PyTac: A Python Based Tool for Tactic Evaluation on Account Access Graphs

Daniel J. Cooper

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2024

Abstract

In the modern world, it is common for people to use many online services and devices throughout their daily lives. As the number of services and devices a person uses increases their security can become related through multi-factor authentication, reused credentials and recovery methods. Poor system design or user choices can cause vulnerabilities in these relationships; this is especially acute when considering "account takeover attacks".

Account access graphs with state introduced by Arnaboldi et al. [3] are a form of a directed graph where the vertices represent accounts, devices, and credentials while the edges are labelled to represent the authentication relationships between them. These graphs were introduced with a language of commands, called tactics, which can describe the changes in a graph that would occur during an attack.

This project aimed to develop a tool that can be used to conduct security analyses through the modelling of account access graphs with state and the associated tactics language. This tool was implemented using a custom, Python-based, model for account access graphs with state which could be interacted with through a PyQT5 user interface. The inputted tactics were then parsed and evaluated using an ANTLR grammar, parser and visitor.

This represents the first known implementation of the tactics language which implements backtracking in its evaluation, and includes all shorthand's outlined by Arnaboldi et al. [3]. Further contributions include an extension to the original tactics language to allow for graph elements that match certain properties to be searched for and tactics evaluated based on the result. The tool's usability for conducting analyses is evaluated by conducting two case studies. The first models how an attacker can exploit cycles of access within a user's account configuration and the second uses graphs to explore the claims around a new token theft form allowing for cookie regeneration.

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.

(*Daniel J. Cooper*)

Acknowledgements

I would like to thank both my project supervisor David Aspinal and Sandor Bartha for their advice, guidance, and time throughout this project.

Table of Contents

1 Introduction **1**

2 Background **3**
2.1 Account Access Graphs . . . 3
2.2 Account Access Graphs with State . . . 4
2.2.1 Properties and Operations . . . 5
2.2.2 Tactics . . . 5
2.2.3 Software Implementations of Tactics . . . 8

3 Developing a Tactic Evaluation Tool **9**
3.1 Model Implementation . . . 9
3.2 The Controller . . . 11
3.3 User Interface and Displaying the Graph . . . 11
3.4 Language Processing . . . 13
3.4.1 ANTLR and Language Grammar . . . 14
3.4.2 Evaluating Parse Trees . . . 16
3.4.3 If Statements and For Loops . . . 19

4 Searching Account Access Graphs **21**
4.1 Selecting Components from Graphs . . . 21
4.1.1 AccessFrom and SELECT . . . 21
4.1.2 Combining Query's . . . 23
4.2 Predicates Based on Graph Searches . . . 23
4.3 Evaluating Tactics on Selected Vertices . . . 25
4.4 Implementation of Graph Searches . . . 25

5 Tool Evaluation and Discussion **26**
5.1 Testing the Model, Operations, and Properties . . . 26
5.2 Testing Language Parsing and Tactic Evaluation . . . 27
5.2.1 Evaluation time of tactics . . . 28
5.3 Findings From the Technical Evaluation . . . 29
5.4 An Attack Using Google's Recovery Method . . . 29
5.4.1 The Attack . . . 31
5.4.2 Recreating the Attack in Practice . . . 32
5.4.3 Mitigating Risk and the Effect of 2FA . . . 33
5.5 Exploitation of a Google OAuth Vulnerability . . . 34

5.5.1 The Attack . . . 34
5.5.2 Regaining Control After the Attack . . . 36
5.5.3 Verification of Attack Feasibility . . . 37
5.6 Heuristic Evaluation of Tool Usability . . . 37
5.7 Improvements Following Evaluation . . . 38

6 Conclusions and Future Work **39**
6.1 Challenges Faced . . . 39
6.2 Limitations and Future Work . . . 40

Bibliography **41**

A Background **44**
A.1 Formal Definition of properties for Account Access Graphs . . . 44
A.2 Operation Definitions . . . 44

B Tactics Grammar **46**
B.1 Original Lexer Grammar . . . 46
B.2 Original Parser Rules . . . 47
B.3 Extended Lexer Grammar . . . 48
B.4 Definition of evaluation for SELECT expressions . . . 50
B.5 Extended Parser Rules . . . 51

C Evaluation and Case Studys **54**
C.1 Google Account Case Study . . . 54
C.1.1 Commands for modeling of Attack . . . 55
C.1.2 Intermediate Figures for Attack . . . 56
C.1.3 Commands to Implement 2FA . . . 57
C.2 OAuth Case Study . . . 57
C.2.1 Commands for modeling of Attack . . . 57
C.2.2 Commands for Account Recovery . . . 58
C.3 Heuristic Evaluation of Usability . . . 58
C.3.1 Improved UI . . . 58

Chapter 1

Introduction

When accounts are considered in a security context, they are often done so in isolation. Security advice, such as that provided by Microsoft [23], Google [11], and NCSC (National Cyber Security Centre) [25] often focuses on the primary access method for an account. This primary access method is typically some combination of a username and password, with the potential use of a two-factor authentication code (using an App/SMS/Email) or biometric factor. It is becoming increasingly common for attackers to gain access to accounts that are considered less important, and therefore have weaker security, then exploit this access to compromise a user's more critical online accounts. This may be performed through the use of recovery mechanisms, such as recovery emails or security questions. An example of this would be an attacker convincing phone carriers to change the SIM card associated with a given phone number, in what is known as a SIM swap attack. Following this, any SMS recovery codes will be sent to the attacker, allowing them to gain access to a user's accounts, as was performed against Jack Dorsey (Former Twitter CEO) to gain access to his Twitter account [5].

To help analyse the security of a user against such attacks, Hammann [14] introduced a form of a directed graph whose vertices represent devices, credentials, and accounts. The edges of these graphs represent the access and authentication relationships, such that a password or recovery email would have a directed edge from it to the account it is associated with. This allows for the interconnection of a user's authentication methods to be visually displayed and analysed.

Attacks against a user may include steps that would result in changes to the user's graph, such as changing the recovery email for an account. To describe and model these steps, Arbanoldi et al. [3] introduced a language of tactics, which when evaluated on a graph performs these required changes. For this language to possess meaningful capabilities, the definition of an account access graph was extended with a state, to store when users have access to each account/device/credential. For example, an attacker may have access to an account's password, but not the Two-Factor Authentication (2FA) token needed to log in to the account and make changes.

These account access graphs have been used effectively in studies to help highlight vulnerabilities to subjects, such as those conducted by Hammann et al. [16] and Abra-

ham et al. [2]. Within these studies, participants were interviewed twice; once where information was gathered about which accounts they had and how they are connected; and again where account access graphs generated based on the first interviews were shown to participants and risks were highlighted. It was found that this visual approach was very effective at both conveying possible risks to non-technical participants and encouraging study subjects to make efforts to mitigate these risks within their configurations. However, much of the work of creating and analysing these graphs had to be conducted manually. Though some tools have been developed to mitigate this issue, such as those by Falk [8] and Bartholomä [6], only one such tool developed by Walker [35] implemented a simplified tactics language, which did not include backtracking or the useful shorthands proposed for the tactics language.

The goals of this project are therefore:

- To Develop a tool which can be used for conducting security analyses through the modelling of attacks using account access graphs with state and the tactics language.
- This tool should allow an inputted tactic to be parsed and then evaluated on a maintained graph state.
- The current graph state should be displayed to the user and updated when tactics are evaluated.
- Develop an extension to the language of tactics that allows for elements of the graph to be queried based on the graph's current configuration and for tactics to be evaluated on the results of these queries.

We will begin by introducing the relevant background information. Then in chapters 3 and 4 we describe how each of these outlined goals was achieved through the combination of an ANTLR [28] parser, Python graph model, networkx [13] directed graph diagrams and a PyQT5 user interface. Then finally in Chapter 5 the tool is evaluated. This evaluation is conducted both from a technical point of view, with the use of unit tests, and a usability point of view through a combination of two case studies and a heuristic evaluation of the user interface. These case studies also demonstrate how the developed tool can be used to identify and verify attacks against a user.

Chapter 2

Background

2.1 Account Access Graphs

Account access graphs[1], introduced by Hammann et al. [16] in 2019, give a convenient way to visually map the connections between different accounts, devices, and credentials. These are directed graphs where the nodes represent accounts, devices, or credentials; while the edges represent the authentication relationships between them. We will often refer to the nodes as simply corresponding to an account since the type of node has little impact on the modelling of the Account Access Graph. A simple example of how these graphs are formed can be seen in Figure 2.1; which demonstrates the account access graph for a smartphone that can be unlocked either with a password or a fingerprint. The colours only have meaning locally to the destination node of each edge. This meaning is tied to the access conditions of each node, with each colour defining a set of edges that must all be accessible to unlock the destination account.

Definition 1. (Account Access Graph [16]) An *account access graph* is a directed graph $G = (V_G, E_G, C_G)$ where V_G is the set of vertices. The set C_G contains the colours used in the graph and $E_G \subseteq V_G \times V_G \times C_G$ are coloured edges.

To demonstrate this, suppose that we wish to add a new method to Figure 2.1 that required all three factors to unlock the phone. Then we would draw three new directed

[1] Originally referred to as User Account Access Graphs

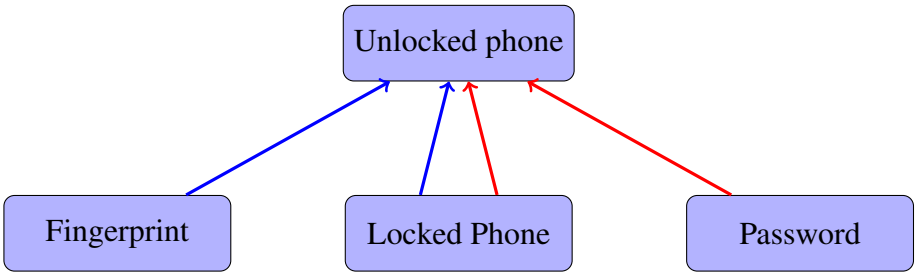


Figure 2.1: The account access graph for a locked smartphone that requires access to the locked phone and either the password or a valid fingerprint to unlock.

edges, one from each factor incident on the unlocked phone node. We would give these three edges the same colour to indicate that all three factors are needed to use this method to access the unlocked phone. We would also easily be able to see that in such a graph this method would provide little extra security, since the two pre-existing methods can provide access with only two of the three factors. This demonstrates how by using account access graphs, we can visually model a user's accounts and use this to identify potential weaknesses and vulnerabilities.

2.2 Account Access Graphs with State

So far, we have only discussed account access graphs that consider the connections that exist between accounts in a static way. However, it is possible that an attacker with a given level of access may be able to change the graph itself, such as in a SIM swapping attack. In such an attack, the phone that receives a two-factor authentication SMS code is changed to a phone to which the attacker has access, allowing them to bypass the second factor [24].

To model these interactions, Arnaboldi et al. [3] introduced a notation that allows access graphs to carry state information, such as the nodes to which a given user/attacker has access. For the following definition, Arnaboldi let $\mathcal{V}$ be a countably infinite set of vertices (accounts, devices, or credentials). Furthermore, they let $\mathcal{L}$ be a countably infinite set of labels and let $\mathcal{A}$ be the set of users/participants that will interact with the graph (e.g. $\mathcal{A} = \{\text{User, Attacker}\}$).

Definition 2. Account Access Graph with State [3] An account access graph with state is a triple $G = (V, E, A)$ where $V \subset \mathcal{V}$ is a finite set of vertices, $E : (V \times V) \to 2^{\mathcal{L}}$ is a map labelling pairs of vertices with finite sets of access methods and $A : V \to 2^{\mathcal{A}}$ is a map labelling vertex with a finite set of participants.

This new construction allows for a more in-depth analysis of the security properties of accounts compared to Hammann's static analysis, achieved through a mechanism called security scoring systems on static account access graphs [16]. There is a slight notational change for graphs with state, using labels to identify the collections of edges, as opposed to colours used by Hammann. To see the benefits of introducing state into the graphs can be seen in Figure 2.2 that shows an example state for the same configuration of accounts as in Figure 2.1. For this example state we can see that the user has access to all three credentials and an attacker has access to the user's password. This allows us to deduce that, for this current state, the user could access the unlocked phone, but the attacker could not. This statement could not be made from the stateless graph alone.

Note that while account access graphs with state are powerful tools when combined with a selection of properties and operations, which we will define in the next subsection. It is often useful to first describe an account configuration without state before we have to discuss which users have access to each node. As a result, a combination of both graph styles will be used within this report.

