`
```

B.1.3 could_access_a(v)

The proposition could_access_a(v) was more complex to implement than expected. There is no set minus function in Cypher and an alternate version that was in development was roughly equivalent to the initial code below.

```
:params {a: 'User/Adversary', v: 'vertex_v'}

MATCH ()-[r]->(v)
WHERE v.name = $v
//type(r) is used as a grouping key for count(*)
WITH type(r) AS accessMethod, count(*) AS requiredCount, v
MATCH (n_a)-[r_a]->(v)
WHERE $a IN labels(n_a)
//type(r_a) is used as a grouping key for count(*)
WITH accessMethod, requiredCount, type(r_a)
    AS accessMethod_a, count(*) AS aCount
//creates a table of the number of access methods and
//the number that 'a' has access to
WITH collect({required: requiredCount, aMethods: aCount}) AS
    actualAndRequired
//compares number of each access method that 'a' has access to by
//total for each method and if any are equal then
//returns true else it returns false
RETURN any(relation IN actualAndRequired WHERE
    relation.aMethods = relation.required) AS `could_access_a(v)`
```

B.1.4 uses_method_l(u,v)

```
:params {l: 'label', u: 'vertex_u', v: 'vertex_v'}

OPTIONAL MATCH (u)-[l]->(v)
WHERE type(l) = $l AND u.name = $u AND v.name = $v
RETURN
```

```
CASE
    WHEN l IS NOT NULL THEN true
    ELSE false
END AS `uses_method_1(u,v)`
```

B.2 Operations

B.2.1 gain_access_a,v

```
:params {v: 'vertex', a: 'User/Adversary' }

//First run could_access_a(v) and check that it returns true
MATCH (v)
WHERE v.name = $v
CALL apoc.create.addLabels(v, [$a]) YIELD node
RETURN v
```

B.2.2 disc_access_a,v

```
:params {v: 'vertex', a: 'User/Adversary'}

MATCH (v)
WHERE v.name = $v
CALL apoc.create.addLabels(v, [$a])
YIELD node
RETURN node
```

B.2.3 lose_access_a,v

```
:params {v: 'vertex', a: 'User/Adversary'}

MATCH (v)
WHERE v.name = $v AND $a in labels(v)
CALL apoc.create.removeLabels(v, [$a])
```

```
YIELD node
RETURN node
```

B.2.4 create_account_a,v

```
:params {v: 'vertex', a: 'User/Adversary'}

CALL apoc.create.node(['Account', $a], {name: $v})
YIELD node
RETURN node
```

B.2.5 del_account_a,v

```
:params {v: 'vertex', a: 'User/Adversary'}

//check that has_access_a(v) returns true
MATCH (v)
WHERE v.name = $v AND $a IN labels(v)
DETACH DELETE v
```

B.2.6 add_account_a,{u_1, ..., u_2},v,l

In this implementation the Python function calls the following Cypher query for each element in the list of u vertices.

```
//check has_access_a(v) returns true
//for u in ulist

:params {v: 'vertex', l: 'access_method', u: 'u_i'}

MATCH (u)
WHERE u.name = $u
MATCH (v)
WHERE v.name = $v
CALL apoc.create.relationship(u, $l, {}, v)
```

```
YIELD rel
RETURN rel
```

B.2.7 rem_access_1_a,v,l

```
:params {l: 'ACCESS_METHOD', v: 'vertex'}

//first check has_access_a(v) returns true
MATCH ()-[l]->(v)
WHERE v.name = $v AND $l = type(l)
DELETE l
```

B.2.8 rem_access_2_a,u,l

The following is almost identical to rem_access_1, a separate version was only included for readability purposes when designing the interpreter and for possible future extensions. The code could be easily modified to only make use of a single remove query and retain all functionality.

```
:params {u: 'vertex', l: 'ACCESS_METHOD'}

//first check has_access_a(v) is true
//then check uses_method_l(v,u) is true
MATCH ()-[l]->(u)
WHERE u.name = $u AND $l = type(l)
DELETE l
```

B.3 Tactics Grammar

```
grammar Tactics;

/*
 Parser Rules
 */
```

```
tactic: t EOF;

//Rules are in order of precedence so
//AND has higher precedence than OR
t:
        t AND t
        | t OR t
        | '(' t ')' //highest precedence as bracket matched first
        | t_single;

t_single: alpha | b | CHECK '(' phi ')';

alpha: (
                gain_access
                | disc_access
                | lose_access
                | create_account
                | del_account
                | add_access
                | rem_access_1
                | rem_access_2
        );
b: (TOP | BOT); //success or failure

phi:
        phi CON phi
        | phi DIS phi
        | NEG '(' phi ')'
        | '(' phi ')'
        | NEG phi_single
        | phi_single;

phi_single: (
                is_account
                | has_access
```

```
                | could_access
                | uses_method
        );


is_account: IS_ACCOUNT '(' V ')';
has_access: HAS_ACCESS a '(' V ')';
could_access: COULD_ACCESS a '(' V ')';
uses_method: USES_METHOD L '(' U ',' V ')';
a: (USER | ADVERSARY);

gain_access: GAIN_ACCESS '(' a ',' V ')';
disc_access: DISC_ACCESS '(' a ',' V ')';
lose_access: LOSE_ACCESS '(' a ',' V ')';
create_account: CREATE_ACCOUNT '(' a ',' V ')';
del_account: DEL_ACCOUNT '(' a ',' V ')';
add_access: ADD_ACCESS '(' a ',' U_List ',' V ',' L ')';
rem_access_1: REM_ACCESS_1 '(' a ',' V ',' L ')';
rem_access_2: REM_ACCESS_2 '(' a ',' U ',' L ')';

/*
 Lexer Rules
 */

AND: ';';
OR: '||';
TOP: 'TOP';
BOT: 'BOT';
USER: 'User';
ADVERSARY: 'Adversary';
CHECK: 'CHECK';
//atomic propositions
IS_ACCOUNT: 'is_account';
HAS_ACCESS: 'has_access_';
COULD_ACCESS: 'could_access_';
```

```
USES_METHOD: 'uses_method_';
CON: '_CON_';
DIS: '_DIS_';
NEG: 'NEG_';
//operations
GAIN_ACCESS: 'gain_access_';
DISC_ACCESS: 'disc_access_';
LOSE_ACCESS: 'lose_access_';
CREATE_ACCOUNT: 'create_account_';
DEL_ACCOUNT: 'del_account_';
ADD_ACCESS: 'add_access_';
REM_ACCESS_1: 'rem_access_1_';
REM_ACCESS_2: 'rem_access_2_';
fragment LOWERCASE: [a-z];
fragment UPPERCASE: [A-Z];
//All vertices v will have a V_ before them
V: 'V_' LETTER+;
//All vertices u will have a U_ before them
U: 'U_' LETTER+;
//list of U and zero or more other members of U
U_List: '{' U (',' U)* '}';
L: 'E_' LETTER+;
//All edges/relationships will have a E_ before them
LETTER: (LOWERCASE | UPPERCASE | '_');
WS: [ \t\r\n]+ -> skip; //Ignore whitespace
```