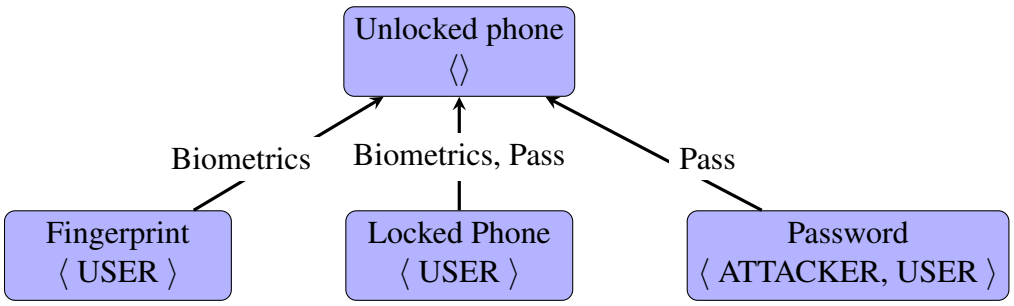


Figure 2.2: The account access graph from of Figure 2.1 once given a state. In this case, the user has access to all three credentials, while the attacker only has access to the password.

2.2.1 Properties and Operations

To develop a language describing the interactions between a user and the account access graph, Arnaboldi et al. [3] introduced properties over a graph G as predicates over the graph's state (V,E,A). There are four predicates defined within the paper. Multiple of these predicates can then be combined in the usual ways to form logical assertions based on a given graph state in order to build propositional combinations. These expressions are written in the following grammar, where $u,v \in V$, $a \in \mathcal{A}$ and $l \in \mathcal{L}$:

$$
\begin{aligned}
\phi ::= \ & \text{is\_account}(v) && v \text{ is an account in the graph} \\
& \mid \text{has\_access}_a(v) && \text{user } a \text{ has access to account } v \\
& \mid \text{could\_access}_a(v) && \text{user } a \text{ has a potential way to access } v \\
& \mid \text{uses\_method}_l(u,v) && \text{account } v \text{ is accessible from user } u \text{ using label } l \\
& \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \text{true}
\end{aligned}
$$

The formal definition for the validity of these properties can be seen in Appendix A.1.

The final building block that we require before a language of tactics may be defined are the operations to be performed on the graphs. There are eight such operations introduced by Arnaboldi et al. [3]. The operations on $v \in V$, $a \in \mathcal{A}$ and $l \in \mathcal{L}$ include actions such as the creation ($create\_account_{a,v}$) and deletion ($del\_account_{a,v}$) of new accounts, users gaining access to accounts ($gain\_access_{a,v}$), and removing ($rem\_access_{a,v,l}$) access for users from an account. The formal definitions for these operations can be found in Appendix A.2.

2.2.2 Tactics

Tactics are programs that act on an account access graph with state σ. These are used within this project to allow users to model changes to a user's account access graph during the course of attacks. Tactics are defined using a language of sequential operations on the graph. The tactics of this logical system[3] are given by the component 't', the possible structures of which are seen below,

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

Where α is an operation, $b \in \{\top, \bot\}$ and ϕ is a property of the graph. Here $t||t$ represents sequential or while $t;t$ represents sequential execution of tactics. The values $\top$ and $\bot$ are used to represent successful and unsuccessful termination of tactics, respectively. The tactic **CHECK**(ϕ) checks if the given property ϕ is true for the actual state of the graph.

These tactics are evaluated on a graph with state σ, which is an account access graph with state, (V,E,A) and the evaluation is notated using the form of judgements

$$\langle\sigma\rangle t \Downarrow b\langle\sigma'\rangle,$$

where $b \in \{\top, \bot\}$.

For example, the successful evaluation of a single operation α (defined as AX-T in the paper) is given by

$$\frac{\langle\sigma\rangle \overset{\alpha}{\rightsquigarrow} \langle\sigma\rangle}{\langle\sigma\rangle\alpha \Downarrow \top\langle\sigma\rangle}.$$

There are however situations where the tactic evaluation will not terminate, for example, the $AX-T$ rule can fail if the graph operation α can not be applied. When the evaluation terminates, the system is deterministic and will allow for the creation of any account access graph from an empty state. The full list of such tactics, all of which are implemented within my tool, is given below.

Definition 3. (Tactics for Account Access Graph Operations [3]) Inductively the following rules define tactic evaluation for account access graphs.

$$\frac{\langle\sigma\rangle \overset{\alpha}{\rightsquigarrow} \langle\sigma'\rangle}{\langle\sigma\rangle\alpha \Downarrow \top\langle\sigma'\rangle} \quad \text{(AX-T)}$$

$$\langle\sigma\rangle b \Downarrow b\langle\sigma'\rangle \quad \text{(CONST)}$$

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

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

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

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

$$\frac{\langle\sigma\rangle \vDash \phi}{\langle\sigma\rangle \text{CHECK}\phi \Downarrow \top\langle\sigma\rangle} \quad \text{(CHECK-T)}$$

$$\frac{\neg\langle\sigma\rangle \vDash \phi}{\langle\sigma\rangle \text{CHECK}\phi \Downarrow \bot\langle\sigma\rangle} \quad \text{(CHECK-F)}$$

Arnaboldi et al. then used these tools in a case study where they performed an analysis of the account configuration for appleID and iCloud accounts on iPhones and, similarly,

Google accounts on Android devices to investigate a vulnerability that was exploited by attackers. This analysis was able to show that the attack on an iPhone user could not be carried out on an Android.

This language is very effective in modelling attacks and can be used to generate any graph we wish. However, there is some ambiguity around the correct behaviour of the || (or) operation within the paper. While following the formal inductive definitions, we see that in the case that an operation's premise fails the program should not terminate, and we should instead end up in a failure state. The "or" operation is then defined to only act based on the inductively evaluated results for its left and/or right tactics, and so should a premise within the left tactic fail the whole statement would not terminate. However, within the iCloud case study in this paper, a tactic of the form

$$\text{rem\_access}_{attacker,AppleID,x}||\top$$

is evaluated, with x ranging over all possible labels sequentially as though it was a series of statements. This suggests another intended behaviour for the or operation, where the failure of an operation's premise is treated as if it were a $\bot$. For my implementation I follow the implied behaviour as it is useful when applying operations to vertices for whom we are unsure of certain properties.

Since the tool developed within this report will only provide mechanisms to modify graphs through the use of tactics, it is important that this does not limit the selection of graphs that can be modelled. Therefore, while the following lemma is not present within the original paper [3], it is important nonetheless to be certain that all graphs can indeed be produced using tactics, thus it is stated and proven here.

Lemma 1. ***Completeness of Graph Tactics*** *Given any account access graph with state (V,E,A), we can construct this graph from the empty graph using only tactics.*

Proof. This result can be proved constructively by providing an algorithm that, given the set V and the maps E,A, will produce the graph (V,E,A). Each of the following steps could be completed individually or together using the SEQ-T tactic. We now define the constructive algorithm as follows:

1. For all vertices, $v \in V$ evaluate the tactic $\text{create\_account}_{\dagger,v}$. Where † represents a user not within the target graph (V,E,A) or, equivalently, $\dagger \notin A(\_)$. Thus, our graph now contains all the required vertices.
2. To add all the required edges, we must perform $\text{add\_access}_{\dagger,\{u\},v,l}$ for all pairs $u,v \in V$ and all $l \in E(u,v)$. This ensures that all the required edges are now within the graph.
3. We no longer wish † to have access to all vertices, and so we must perform $\text{lose\_access}_{\dagger,v}$ for all $v \in V$.
4. Now, for all vertices $v \in V$ and all users, $a \in A(v)$ evaluate the tactic $\text{disc\_access}_{a,v}$. Thus, we now have that all users have appropriate access to each account.

Notice that no assertions were made on the structure of the target account access graph with state, and so this algorithm will work generically on any given graph. We can also

see that the only operation used that required a premise to be satisfied (see Appendix A.2) is add_access. However, this operation simply requires that the user carrying out the attack has access to the incident vertex of the added edge. Since † created all the accounts, and we had not yet removed this user's access this is trivially true for all $v \in V$, thus the operation terminates. Furthermore, since when they terminate, tactics are deterministic [3] the graph is uniquely determined to be our intended graph.

□

2.2.3 Software Implementations of Tactics

Introducing the rigour of an automated language implementation allows researchers to verify that a given tactic will indeed execute and result in the intended graph. Within his MSc project, Walker [35] used a combination of the Neo4j graph database system along with the cypher query language to build an interpreter for the tactics language defined by Arnaboldi et al.[3].

The language was parsed with the assistance of the ANTLR(ANother Tool for Language Recognition) [30] parser generator. This, once given a defined syntax, can be used to generate a language parser and lexer. Once given a user input such a lexer and parser can then be used to build a syntax tree that can be traversed to evaluate user inputs [30].

Walker then used this tool to evaluate several case studies and verify example tactics defined within a draft version of Arnaboldi's paper [3]. Within this process, Walker discovered an error within these example tactics, further demonstrating the benefits of this automation. There were, however, limitations to Walker's implementation. In the abstract notation of the tactics language [3] the graph state is updated continuously as we conduct tactics. If a tactic path results in $\perp$, we backtrack and traverse the next path. For example, the evaluation on an empty graph of the tactic

$$(\text{create\_account}_{a,v}; \text{CHECK}(\text{is\_account}(u)))||\text{create\_account}_{a,u}$$

would result in the creation of only the account u. This is due to the check statement of the first branch failing, thus the state is reverted and $\text{create\_account}_{a,u}$ is evaluated. Due to time constraints, Walker did not include backtracking within his implementation. Instead, Walker evaluated all predicates on the initial graph state and maintained a list of valid operations to evaluate. While this simplifies the language, it does alter the outcome of some tactics defined within the grammar. For example, the following tactic evaluated on an initially empty graph would, in Walker's implementation, result in the account u being created. Since when evaluated on the initial state is_account(v) will return false. Whereas in the abstract notation would result in the creation of the vertex v,

$$(\text{create\_account}_{a,v}; \text{CHECK}(\text{is\_account}(v)))||\text{create\_account}_{a,u}.$$

This discrepancy is something the implementation outlined within this paper aims to avoid, and as a result, backtracking will be implemented. The process of how this is achieved is outlined in the next chapter.

Chapter 3

Developing a Tactic Evaluation Tool

Since most of the graphs that we are concerned with belong to an individual user it is reasonable to expect them to be of moderate size. With graphs generated by Hammann et al. [15] during real-world user studies containing at most around sixty sparse vertices, which were able to represent all the key accounts for a study participant. Although previous work on automatic tactic evaluation was completed using graph databases, this only provides meaningful performance improvements for very large graphs.

For graphs of a size similar to or smaller than those seen within the studies, a custom implementation of account access graphs in a traditional language can provide suitable performance when evaluating tactics. For this implementation, two languages were considered, Python and Java. While both Python and Java have many packages that allow for the modelling of graphs and language parsing; I found that many of the Java desktop GUI packages, such as Swing, were often not as capable as their Python counterparts.

It was however found that Swing would not easily integrate well to allow for the continuous update of a displayed graph. Whereas in Python, PyQt5 allows for elements that can hold and update Matplotlib [17] diagrams, including those of directed graphs. Furthermore, due to Java's verbose style, the logic required to implement graph operations and predicates was more distant from that of the abstract notation when compared to Python. Thus, it was decided to implement my tactic evaluation tool in Python.

3.1 Model Implementation

The Python tool was developed using a model-view-controller structure, the model component is separated into two main components. The first is the account access graph itself, which is provided by a graph class that mirrors the abstract notation outlined in Definition 2. As a result, the code required within this section is not complicated in its own right. However, it was found that many revisions and rereadings of the abstract semantics for the tactic language were needed before confidence that the implemented behaviours were correct. could be gained. This learning was later useful in developing section 4 as it gave me a greater understating of how languages can be written into

formal semantics.

The three components (V, E, A) of an account access graph with state are implemented within the graph class using a set and two dictionaries. The set contains the names for each of the vertices as strings, representing V; while the two dictionaries represent the maps A and E that both take the vertices as arguments and then return sets of users or labels, respectively. If we wished more information to be associated with each node we could follow the precedent established by the abstract notation and introduce a new map that takes a vertex and returns the new information. This simplifies the integration of tactic evaluation on the graphs, as we may take strings from the user and compare them directly with the vertices stored in the graph. Similarly, the labels and users on a graph are also stored as strings.

The vertex set is provided with safe functions for any required interactions and similarly, each of the dictionaries are provided with a wrapper function (A_func and E_func respectively) that allows safe usage of the dictionary, returning an empty set if an unknown key is provided. This means the dictionaries do not have to be updated to include keys that would hold no value, and in the case of vertices that are not within the graph, this also matches a reasonable expected behaviour. The use of wrapper functions allow the model to be completely replaced without changing the rest of the code base. This graph model allows the graph implementation to be lightweight, with only a set and two dictionaries needed to maintain all the graph information. This is useful for both comparing the equality of the graphs and making deep copies, both of which are operations used for later testing. However, the key advantage of being able to duplicate and overwrite graphs by manipulating these three values is seen when implementing backtracking for tactic evaluation in Section 3.4.2.

All atomic propositions defined on account access graphs (*is_account*, *has_access*, *could_access*, and *uses_method*) are seen as ways to check if the graph satisfies a property. As such, the graph class also contains four methods, one for each of the predicates defined on account access graphs. These methods implement the logic for each of these checks and return the appropriate boolean value. This code closely follows the formal definitions, for example, the $has\_access_a(v)$ property has its validity formally defined as:

$$\langle V, E, A \rangle \vDash \text{has\_access}_a(v) \text{ if } v \in V \text{ and } a \in A(v).$$

This formal definition is mirrored closely within the Python implementation for the *has_access* method, the code for which is shown in the code cell below. The similarity of expression between the formal definition and implementation illustrates how closely the Python implementation's methods can match the abstract notation.

```
def has_access(self, v, a):
    return self.is_account(v) and a in self.A_func(v)
```

The second component that must be implemented within the model is the graph operations used to modify graphs. Since these are viewed as acting on a graph, as opposed to checking a property of a graph, they are created as static methods that take a graph object

as an argument. Each operation has a premise that must be satisfied for the operation to be carried out. If the premise of an operation is not met, then a custom PremiseError exception that contains relevant information as to what might have caused the premise to fail is thrown. These exceptions can later be used to provide more details on why a tactic failed to be evaluated to the user. If this premise is satisfied, however, we know that the operation will successfully terminate and thus on the successful completion of an operation true will be returned. The side effect of the operation being executed is an appropriate alteration in the graphs' state.

3.2 The Controller

The controller class acts as the central class by performing the program logic for interactions between the model and the user interface. This separation of roles was established by creating listeners within the controller that upon the activation of intractable elements of the UI call functions within the controller class. This allows UI elements to be replaced or re positioned without requiring changes to the controller code. As long as the same name is used by the UI element responsible for a given signal, the same action will take place in the controller. This also means that only one graph state, which belongs to the controller, needs to be maintained within the application, and it can then be passed to the UI when the displayed diagram needs to be updated.

The controller is also responsible for calling all functions related to the parsing of the tactics language. This is achieved by reading the inputted text from the UI and then passing it as an argument to the parsing component of the system. If an exception is thrown by the parser, then the error message, potentially slightly altered for readability, is displayed to the user as outlined in the UI section.

3.3 User Interface and Displaying the Graph

Now that the basic model has been established, we require a user interface that allows a user to interact with this model and evaluate tactics based on it. A set of goals determined for what this user interface should allow a user to do:

1. Provide a method for a user to enter and execute a tactic,
2. Display the current graph state to the user in a way that allows the effect of a tactic to be understood,
3. Allow a user to save and load a graph state to a file,
4. Allow a user to clear the graph state,
5. Provide feedback to the user on a tactics evaluation, such as if an operation's premise is not fulfilled.

Many Python packages allow for the generation of such user interfaces, the two considered were Tkinter [21] and PyQt5. Tkinter is a standard Python library package that allows for the creation of basic user interfaces. PyQt5 on the other hand is a GUI

framework developed by Riverbank Computing Limited. Which provides a much more extensive selection of features for implementing both GUIs and some back-end components. While Tkinter is integrated into Python and would provide sufficient features to implement the goals for the project, it would limit options for future development and expansion of the program. The decision was therefore made to utilise the PyQt5 framework for developing the user interface.

A PyQt5 interface is built from widgets, each of which adds a different component to the UI. The design starts with a custom QMainWindow widget that represents the pop-up window. Within this window a toolbar is added that contains save, load, and clear buttons. These buttons will be linked to functions within the controller that perform the appropriate actions for goals 3 and 4. Due to its ability to maintain the structure of Python objects, it was decided that when a user saves a graph it would simply be exported as pickle files. These pickle files can then be loaded back into graph objects and used to overwrite the current state when the user loads a file.

Following this toolbar a text input is placed for the user to input their tactics, and a line of black text that prompts the user to input a tactic. This text prompt is then changed upon the execution of a tactic. Giving useful feedback to the user such as whether a tactic returned $\top$ or $\bot$, as well as any syntax or premise errors. An example of this feedback can be seen in Figure 3.2 which shows the response to a syntax error.

The final component of this UI section is an execute button that, once clicked, causes the user input to be read and evaluated. for ease of use, an equivalent action also takes place when the user presses the return key while the tactic input is selected. This combination of text input, text prompt, and execution button achieves goals 1 and 5.

The most difficult component of developing the user interface was achieving goal 2. This was mainly due to requirement of integrating a diagram for the account access graph, which could be updated regularly when a tactic executes or a new graph is loaded. This required the creation of a custom PlotWidget class as a child of the basic QWidget class provided by PyQt5. This custom widget contains a canvas that displays a Matplotlib [17] diagram. Alongside the canvas, this widget also contains a toolbar that allows the user to pan and interact with the diagram displayed within the canvas as though it were a Matplotlib window. Note that while a PlotWidget object must be instantiated with a graph since this provides the graph state that should initially be displayed, it does not store any graph state. Instead, the widget provides a method that given a new graph will clear the canvas, and then display the new graph. This avoids the need to hold state within the widget.

To draw the account access graphs onto a PlotWidget canvas, the networkx [13] package was used as an intermediary. This package providing a good collection of methods for displaying directed graphs as Matplotlib plots. This was done by adding a method to the graph object that packages the custom graph object into the standard networkx format. With vertices whose labels consist of the account name followed by the users who have access to the vertex on a new line. Similarly, the edges were labelled by the list of methods associated with the result of $E(u,v)$, where u is the source vertex and v is the destination vertex for the edge. The networkx draw functions were then used to display these converted access graphs on the PlotWidget's canvas.

Although this system provides a readable diagram that contains all the relevant information, it does have limitations. The main drawback is that, while networkx does provide many positioning algorithms for the vertices, experimentation shows that none of these are perfect for this use case. Sometimes text may overlap, or new vertices will be placed in unusual locations far from other nodes. This issue is particularly bad with the spring layout, and by using a Matplotlib canvas, there are limited options for the user to reposition the nodes. The fact that each time the graph is updated, it is plotted again also limits the continuity between executions, with one new vertex often causing all vertices to be placed differently. These limitations can be minimised by making use of the graphviz layout, which produces a hierarchical layout for the nodes based on the edges between them. The graphviz layout does however become slow at positioning nodes once a high number of nodes and edges are included in the graph, therefore kamada kawai layout is used instead for these large graphs.

Other designs were also considered, such as using a JavaScript plotter like Pyvis then displaying the result as an HTML element within the UI. This would have provided the user with the ability to interact with the plot and move nodes to be more readable in the event of an imperfect automatic vertex placement. There were many difficulties when attempting to take this approach, with every attempt made resulting in a blank rendering within the UI, even when a graph was meant to be displayed. This is likely a result of some incompatibility between PyQt5 and the JavaScript/HTML generated for displaying such graphs. Improving the intractability and automatic node placement of the application do therefore mark areas for improvement and potential further work. One potential solution would be to utilise a system similar to the scoring schemes introduced by Hammann[14] to rank the centrality of the nodes and position them appropriately.

The user interface design resulting from the combination of these key elements can be seen in Figure 3.1, which shows the application open while displaying the example account access graph in Figure 2.1.

3.4 Language Processing

Now that the model and interface have been established, attention must be turned to the process of converting strings from the user into programmatic instructions that can be evaluated on the graph. This is a common problem within language design and thus there are many approaches to solving it, with three key components of such a solution. First, we must perform a lexical analysis to split the input text into a sequence of tokens representing the basic syntactic units recognized by the language [1]. The code that conducts this analysis is known as a lexer. Next, we must apply a parser which, given the input tokens, will analyse the structure of the input and encode this into a syntax tree, e.g. Figure 3.4. Using this syntax tree, we can now evaluate the user-inputted tactics as a series of actions to take on the graph model's state by traversing the tree. An overview of these steps can be seen in Figure 3.3.

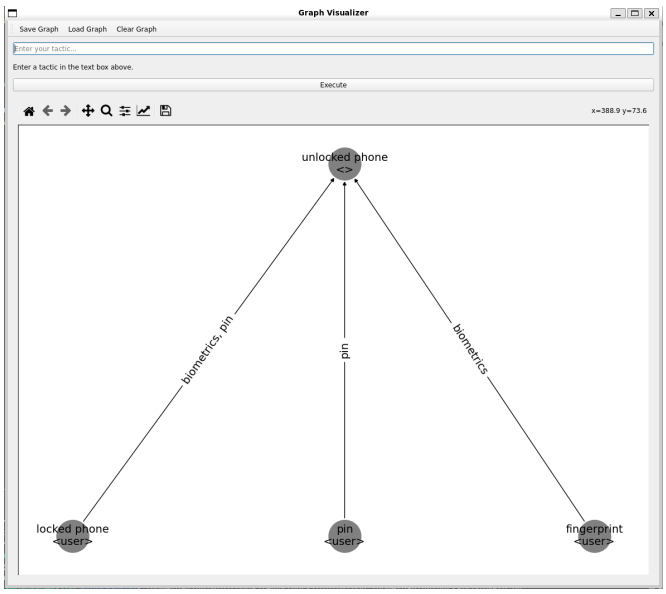


Figure 3.1: Image showing the user interface for the tactic evaluation tool while showing the example graph in Figure 2.1.

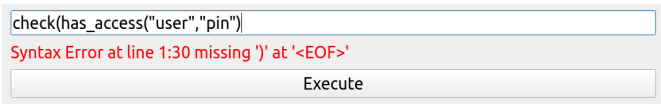


Figure 3.2: A screenshot showing the Feedback provided after a syntax error was made.

3.4.1 ANTLR and Language Grammar

Initially, a custom implementation was pursued, in the form of a recursive-descent parser. This proved to be difficult due to my inexperience in formal language design and would not supply any material benefit to the capabilities of the project. The decision was made to use a parsing package that would aid in the implementation of this component. Many such packages exist, but one of the most common is ANTLRv4 (ANother Tool for Language Recognition) [30], which is a Java-based parser generator maintained by Terence Parr.

Given the grammar for a language in a specified file format, ANTLRv4 will generate not only lexer and parser classes for the language but will do so for multiple target languages. In this case, we are working in Python; however, the same grammar files could be used to parse the language in a program created with any other supported language, such as Java. ANTLRv4 was also used in the Neo4j tactic interpreter developed by Walker [35], and therefore his language grammar was used as the basis for my implementation.

Two files were used to define the tactics grammar, one that contains the lexer rules and another that includes the rules for parsing. Within the lexer file, regular expressions are provided for each of the tokens used within the language. Following convention, these tokens are named in all capitals. In many cases there are multiple valid regular expressions that we wish to recognise as the same token. These alternatives are separated by a vertical bar (|) within a token definition. In commonality with Walker's approach to

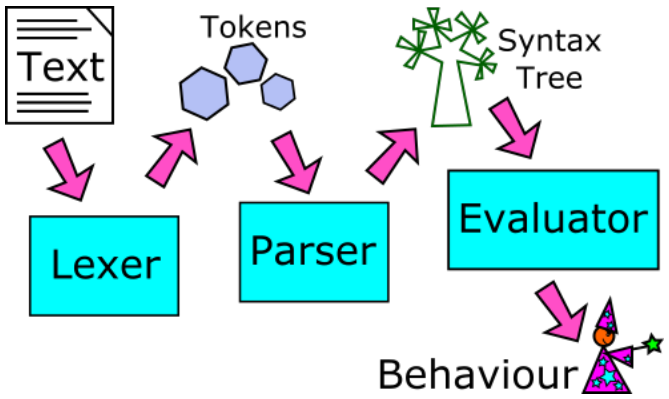


Figure 3.3: The steps required to process an input from a well-defined language into actions to be performed by the application. Diagram by Andy Balaam [4].

the ANTLR grammar, ‘AND’ is used for the lexing rule of ‘;’ and ‘OR’ is the lexing rule of ‘||’. As an example of how a token definition is expressed the rule for the CHECK token, which accepts both the string ‘check’ and ‘CHECK’, can be seen below.

CHECK : ‘CHECK’|‘check’;

Several key changes were made relative to the lexer utilised by Walker. Firstly, shortened versions of the operations and graph properties were introduced as alternatives for their respective tokens; for example, the token for del_account will now accept both ‘del_account’ and ‘del_acc’. This helps to improve usability by making commands less verbose. Similarly, to make the syntax for properties and operations more consistent, an ending underscore present in the regular expressions for many of Walker’s tokens was removed.

The biggest change made was the removal of separate tokens for vertices, labels, and users (User and Adversary in Walker’s system). These tokens were replaced with a string token that will match any input contained within either single or double quotes. Since the only context in which data is entered within the language is within calls to a graph property or operation, what type of object is referred to (e.g. user or account) can be inferred by the position of the string input within these calls. This decision allows for more than two users to be modelled within the system. A fact that could be useful if, for example, a recovery method was based on another user’s account or there were multiple attackers. The elimination of argument/parameter types also reduced the number of environmental maps required when variables are introduced in Section 3.4.3. As a result of these changes, a call to the operation create_account is now defined to take the shape,

```
CREATE_ACCOUNT LPAR arg COMMA arg RPAR
```

where CREATE_ACCOUNT, LPAR, COMMA and RPAR are tokens while arg is another rule that includes strings surrounded by quotes.

Within the second file, these tokens are used to define all the grammatically valid structures that the parser should recognise. Each of these rules is presented similarly to that of the lexer rules. Though now they are each given a label to aid in the development

of the visitor in the next section. The rules for a tactic can be seen below, where t_simple represents the evaluation of $\top, \bot$, CHECK, or a single operation. The precedence of operations is an important part of parsing a grammar, as it informs which pattern should be matched first (for example, in numeracy * has a higher precedence than +). Within an ANTLR grammar, this is given by the order in which the rules are defined, with the highest precedence being defined first. Much like in Walker's grammar[35], AND (';') is given higher precedence than OR ('||'). This was done to stay in keeping with the precedence of the predicate logic, which represents a component of the language within the check condition. For a user to specify the order of operations, outside the canonical ordering implied by the language precedence, we also introduce brackets in the parenthesized rule. The first two parser rules as outlined here can thus be seen below.

```
tactic: t EOF ;

t:t AND t # TOp
 |t OR t # TOp
 | LPAR t RPAR # Parenthesized
 | t_simple # Simple
 ;
```

Note that the rules used for t make this language left recursive since many of the alternatives for t immediately refer back to t [33], while this was not supported in ANTLRv3 within ANTLRv4 such languages are automatically rewritten into non-left recursive languages when applied to grammar[28].

The rest of the lexer and parser rules can be found in Appendix B.1 and B.2 respectively. We can combine these rules inductively such that any tactic as defined by Arnaboldi et al. [3] can be inputted as a valid string. For example the tactic,

$$\begin{gathered}\text{CHECK}(\text{is\_account}(\text{phone}));\\ \Big(\text{CHECK}(\text{could\_access}_{\text{attacker}}(\text{phone}); \text{gain\_access}_{\text{attacker, phone}}\Big) \,||\\ \big(\text{CHECK}(\neg\text{could\_access}_{\text{attacker}}(\text{phone})); \text{disc\_access}_{\text{attacker,phone}}\big) \,||\top\end{gathered}$$

can be encoded in the grammar as the following string.

```
check(is_account("phone"));
(check(could_access("attacker","phone"));
gain_access("attacker","phone")) ||
(check(!could_access("attacker","phone"));
disc_access("attacker","phone")) || TOP
```

3.4.2 Evaluating Parse Trees

Using ANTLR to generate Python lexer and parser classes for the grammar described above then allows us to tokenise and then parse any string into a parse tree. The result

of this process for the example tactic in the previous subsection can be seen in Figure 3.4. The remaining challenge is to use this tree to execute the inputted tactic. For this purpose ANTLR can generate tree walkers that visit the nodes of a parse tree with two different patterns, listeners, and visitors [28].

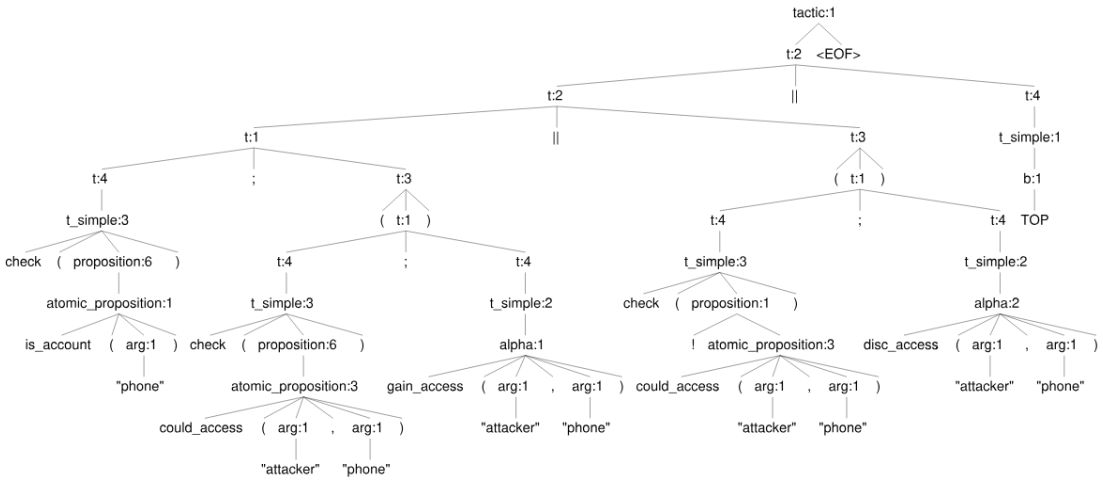


Figure 3.4: An example tactic parse tree as produced by an ANLR generated parser for the tactic language. Generated using ANTLR lab [29].

When using the listener pattern a default walker class performs a depth-first walk of the parse tree. In the case of the parse tree in Figure 3.4 this would result in the $\mathrm{CHECK}(\mathrm{is\_account}(\mathrm{phone}))$ branch being explored first. When this walker enters or leaves a node, it will send a signal that will trigger a function within the listener corresponding to the type of node visited (the node type is determined by the label given within the parser rules). While this could theoretically be used to conduct our tactic evaluation listeners do not give control of graph traversal and we will not always wish to evaluate all breaches of the tree. For example, when the first branch of an or statement returns $\top$ we don't wish to evaluate the second branch. The listener also loses the structure of the parse tree and so is not useful for constructing objects whose structure is based on that of the parse tree, such as an AST (Abstract Syntax Tree). As a result of these limitations, the listener pattern was not used.

The visitor pattern on the other hand gives full control of the traversal down the parse tree, starting with the root node. By default, the visit function is recursively called on each of the node children. This results in the same depth-first walk of the parse tree as in the listener. However, when using the visitor pattern we may override the default behaviour for each node type, changing the order in which the children are visited. If when visited the children return a value we can then use this returned value to conduct calculations or even create objects at each stage of the visitor's graph traversal.

This leads to two reasonable designs for the evaluation of the parse tree using a visitor. The first involves directly evaluating the tactic within the visitor. While this would be sufficient for simply executing the tactic that the user provides, it does limit the extensibility of the system. This also means that the visitor logic often gets cluttered with large amounts of code for executing the tactics. The second option is to build a set of classes, one for each of the node types within the tree. Then use the visitor to

instantiate nested instances of these classes producing an intermediate representation of the parse tree. For example, the visitor for the '||' and ';' operators are given below. Here, we can see that a TOp (Tactic operator) class is created using the returned objects from both the first and second t rules.

```
def visitTOp(self, ctx:tacticParser.TOpContext):
    # get token for operatior
    op = "OR" if ctx.OR() else "AND"
    return TOp(op, self.visit(ctx.t(0)), self.visit(ctx.t(1)))
```

We call this intermediate representation an Abstract Syntax Tree (AST) and it holds all the key information that is relevant to the evaluation of the tactic, while loosing information that is only relevant to the parsing of the input such as the position of brackets or other excess tokens. Each of the classes used to build this AST are given an execute method. This method takes a graph state and an environment dictionary holding any variables, then performs the logic for evaluating the tactic on the graph state. Within the evaluation of tactics, True and False are used as substitutes for the formal $\top$ and $\bot$ values that the tactics formally return, allowing for much easier use of Pythons built-in logic.

Within the first iteration for my design for the AST each rule within the parser, was given its class within the AST. However, upon comparing this design to other common AST structures, such as those for numerical expressions, it was found that this resulted in far too many overly specific classes. To remedy this the parser rule labels were changed and the visitor was modified to return elements of a new and reduced selection of classes. This reduction was mainly due to the combining of operations that shared similarities in the number and types of input. This included combining the predicate operations $\wedge, \vee$ and $\neg$ into a single class. These can be combined since they all take in either one or two inputs, all of which are of the type predicate. The combined classes then simply store a string containing which operation to complete and the needed arguments. It was decided, however, to leave the operation and atomic predicates as separate classes (one for each of them, i.e. create_class and has_access both have their classes). This was mainly because they all take different numbers and types of input, making any combined class needlessly complex and difficult to read. Such a combined class would also require special conditions for the visitor further complicating their usage.

As an example of these new AST node classes, the TOp class can be seen below. This class represents the two binary operations ('||' and ';') that can be performed on tactics. Which of these should be performed is stored within the op(operator) property, then at execute time the operator is checked and the appropriate logic carried out. We can also see that when the or operation is used backtracking is achieved by making a temporary copy of the graph, and then reverting the graphs state if execution fails on the first branch. Notice that because the AND operator requires both the left and right tactics to be correct it does not introduce an opportunity for backtracking, while the OR operator does. Furthermore, we wish the '||' operation on tactics to treat a premise exception for the left tactic as if it were a return value of $\bot$ a try-except clause is used to catch such premise errors.

```
class TOp(ASTNode):
```

```
    def __init__(self, op, left, right):
        self.op = op
        self.left = left
        self.right = right

    def __str__(self):
        return f"TOp({self.op},{self.left}, {self.right})"

    def execute(self, graph, env):
        match self.op:
            case "OR":
                temp = graph.copy()
                try:
                    b = self.left.execute(graph, env)
                except PremiseError:
                    b = False
                if not b:
                    graph.overwrite(temp)
                    b = self.right.execute(graph, env)
                return b
            case "AND":
                return self.left.execute(graph, env) and \
                       self.right.execute(graph, env)
```

3.4.3 If Statements and For Loops

Arnaboldi et al. [3] introduced three shorthand's on top of the core tactics and operations. The first was an if statement of the form

$$\text{IF } \phi \text{ THEN } t_1 \text{ ELSE } t_2$$

where ϕ is some predicate on the state of the graph and t_1, t_2 are two tactics. This was introduced as short hand for the following tactic which conditionally branches based on ϕ,

$$(\text{CHECK}(\phi); t_1) || (\text{CHECK}(\neg\phi); t_2).$$

This is implemented by adding new tokens for IF, THEN and ELSE then adding the if statement as an alternative to the t_simple parser rule. When this alternative is visited while building the AST, we then simply instantiate classes that match the conditionally branching tactic. This means that no additional classes or execution logic is required within the AST classes.

The next shorthand introduced was the FORALL loop, which allows a tactic to be repeated over several vertices. The calls to this shorthand are of the form

$$\text{FORALL } x \text{ IN } [v_1, v_2, \ldots, v_k] \text{ DO } t$$

where x is the variable name to be used, v_i are vertices, and t is a tactic that uses the variable x. This would then expand to $t[x \mapsto v_1]; t[x \mapsto v_2]; \ldots; t[x \mapsto v_k]$, where $[x \mapsto v_i]$

denotes assigning the value of the variable x to be v_i. Due to the removal of strict typing within my language implementation the requirement of x to represent a vertex was removed. However this does allow the user more flexibility in how this short hand is used.

To implement this for loop, a new variable token was added to the lexer which matches any input containing only characters, numbers and underscores. This token was then included within the choices for an argument of all the operations and predicates.

However, since a variable token may match with a wide selection of possible inputs it introduces the possibility of collisions with ket words within the language. These potential collisions could be removed by adding a special character in front of variable names (e.g. &x instead of x). This would make the language more cluttered and does not provide substantially more functionality to the user. Another possible solution could be found in the lexer modes offered by ANTLR4 [28], allowing different matches to be made based on which mode the lexer is in. Again this would introduce unnecessary complexity into the design for limited functionality gains, and as a result, it was decided to simply provide the user with a clear syntax error when such a restricted word is used as a variable name.

To use this variable argument an environment map was introduced, which is stored in the controller and when given a variable name returns the string value currently associated with it. Then when a variable is evaluated, it is looked up within this map, and the resulting string is returned, allowing the variable to perform identically to a user inputted argument string from the perspective of the tactic call. A FORALL class was then added to the possible node types for the AST, which, when evaluated, will loop through the choices for the variable, updating the environmental map, and then evaluating the tactic. If any of these tactics returns false, then the entire for loop returns false.

The final shorthand introduced by Arnaboldi et al. was a FORONE loop, which is similar to the FORALL loop described above aside from that it represents an integration of the OR tactic as apposed to the AND tactic. This shorthand was therefore implemented with a similar approach to that of the FORALL shorthand, with the only significant difference being a change in the execution logic to match the behaviour of the OR tactic.

Chapter 4

Searching Account Access Graphs

So far all defined tactics have depended on knowing which vertices, labels and users you wish the tactic to act on. For example, to evaluate gain_access$_{a,v}$ we need the two explicit values of a and v. It is however common within attacks to conduct the same actions on many different accounts/devices which all match certain properties. For example, after stealing a reused password an attacker may wish to gain access to all accounts which use it for authentication.

As a result of this, the goal of this chapter will be to design an extension to the tactics language that allows for the evaluation of tactics based on searches/queries of the graph.

4.1 Selecting Components from Graphs

The process of developing this extension to the tactics language started by defining a selection of simple queries. These simple queries are notated within the extended tactics grammar with the following rule.

$$\gamma := V \mid E(u,v) \mid A(v) \mid \text{AccessFrom}(U) \mid \text{ FROM } S \text{ SELECT } x \text{ WHERE } \phi \mid \{a_0, \ldots, a_n\}$$

for ϕ some predicate, x some variable name, $u, v \in V$, S is some set of graph elements, and $U \subseteq V$. We also define $a_i := v \mid l \mid a$ for $v \in V, l \in \mathcal{L}$ and $a \in \mathcal{A}$. Where we are taking V, E, A to be the graph components as in the definition of an account access with state 2. As such V is the set of all vertices in the graph, $E(u,v)$ is considered to be the set of all labels for the edge from u to v and $A(v)$ is the set of users who have access to the vertex v. Similarly, the final alternative for γ, given by $\{a_0, \ldots, a_n\}$, is an explicit set of elements from the graph.

4.1.1 AccessFrom and SELECT

Following a syntax similar to other query languages, such as SQL [7], the SELECT query is a way for the user to query the set S for elements that satisfy the given predicate. The evaluation of SELECT is inductively defined by the following rules:

$$\frac{\langle\sigma\rangle S \Downarrow \varnothing\langle\sigma'\rangle}{\langle\sigma\rangle\text{FROM } S \text{ SELECT } x \text{ WHERE } \phi \Downarrow \varnothing\langle\sigma'\rangle} \quad \text{(SELECT-E)}$$

$$\frac{\langle\sigma\rangle S \Downarrow S'\langle\sigma'\rangle \quad Y_1 \cap Y_2 = \varnothing \quad Y_1 \cup Y_2 = S' \qquad \forall y_i \in Y_1 \langle\sigma\rangle CHECK(\phi) \Downarrow_{[x\mapsto y_i]} \top\langle\sigma'\rangle \quad \forall y_i \in Y_2 \langle\sigma\rangle CHECK(\phi) \Downarrow_{[x\mapsto y_i]} \bot\langle\sigma'\rangle}{\langle\sigma\rangle\text{FROM } S \text{ SELECT } x \text{ WHERE } \phi \Downarrow Y_1\langle\sigma'\rangle} \quad \text{(SELECT)}$$

Where $\varnothing$ represents the empty set. An alternative considered for selecting graph elements would be to check for relational rules, as is done in other graph query languages like Cypher. To make use of the pre-existing predicates within the tactics language this option was not taken.

Once more language for predicates using graph queries has been developed we will be able to make more meaningful searches. However by only using the original predicates on graphs, as defined by Arboldi et al [3] we can define two very useful shorthands, $\text{Pred}_{l,v}$ and $\text{Succ}_{l,u}$, where $l \in \mathcal{L}$, $u, v \in V$. These provide short and easy ways to query for all vertices preceding and succeeding, respectively, the given vertex v when considering only edges with the label l. These represent shorthand as opposed to new commands, since they may be expressed as SELECT calls.

$$\text{Pred}_{l,v} := \text{FROM } V \text{ SELECT } u \text{ WHERE uses\_method}_l(u, v),$$

$$\text{Succ}_{l,u} := \text{FROM } V \text{ SELECT } v \text{ WHERE uses\_method}_l(u, v).$$

Some other useful shorthand's were also introduced into the searching language. This includes the use of '_' as a vertex. This shorthand indicates that we wish to return the results for all vertices in V. For example, $E(\_, v)$ would mean we wish to return all the labels for edges that end at the vertex v.

The other non-trivial query, AccessFrom(U), included in the definition of γ is defined similarly to the function of the same name within Hamman's paper [14], as follows.

Definition 4. AccessFrom(U) is defined to be the set of vertices in V that can be accessed directly or transitively from the given set of vertices U. This can be recursively defined as

$$\text{AccessFrom}(U) := \{v \in V : v \in U \vee (\exists u, l.l \in E(u, v) \wedge \forall x \in V.l \in E(x, v) \Rightarrow$$

$$x \in \text{AccessFrom}(U))\}.$$

This allows the user to see which vertices could be accessed (potentially transitively) by an attacker with access to the accounts within U. For example, when applied to the account access graph in Figure 5.2 the query AccessFrom(Laptop) would return $\{\text{Laptop, Google Chrome, Google}_{\text{open}}\text{, Google PWD Manager, MS Password, Outlook, Google}_{\text{full}}\}$.

4.1.2 Combining Query's

To provide a method for combining queries, we introduce the S rule given by,

$$S := S \cup S \,|\, S \cap S \,|\, S \setminus S \,|\, \gamma \,|\, x$$

where γ is one of a selection of simple queries as described in the previous section, and x is a variable. Here we also include the standard set operations, union ($\cup$), intersection ($\cap$) and set minus ($\setminus$). These combinations are defined to evaluate as you would expect, such as returning the intersection of the sets S_1 and S_2 for $S_1 \cap S_2$. Note also that the evaluation of these combinations can be formally defined inductively, similar to previous language components. However, since these represent standard set operations, for brevity only the definition for the UNION operation is included with the remaining set operations defined in Appendix B.4.

$$\frac{\langle\sigma\rangle S_1 \Downarrow \{a_1,\ldots,a_i\}\langle\sigma'\rangle \;\; \langle\sigma\rangle S_2 \Downarrow \{b_1,\ldots,b_j\}\langle\sigma'\rangle}{\langle\sigma\rangle S_1 \cup S_2 \Downarrow \{a_1,\ldots,a_i,b_1,\ldots,b_j\}\langle\sigma'\rangle} \qquad \text{(UNION)}$$

When using the γ alternative, we allow the use of any S rule in place of any set, such as when calling AccessFrom, allowing for nested searches. To allow for the results of searches to be saved and reused without needing to be retyped, a variable assignment tactic of the form:

$$\text{LET } x = S$$

where x is a variable name and S is some search that is also added to the language. This rule is evaluated according to the following semantics.

$$\frac{\langle\sigma\rangle S \Downarrow S'\langle\sigma'\rangle}{\langle\sigma\rangle \text{LET } x = S \Downarrow_{[x\mapsto S']} \top\langle\sigma'\rangle} \qquad \text{(VAR-S)}$$

To allow these variables to be used, we also included x as an alternative of S. This means they may be used in place of a search anywhere within the language, if a variable contains a single element instead of a set where a set is expected or vice versa an error is displayed to the user.

4.2 Predicates Based on Graph Searches

To make full use of the querying ability introduced in the previous sections, we require methods for checking the properties of the resulting sets. These predicates can then be used within nested SELECT and branch statements to produce more sophisticated results. We therefore extend the original set of predicates ϕ by the following new rules.

$$\begin{aligned}\phi_{extention} ::= {} & \text{transitive\_access}_a(v) \\ & |\, S == S \,|\, S \subseteq S \,|\, S \supseteq S \,|\, s \text{ in } S \\ & |\, exp > exp \,|\, exp < exp \,|\, exp >= exp \,|\, exp <= exp \,|\, exp == exp\end{aligned}$$

With exp defined by,

$exp := exp + exp \mid exp - exp \mid exp * exp \mid exp \,\%\, exp \mid exp / exp \mid num$

where *num* is either an integer or the length of a set, as follows,

$num := [1-9] + \mid LEN(S)$.

Here we can see three different types of predicated have been added. The first consists of the transitive_access$_a(v)$ property. This graph property is the natural extension of the could_access property from the original tactics language. Informally, this property would return true if the given user a could, after performing nothing but gain_access operations, eventually gain access to the given account 'a'. The formal definition for the validity of this property is provided below.

Definition 5. Definition of validity for transitive_access.

$\langle V, E, A \rangle \vDash$ transitive_access$_a(u)$ if $u \in$AccessFrom(FROM V SELECT x WHERE has_access$_a(x)$) where a is a user and $u \in V$.

The next type of comparison introduced are set comparisons. These include checking if one set is entirely contained in another, or if an element is contained within a given set. These provide for a much more expressive selection of possible comparisons.

The final type of predicate introduced is based on comparing expressions (exp). These expressions are numerical equations involving only integers, and the length of searches. The validity of these comparisons between expressions is defined to be the same as equivalent comparisons between integers. For example, $1 >= 1$ is true, while $1 > 1$ is false. These forms of comparison are useful for checking if a given property of the graph state is true. For example, if a given account has more than four predecessors for any given label.

These expressions have many rules for how they may be combined. This includes a division rule, however, to keep any results of division within the ring of integers, this represents a floored division method and a separate mod rule for calculating the remainder is provided. The evaluation of these exp rules are formally defined in Appendix B.4.

Since the length rule is particularly important, allowing for queries to be compared using the exp rules, unlike the other rules it is formally defined here.

Definition 6. Evaluation of length of sets. The evaluation of the length of sets by the following rule.

$$\frac{\langle \sigma \rangle S \Downarrow U \langle \sigma' \rangle \quad |U| = n}{\langle \sigma \rangle \mathrm{LEN}(S) \Downarrow n \langle \sigma' \rangle} \qquad \text{(LENGTH)}$$

These predicates may now be combined with SELECT statements to search for elements of the graph that satisfy certain properties. For example, the search query given by,

FROM V SELECT x WHERE (
    FROM E(_,x) SELECT l WHERE (LEN(PRED$_{l,x}$) == 1) ! = {})

will select all vertices that can be accessed with the use of only one factor. This could be a single password or a recovery mechanism.

4.3 Evaluating Tactics on Selected Vertices

The final requirement before we have a full system for querying account access graphs is to have a mechanism that allows for the results of these searches to be used in evaluating tactics.

This can be achieved through the use of the pre-existing "for all" and "for one" short-hand's. Here, we simply replace the array we iterate through with the result of a search. For example, we may wish to execute the tactic,

For All x in FROM V SELECT x WHERE has_access(a) do lose_access(a,x)

which would search for all the graph vertices the user a has access to, then would remove the user's access to all of these vertices. This then completes the formal definitions for the changes made to the tactics language.

4.4 Implementation of Graph Searches

The extended tactics and query language was implemented on top of the original system for tactic evaluation as described in section 3. The majority of this development involved simply altering the ANLR lexer and parser files to match the new extensions, and then expanding the visitor and AST classes to match this larger language. All of this follows the same patterns and mechanisms as described in Section 3 and thus will not be repeated here. The starting Parser rule was also altered to allow the user to input just an expression or search, the results of these inputs are then displayed when evaluated. This allows for searches and checks to be performed on the graph without requiring them to be contained within a larger tactic.

The updated ANLR lexer and parser, with support for this new extended language, can be seen in Appendix B.3 and Appendix B.5 respectively. The only significant change relative to the abstract notation is the use of \$ at the start of the fundamental sets, such as $\$V$ for the set of vertices rather than simply V. This change prevents the new keywords, many of which are simply capital letters, from overlapping with common variable names.

Chapter 5

Tool Evaluation and Discussion

Before any tool can be used practically, it needs to be tested and evaluated to ensure it is operating properly. From a technical point of view, we need to make sure that the tactics are interpreted and then evaluated correctly. To achieve this evaluation, a series of unit and integration tests of the language are performed using the unittest Python module. These tests are outlined within sections 5.1 and 5.2.

While it is important to ensure the technical correctness of such a tool, potentially just as important is its usability. This is hard to quantify in terms of formal tests and since the tool is designed for a user already familiar with the tactics language, a user study would not be practical. Instead of this, two worked examples are performed using the tool in Section 5.4, and Section 5.5. This provides an opportunity to practically use the tool and recognise any shortcomings in its usability for conducting security analyses. Within section 5.6 this usability will then be assessed using 10 rules outlined in the heuristic evaluation framework introduced by Jakob Nielsen [26, 27].

5.1 Testing the Model, Operations, and Properties

Testing of the technical components began by focusing on the core Python model and its functions. Throughout this process, two example graphs were used. The first of these was the empty graph that contained no vertices or edges, this provides a useful edge case for many functions. The second was the graph of Example 2.2 modified to include an additional label on the edge between the locked phone and the unlocked phone. This second example graph was referred to within the testing as the prototype graph and contains all the typical components that we will see within a graph. These graphs were instantiated with the appropriate components rather than built using tactics since this isolates any errors in the Python model's implementation. A test was also conducted to ensure that the two mechanisms resulted in an equivalent graph being generated. Two examples of the tests carried out can be seen in Figure 5.1.

So that these preset graphs could be trusted for further analysis, testing began by ensuring that these graphs had been instantiated with the correct properties. This was done by directly asserting what the set V and the maps E, A should be for a given graph

once the object was created. Other functions such as has_access were not used at this stage, as they have not been tested. These initial tests also included unit tests to ensure a graph model is exported correctly to a networkx model by the to_networkx method. This is used for piloting any graphs within the UI. Ensuring any plots used in testing are accurate to the graph modelled. Similarly, at this stage the methods to copy, overwrite, and check equality for graphs were tested; including their behaviour on edge cases such as the empty graph.

Now that assurances have been made about the functionality of the graph class, the next phase of testing involves the operations and graph properties. For each of the graph properties (is account, has access, could access, and uses method). Only a few tests were required since they do not modify the graph in any way and simply return Boolean's. As a result, each was evaluated on around 4 test cases where it was asserted the correct values were returned. For the operations, much more extensive testing was required,

```
def test_is_account(self):
    # test is account
    self.assertTrue(self.g.is_account("locked_phone"))
    self.assertTrue(self.g.is_account("unlocked_phone"))
    self.assertTrue(self.g.is_account("pin"))
    self.assertFalse(self.g.is_account("hat"))
    self.assertFalse(self.g.is_account("user"))
```

(a) Screenshot showing a test for the is_account property. Where self.g is a prototype graph, as described within Section 5.1.

```
def test_gain_access(self):

    g = prototype_graph()
    self.assertFalse(g.has_access("user", "unlocked_phone"))
    self.assertTrue(g.could_access("user", "unlocked_phone"))
    self.assertTrue(operations.gain_access(g, "user", "unlocked_phone"))
    self.assertFalse(g.could_access("hacker", "unlocked_phone"))
    self.assertTrue(g.has_access("user", "unlocked_phone"))
```

(b) Screenshot showing a test for the gain_access operation. The graph g is a prototype graph, as described within Section 5.1.

Figure 5.1: Screenshots showing two examples of unit tests conducted on the model.

with a total of 23 tests performed on these 8 functions. While all of these operations had at least one test dedicated to them some are substantially simpler than others, thus requiring fewer tests. For example, disc_access, as formally defined in Appendix A.2, does not have a premise and thus has a simple expected behaviour. Whereas add_access has multiple conditions within its premise requiring more tests to ensure the method follows the expected behaviour.

5.2 Testing Language Parsing and Tactic Evaluation

Now that we have some assurances based on the underlying model, the next key area for testing is the parsing and evaluation of text inputs from the user. This process started by ensuring that the abstract syntax trees for each of the basic types of input (e.g. individual operations or || tactics) were of the correct form.

The structure of large ASTs produced by the parser was not directly inspected during testing. This was due to the number of objects this would require the creation of. The testing instead focused on the results from the evaluation of these trees on given graphs. However, we can be sure that if a tactic is evaluated correctly, then the appropriate AST is also constructed correctly.

The next selection of tests focused on the evaluation of user inputs for multiple different combinations of tactics. In the order of evaluation, these tests are:

1. For each of the simple tactics, that have a direct correspondence to a method or function within the Python code (i.e. operations and graph properties), any tests performed on the equivalent Python function in the section above were repeated. This time, the functions were called by parsing and evaluating an appropriate input string (with predicates wrapped in a CHECK statement).

2. A series of eight compound predicates involving a combination of conjunction, disjunctions and negations, such as "is_account('phone') $\vee$ is_account('laptop')", were evaluated. Ensured that they returned the expected truth value (true for $\top$ and false for $\bot$).

3. To evaluate the precedence of evaluation for the binary operations ('||' and ';') 18 statements that only used $\{\top, \bot, ||, ;, (,)\}$ were performed. Treating $\top$ as True, $\bot$ as False, || as OR and ; as AND the value these different statements return is dictated entirely by the order en which they are evaluated. If they all return the expected values, we can be sure that the precedence, associativity, and distributively of tactic evaluation are correct.

4. To test that the graphs are indeed being modified correctly, and reverted when backtracking is required, a combination of tactics were used to ensure modifications are being continuously made to the graph, and changes are reverted when backtracking should take place. Some such tactics use CHECK statements to ensure only the appropriate modifications had been made at a given stage of the tactic, while others perform operations that would fail if operations earlier in the tactic had not been carried out.

5. Testing of the for loops mirrored that of the binary operations since they functionally represent a shorthand for a long chain of || or ; tactics. The only new tests introduced were for when an invalid variable was used within the tactic or if the user chose to nest two for loops.

After these tests on the core tactics language, a set of 23 new unit tests following a format similar to that of the tests above were introduced to test the additional language components introduced for the graph queries. These focused on asserting that the correct result was returned by any combination of searches and expressions, as well as that any new predicates were evaluated correctly.

Overall, this means that throughout all testing a total of over 100 tests have been created, each of which contains multiple assertions and/or tactics which are evaluated. All of these tests have been passed successfully by my program and combined provide 92% coverage, as reported by Coverage.py. The majority of untested lines were string methods for the AST classes since these are intended for debugging. During future development, these tests can be repeatedly run with minimal modification; aiding to ensure any modifications do not affect the system's correctness.

5.2.1 Evaluation time of tactics

To evaluate the performance of the implementation, a selection of tactics were evaluated over a variety of different graphs. The parsing and evaluation time for each of these

were then recorded. It was found that even after evaluating large tactics on graphs with 150 vertices, the evaluation time rarely reached above $0.2s$ on large graphs. One such tactic used (seen in Appendix C) selects all vertices for which an attacker requires access to only one more predecessor before they could gain access to the vertex. The tactic then performs an attack on one of the selected vertices by discovering access to the required predecessor. When evaluated on a randomly generated graph with 150 vertices and around 20 edges per vertex, this tactic took $0.13s$ to be parsed and then evaluated. This evaluation time takes only a small fraction of the total end-to-end (input to new graph displayed) latency for such a situation.

The largest contributor to the end-to-end latency was found to be the networkx vertex positioning algorithm. For 150 vertices, this took $57s$ to position the vertices. To correct this, a faster standard algorithm could be chosen, however the current algorithm is sufficient to feel responsive to the user and to be practical for conducting security analyses. This is especially true considering that most relevant graphs found in studies contained at most around 60 vertices.

5.3 Findings From the Technical Evaluation

This suite of tests discovered numerous small bugs and errors in the software, such as swapped argument placement with the abstract syntax tree's evaluation functions. One of the potentially more significant errors discovered within this testing was that when defining the lexer rules the VAR token, used to match with variables in the language, will match with other keywords within the language. This was mitigated by simply giving the variable rule the lowest precedent, allowing all other keywords to match first. This does, however, lead to errors when a user tries to use such a reserved keyword as a variable name since it is matched to a token other than VAR which the parser will not be expecting. This is not a critical error as the user may simply choose a different variable name, and indeed reserved words are not uncommon in programming languages, such as 'while' or 'for' in Python.

Now that there are some assertions as to the technical correctness of the system it is important to evaluate the overall usability of the program as a tool to explore the evaluation of tactics. This will be explored in the next two sections where we conduct case studies based on real world vulnerabilities.

5.4 An Attack Using Google's Recovery Method

Google Chrome has a 64.7% market share in web browsers, its second-closest competitor being Safari [10]. This popularity is not unwarranted, with Chrome providing many useful features to users. The feature of interest for this attack is the password manager. The Google password manager is a built-in way, which is on by default, for users to save their login details to their Google account; these can then be accessed from any computer signed in Chrome [12]. Use of such an inbuilt browser password manager has been discouraged for as long as they have existed [18], mainly due to the limited

protection they provide. Despite this evidence, these mangers do still prove to be highly popular due to their convenience and ease of use.

Since Google allows a user to set a recovery email, access to which is sufficient to regain complete control of the Google account, if a user were to save the password to their Google account's recovery email into their Chrome password manager, a cycle would be created. Such a configuration of accounts is shown in Figure 5.2, where $\text{Google}_{\text{open}}$ represents an open Google session, while $\text{Google}_{\text{full}}$ represents access to a fully logged-in Google account. We can see the cycle produced by following the arrows from the $\text{Google}_{\text{open}}$ node through the password manager, Microsoft password, Outlook account and finally back to the Google account. Such cycles can be indications that there are weaknesses in the configurations of a users accounts, which is indeed the case here.

By utilising the AccessFrom set and the set comparisons/predicates from Chapter4 we may discover the presence of this cycle. This is achieved by performing the following check statement with $U = \{\text{Laptop}\}$,

$$CHECK(U \subseteq \text{AccessFrom}\,(\text{AccessFrom}(U) \setminus U)). \tag{5.1}$$

This will return $\top$ if the set U can be accessed from the vertices which are transitively accessible from U which are not contained within U. This occurs only when a cycle originating at the vertices starting in the set U exits. It is also important to note that within this check statement, we do not refer to a single concrete element of the graph, instead referring to the generic set U. Thus, this rule can be reused with any given set of vertices to check for cycles.

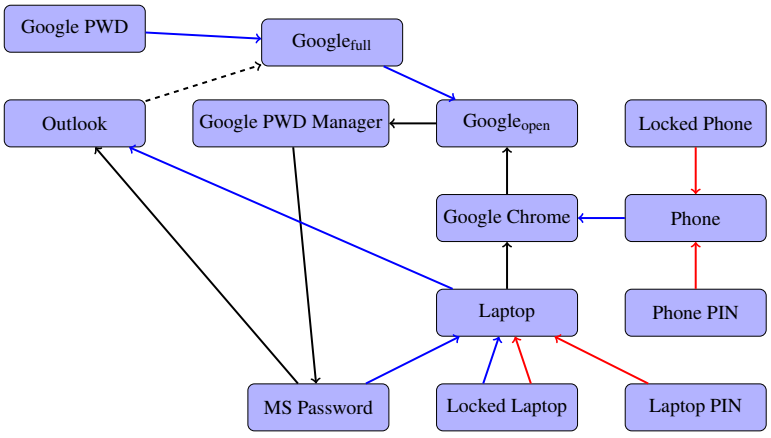


Figure 5.2: An account access graph without state showing an Outlook and Gmail account configured with an Outlook recovery email. This Outlook account has its password stored within the Google password manager.

We can take this account configuration and model it as a state-full account access graph within the tactic evaluation tool. To do this we need to make some assumptions around who initially has access to each vertex. We will take that initially the user has access to all credentials and is logged into their laptop and smartphone; while the attacker has no access to any accounts, credentials, or devices. Starting with a blank canvas, we

may create this environment within the tactics tool by performing the following tactics, which results in the graph shown in Figure 5.3a. Note that some components of these tactics have been removed for brevity. The full tactics, along with all the tactics used within this section can be found in Appendix C.

```
for all x in ['Google PWD', 'Outlook', ...]
do create_account('user', x);
add_access('user', ['Google PWD'], 'Google full', 'PWD');
                          ...
add_access('user', ['MS Password'], 'Outlook open', 'PWD');
(for all x in ['Outlook', 'Google PWD Manager',
'Google full', 'Google open', 'Google Chrome']
do lose_access('user', x))
```

We can see in the above commands that thanks to the inclusion of the FOR ALL shorthand in the new tool. This reduces the number of tactics required to generate the initial state of the system is significantly reduced when compared to previous systems. This reduction in input length reduces the opportunity for user error in inputting tactics and makes the system as a whole easier to use when conducting these evaluations.

5.4.1 The Attack

Now, suppose the user was to walk away from their laptop or smartphone while logged in. At this point, if an attacker could gain temporary access to this laptop or smartphone, they would be able to gain access to the user's Google password manager via the open session within the Chrome browser. While this would be a concern alone, due to the need for physical access to the machine this access alone is not especially interesting. This type of vulnerability was also noted by Aranza Trevino [34] in a blog post about the weakness of such integrated browser managers. However, due to the cycle of access introduced by the Outlook recovery method, if the attacker can simply copy the Microsoft password while they have access to the user's device, they would be able to gain persistence within the users' accounts. The user stepping away from their laptop or phone and then the attacker moving over to use it can be modelled by the tactic;

$$(\text{lose\_access}_{\text{User,Laptop}}; \text{lose\_access}_{\text{User,Locked Laptop}}; \text{disc\_access}_{\text{Attacker,Laptop}}) ||$$

$$(\text{lose\_access}_{\text{User,Phone}}; \text{lose\_access}_{\text{User,Locked Phone}}; \text{disc\_access}_{\text{Attacker,Phone}}).$$

Since this attack is the same regardless of which device the attacker can access. We will now continue without loss of generality by only considering the case where the attacker gains access to the user's laptop. If this failed, the attack using the phone would be identical, just with all references to 'Laptop' replaced with 'Phone'. Should a user have more devices with a similar configuration, this '||' tactic could be generalised to a for-one rule with the set {'Laptop', ...,'Phone'} containing all such devices. Then all the tactics conducted within this case study with 'Laptop' replaced with the chosen variable name could be included as the tactic to try.

The next step of the attack to be modelled is to open the Chrome browser and in some way record the Microsoft password contained within its password manager. In tactics,

this is modelled by repeatedly applying the gain access operation, and the resulting graph can be seen in Figure C.1.

$$\text{gain\_access}_{\text{Attacker,Google Chrome}}; \text{gain\_access}_{\text{Attacker,Google}_{\text{open}}};$$

$$\text{gain\_access}_{\text{Attacker,Google PWD Manager}}; \text{gain\_access}_{\text{Attacker,MS Password}}$$

From this stage, the attacker may use their own device when the user returns. At which point the following will be executed,

$$\text{lose\_access}_{\text{Attacker,Laptop}}; \text{lose\_access}_{\text{Attacker,Google Open}};$$

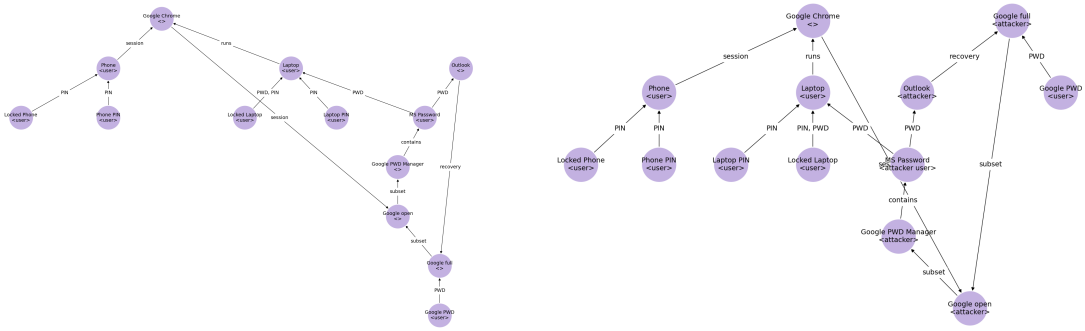
$$\text{lose\_access}_{\text{Attacker,Google Chrome}}; \text{disc\_access}_{\text{User,Locked Laptop}}; \text{gain\_access}_{\text{User,Laptop}}$$

The resulting graph can be seen in Figure C.2. However, since the attacker has the password to the users Microsoft account from any other device they may now perform,

$$\text{gain\_access}_{\text{Attacker,Outlook}}; \text{gain\_access}_{\text{Attacker,Google}_{\text{Full}}};$$

$$\text{gain\_access}_{\text{Attacker,Google}_{\text{open}}}; \text{gain\_access}_{\text{Attacker,Google PWD Manager}}.$$

Resulting in the attacker being able to access the Google password manager and all related credentials once again. The attacker could then use this to access all of the accounts the user has connected to the Google password manager, changing their passwords and gradually locking the user out. The graph following this third and final stage of the attack can be seen in Figure 5.3b, and all the tactics entered into the tactic evaluation tool can be seen in the appendix C. Many of the stated commands within this case study could be simplified to use a for all shorthand instead of a series of individual commands.



(a) Initial State.

(b) Final State.

Figure 5.3: Graphs generated by the tactic evaluation tool showing the initial and final stages of the attack conducted in the Google recovery method case study.

5.4.2 Recreating the Attack in Practice

This attack was replicated in the real world by creating a new Microsoft/Outlook account and then creating a Windows virtual machine (VM) signed into this account. Google Chrome was then installed on the VM and the instructions to create a new Google

account were followed. The Outlook address for the freshly created Microsoft account was used as Google's recovery email when prompted. Outlook was then visited using Chrome, saving the login details into the Google password manager when prompted.

After this setup, the attack was carried out by following the steps outlined above. I was indeed able to gain access to the Google account through the recovery email. Notably, this did not even force the reset of the Google account password. However, there was inconsistency in when it was successful, due to Google occasionally asking the user to re-login to Windows before viewing passwords. This often occurs when the password manager has not been used since the beginning of the current browser session. Other than a login notification being sent to the Gmail for the Google account, which would likely be ignored if the attacker was logging in from a similar location as the victim (i.e. the same café), there would be no signs of the breach to the user. This is also concerning as it is highly likely the details of many other accounts would be stored within the Google password manager.

Although reality provided some extra variability in the success of the attack, some aspects of this attack were made easier than expected in practice. This was mainly because the Chrome account management page would display enough characters of the recovery email to be recognisable as a username within the password manager. This allows an attacker with no prior knowledge of a user's configuration to simultaneously discover the username and password for the account recovery email.

5.4.3 Mitigating Risk and the Effect of 2FA

A simple low-tech way to mitigate this risk is simply not leaving devices unattended especially not while logged in. This prevents the opportunity for the attack to even take place. However, this would not always be a practical choice.

One possible technical mitigation for this attack would be to enable two-step verification on the Google account, requiring a user to enter more information when resetting the account, such as their full name or a code sent to a phone number. This makes the attack more difficult but it may be possible to gather this additional information either through OSINT [9] or while the attacker has access to the user's computer. If the user also introduced 2FA to their Outlook (recovery) account this would now completely prevent an attacker from logging in using only the password stored by Chrome.

To see why enabling 2FA on the recovery email prevents this attack we may introduce an SMS-based 2FA token. This token will then be made an extra requirement for accessing the Outlook email. The tactic which describes the user implementing this change is,

$$\text{gain\_access}_{\text{User,Outlook}};\text{create\_account}_{\text{User,2FA code}};$$

$$\text{add\_access}_{\text{'User', \{'2FA code'\}, 'Outlook', 'PWD'}};\text{add\_access}_{\text{'User', \{'Phone'\}, '2FA code', 'SMS'}};$$

$$\text{lose\_access}_{\text{'User', 'Outlook'}}.$$

Then trying to repeat the tactics within the attack as described within the case study, we now receive the premise error, "gain_access('attacker', 'Outlook') failed: attacker does

not have the required factors", which is thrown when the attacker attempts to perform

$$\text{gain\_access}_{\text{Attacker,Outlook}}$$

within the final stage of the attack. This is because the attacker will no longer possess all the required credentials to access this account. Re-applying the check-in equation 5.1, which now returns $\perp$, we can also see that the cycle has now been removed. Though it is possible the attack could still be carried out in conjunction with a SIM swap attack [24], it is clear the configuration security has been improved.

5.5 Exploitation of a Google OAuth Vulnerability

To reduce how often a user must sign into a service it is common for authentication cookies to be used. These cookies identify the user and allow a web service to authenticate them without the use of any other credentials. It has been an increasingly common attack vector for these cookies to be stolen[32], allowing an attacker to authenticate with services as an account owner and without entering any other credentials, even allowing access after credentials have been changed if the cookie is not revoked. This is what happened when in March 2023 the YouTube channel Linus Tech Tips was breached through the theft of a cookie. The stolen cookie provided administrator permissions to their channel[31], with the attackers continuing to have access to the accounts even after passwords were changed.

An advanced version of this attack technique, which makes use of a vulnerable Google OAuth endpoint named 'MultiLogin', was first described by the threat actor PRIMSA on the 20^{th} October 2023[20]. This technique claims to allow an attacker to restore expired cookies for Google services providing persistent access, even if the session is disrupted or the password is reset [22]. This is especially significant, as it allows an attacker to maintain access to a user's Google services for much longer than in typical Cookie-based attacks. An account access graph displaying the core components of this attack against a Google account with 2FA enabled can be seen in Figure 5.4.

5.5.1 The Attack

The attack begins, as described by Connor Jones of The Register [19], with the attacker gaining access to the user's Chrome files. This access may be gained through a malicious download, link or one of many other methods. We, therefore, assume, for our initial state, that the attacker has access to the Chrome Files and MultiLogin endpoint (since MultiLogin is a public endpoint either user could query). While the user has access to all vertices. The resulting initial state can be seen in figure 5.5a, and as with the previous case study all commands evaluated in the tool can be seen in Appendix C.2.

With access to the Chrome Files an attacker may then steal all the encrypted_tokens and GAIA (Google Accounts and ID Administration) identities associated with all accounts currently logged into Chrome[20]. Similarly, the encryption keys for the tokens are stored within the UserData directory of Chromes LocalState [20] and thus may also be stolen by the attacker. When considering the attack for one chosen account this stage is

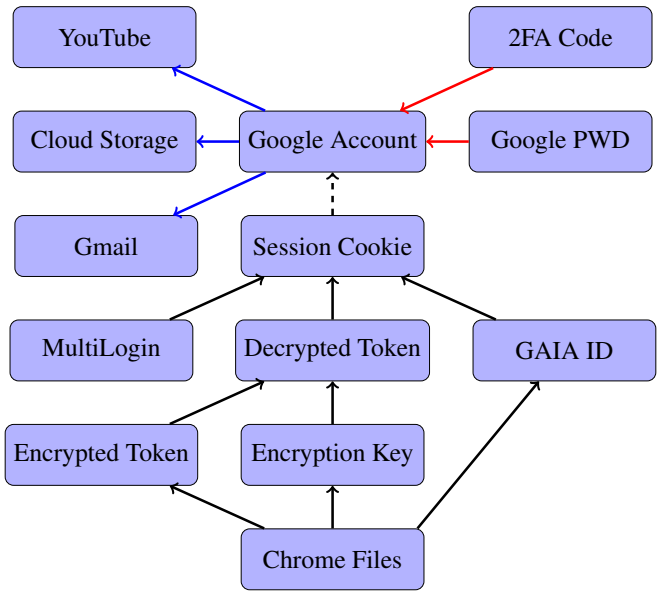


Figure 5.4: An account access graph without state showing the relevant components of the MultiLogin OAuth API configuration .

modeled by the tactic,

FOR ALL x IN [Encrypted Token, Encryption Key, GAIA ID] DO $\text{gain\_access}_{\text{attacker},x}$.

The attacker may then decrypt the user's token with the stolen key. This is modelled once again by the operation "$\text{gain\_access}_{\text{attacker, Decrypted Token}}$".

MultiLogin is a publicly accessible endpoint that takes a vector of Token and GAIA ID's then enables the regeneration of Google service cookies [20]. While this endpoint was initially intended for synchronizing Google accounts across services since the attacker now has both the user's Decrypted Token and GAIA ID they may use it to regenerate valid cookies. Each of these cookies then provides access to the user's Google account without the use of a password or the 2FA code. The graph produced after this final stage is shown in Figure 5.5b, and the process can be modelled via the tactic below.

$$\text{gain\_access}_{\text{attacker, Session Cookie}}; \text{gain\_access}_{\text{attacker, Google Account}}.$$

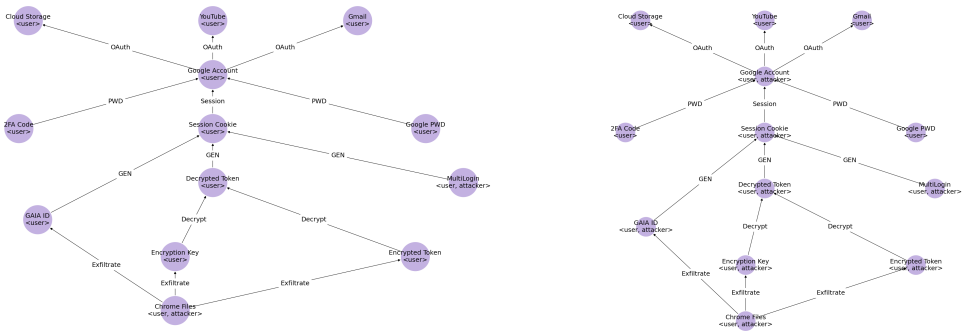
From this Google account, the attacker can access many different services that utilise Google for authentication. This includes most Google services such as YouTube, Gmail and Google Cloud as well as many third-party services authenticated with OAuth. The attacker can breach each of these accounts with the following tactic (where OAuth is the label used for all blue edges within Figure 5.4, which each only requires one factor),

$$\text{FOR ALL } x \text{ IN } \text{Succ}_{\text{OAuth , Google Account}} \cap \text{AccessFrom}(\{\text{Google Acount}\}) \text{ DO}$$

$$\text{gain\_access}_{\text{attacker, } x}.$$

Notice that with this tactic we are not required to specify the names of the accounts the attacker will access, instead simply all relevant vertices are searched for then

accessed, demonstrating some of the utility simple queries provide for conducting security analyses. Using this new access an attacker could steal many different forms of personal information, from photos stored in Google Photos to sensitive information shared via email. This also allows an attacker to read and respond to any recovery emails sent to the related Gmail account, potentially allowing accounts which don't use Google for authentication to also be breached, mirroring the attack within the previous case study.



(a) Initial State. (b) Final State .

Figure 5.5: The initial and final states of the account access graph with state generated by the tactic evaluation tool for the Google OAuth case study.

5.5.2 Regaining Control After the Attack

In response to this attack, Google stated that if a user thinks they have been breached this way they should log out of the affected browser and revoke all active Google sessions [19]. This should be done in conjunction with removing any malicious software on a user's computer, to remove the attacker's access from the Chrome Files. This would have the effect of invalidating the token stolen by the attacker and replacing it with a new one. We can model the action of malware removal and replacement of the token with the following command (where "GEN" is the label associated with the edges incident on Session Cookie),

$$\text{lose\_access}_{\text{attacker, Chrome Files}}; \text{create\_account}_{\text{user, New Token}};$$

$$\text{rem\_access}_{\text{user, Session Cookie, GEN}}; \text{lose\_access}_{\text{attacker, Session Cookie}}$$

$$\text{add\_access}_{\text{user, \{MultiLogin, New Token, GAIA ID\},Session Cookie, GEN}}.$$

After this change when the attacker attempts to regenerate a cookie, they will now receive the error,"gain_access('attacker', 'Session Cookie') failed: attacker does not have the required factors".

According to Karthic [20] the MultiLogin endpoint will still allow a cookie to be refreshed once after the password of the account has been changed. Therefore, if a user were to simply change their password and not revoke the stolen token an attacker's access would not be removed. This can be seen using the tactic evaluation tool by first changing the password with the tactic,

$$\text{create\_account}_{\text{user, New PWD}}; \text{rem\_access}_{\text{user, Google Account, PWD}};$$

$$\text{add\_access}_{\text{user, \{New PWD, 2FA Token\}, Google Account, PWD}\cdot}$$

where "PWD" is the label used on the red edges of Figure 5.4. Then evaluate the following predicate to check if the decrypted token will still give an attacker access to the Google account.

CHECK(Google Account IN AccessFrom(Decrypted Token)).

Performing this with the tactic evaluation tool returns $\top$ verifies that changing the password alone is insufficient. Since MultiLogin only allows cookies to be regenerated once the password is changed after this ability has been used once the commands outlined for replacing the token are evaluated. This locks the attacker out should they no longer have access to the user's Chrome Files.

5.5.3 Verification of Attack Feasibility

It was not possible to verify the whole attack chain for this case study due to the required use of specialist information-stealing malware, for which there would be no ethical way to attain. However, it was possible to test the behaviour of Google sessions when account passwords were changed. This is an important component to verify, since the ability to maintain access to a user's account, even after a password change is key to the utility of this attack over generic cookie theft. To perform this verification, a new Google account is created with multi-factor authentication enabled. Then this account is used to log into Chrome on two separate devices. One of these devices is connected to a VPN to simulate an attacker in another location. To model third-party services authenticated with OAuth a new Epic Games account is also created by using the login with Google option.

It was found that upon password reset the device on which it was changed remained logged in, while the second device was immediately logged out of any first-party services. However, if the second device had recently authenticated with a third-party service using OAuth these services remained logged in. This second device mirrors the behaviour of an attacker simply stealing the user's password. Since the tokens stolen within the attack correspond to that of the first device it is indeed feasible that this attack would still be possible, as this browser remains authenticated. Since the attackers stolen by the attacker belong to the first device's browser, which remains authenticated, it is indeed possible the attacker may be able to maintain access.

5.6 Heuristic Evaluation of Tool Usability

To provide a structured evaluation of the developed tools' usability the ten heuristic principles as defined by Nielsen [26, 27], a description of which can be seen in Appendix C.3, were used. For these principles to be properly applied they should be evaluated by around 5 users, however, the tool requires a user to posses a good understanding of the tactics language before they can utilise it making any such user study impractical. Instead, the analysis was performed only once; while this may reduce its effectiveness, it still provides a useful insight into the tool's usability.

Within this evaluation, all ten characteristics were given a score out of 5 and any particular highlights or failings were noted. Overall this resulted in an average score of 3.2 out of 5. The largest weaknesses according to this rubric were User control and Freedom as well as Error prevention which scored just 1 and 2 out of 5 respectively.

Due to the use of only a single line for tactic input, it was found that errors were easy to make, especially when using long tactics, since finding such errors would require scrolling back through the line and meticulously searching for errors by hand. While error messages do highlight syntax errors, it remains easy to make logical errors, such as providing the wrong name for a vertex in an operation. Similarly, the leading cause of the poor User control and Freedom score was that if an earlier version of the graph was not saved, correcting a mistake could require resetting the graph and starting again, or at least having to enter numerous other tactics to undo any changes made.

Nonetheless, throughout the case studies and evaluation, the tool provided a meaningful way to test that tactics were well formed (do not throw errors) and would perform the intended changes to an account access graph. This is something that previously would have to be manually checked, which can be an error-prone process.

5.7 Improvements Following Evaluation

To remedy the shortcomings in User Control and Freedom demonstrated by the heuristic evaluation an undo button was added to the interface. Since this is mostly intended for correcting small errors it was implemented by storing one previous version of the account access graph, so when the undo button is pressed the old version is loaded. Another more complicated design might have been storing the ASTs for each tactic executed in a heap. Then each AST class would be provided with an undo method that could then be called much like the evaluate method. However, due to time constraints, this approach was not taken. Similarly, a redo button was not implemented, though due to executing an input not clearing the tactic it was found that this is rarely needed.

It was highlighted in this evaluation that for longer chains of tactics the single-line input was inadequate. This was therefore replaced with a multi-line input and moved to be at the left of the graph display, allowing for long tactics to be entered without significant difficulty. I also found that the grey nodes combined with occasionally small text made the graphs difficult to read. So, the font size was increased and the nodes were changed to purple. This updated design can be seen in Appendix C.3.

Following these alterations, the steps involved in the case studies were recreated and the heuristic evaluation was reapplied. Upon reevaluation, the software scored an average of 4 out of 5. This is driven by a substantial improvement to the Error Prevention (from 2 to 4) and User Control and Freedom (1 to 3) of the software, as both the number of errors made in conducting the analysis dropped and the ease of undoing remaining mistakes reduced.

Chapter 6

Conclusions and Future Work

Within this project, a tool that allows for security analyses to be conducted using account access graphs with state was implemented. This tool provides for tactics to be inputted, evaluated and then have the resulting graph state displayed to the user. This goal was achieved through a combination of a Python-based graph model and user interface with an ANTLR grammar and parser. Requiring several thousand lines of code this represents the first known implementation of such a system which incorporates the IF and FOR shorthand's for the tactics language as well as the first such implementation to evaluate tactics using backtracking. Beyond this a new extension to the tactics language was designed and implemented within the tactics tool. This allowed for simple queries to be performed on a graph state for elements which match certain properties and tactics evaluated based on the resulting components through either indexing or for loops.

To verify the quality of the implementation, and make any future changes to the system easier, a suite of unit tests were devised that provide a good coverage (92%) of the code base and a wide array of edge cases for language parsing/evaluation. These tests also include cases where backtracking affects the resulting graph, and where the IF and FOR shorthand's are used.

Since this tool is intended to aid in conducting security analyses, it was used to model and analyse two case studies. Within this, we were able to successfully use the tool to verify the possibility and capability of two different attacks, both of which utilised a secondary authentication mechanism to bypass the primary method and conduct an account takeover attack. Throughout these case studies, attention was paid to the usability of the tool for conducting such security analyses and a formal framework was used to conduct a heuristic evaluation. The tool also allowed for the detection and correction of logical errors within initial versions of many tactics used within this report.

6.1 Challenges Faced

The most significant challenge faced during this project was the development of the ANLR grammar and syntax for the tactics language. This required the interpretation,

then conversion into code, of big-step semantics and grammar structures. Due to my lack of knowledge and experience in language design and parsing this process was notably difficult. However, through the use of multiple online guides and resources an understanding of these topics was built and experience was gained through the implementation of the tactics language. This learning was then applied by designing and implementing the querying extension to the tactics language.

6.2 Limitations and Future Work

The querying extension to the tactics language outlined within this report does allow simple queries to be written, and tactics defined that utilise the results of these queries. However, to make these queries more relevant to real-world situations and allow for more general attack patterns to be defined a fourth map $T : V \to \mathcal{T}$, where $\mathcal{T}$ represents a set of possible vertex types, could be introduced to the account access graphs with state. This would result in an account access graph with state and type represented by a quintuple of the form (V, E, A, T), with a natural set of such types $\mathcal{T}$ being

$$\{account, device, credential\}.$$

This introduction of types would allow for much more specificity in generic predicates over the graph, and thus stronger search capability. For example, we would then be able to select all vertices $v \in V$ of type *account* where for some $l \in \mathcal{L}$, $\text{Pred}_{l,v}$ contains only one credential. All of these selected accounts would be such that if an attacker was able to steal, phish, or buy the appropriate credentials, they would be able to access them. Future work could use this extension to express the configuration patterns required to conduct common attacks, such as SIM swapping.

Another limitation is the graph visualisation system. It was found in the evaluation that the current algorithm for vertex positioning was the largest contributor to end-to-end latency by a large margin. Furthermore, it rarely produces a perfect positioning of the vertices when visualising account access graphs with state. Future work could seek to integrate an interactive graph diagram that allows a user to move vertices. Alternatively a new positioning algorithm could be designed focusing on account access graphs, this could use the additional information included in the graphs to produce better visualisations and could focus on reducing position generation time.

A final area for future work is related to the usability of the tactic input system. The current system uses a simple text input, however, as tactics grow it would be useful if syntax highlighting and auto-fill suggestions based on the tactics language and current graph state could be shown to users. These changes would reduce the time needed to input tactics as well as help to reduce the number of errors made while inputting tactics, helping to make the tool more usable.

Bibliography

[1] IBM Documentation. https://www.ibm.com/docs/en/rdfi/9.6.0?topic=le-tokens, March 2021. (Accessed on 12/01/2024).

[2] Melvin Abraham, Michael Crabb, and Saša Radomirović. "i'm doing the best i can.". In Simon Parkin and Luca Viganò, editors, *Socio-Technical Aspects in Security*, pages 86–107, Cham, 2022. Springer International Publishing.

[3] Luca Arnaboldi, David Aspinall, Christina Kolb, and Saša Radomirović. Tactics for account access graphs. In Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, editors, *Computer Security – ESORICS 2023*, pages 452–470, Cham, 2024. Springer Nature Switzerland.

[4] Andy Balaam. How to Write a Programming Language: Part 2, The Parser. https://accu.org/journals/overload/26/146/balaam_2532, January 2024. (Accessed on 10/01/2024).

[5] Brian Barrett. How twitter ceo jack dorsey's account was hacked. https://www.wired.com/story/jack-dorsey-twitter-hacked/. (Accessed on 28/02/24).

[6] Marc-Philippe Bartholomä. Automated mining of user account access graphs. Master thesis, ETH Zurich, Zurich, 2021.

[7] Chris J Date and Hugh Darwen. *A Guide to the SQL Standard*, volume 3. Addison-Wesley New York, 1987.

[8] Jan Peter Falk. A frontend for account access graphs. Master thesis, ETH Zurich, Zurich, 2019.

[9] Ritu Gill. What is osint? https://www.sans.org/blog/what-is-open-source-intelligence/. (Accessed on 11/03/2024).

[10] statcounter globalstats. Browser market share worldwide. https://gs.statcounter.com/browser-market-share. (Accessed on 28/01/2024).

[11] Google. Make your account more secure. https://support.google.com/accounts/answer/46526?hl=en. (Accessed on 28/02/26).

[12] Google. Use passwords across your devices. https://support.google.com/chrome/answer/6197437?hl. (Accessed on 28/01/2024).

[13] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[14] Sven Hammann. *Secure, Private, and Personal: Advancing Digital Identity*. Doctoral thesis, ETH Zurich, Zurich, 2021.

[15] Sven Hammann, Michael Crabb, Saša Radomirović, Ralf Sasse, and David Basin. "i'm surprised so much is connected". In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery.

[16] 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.

[17] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90, 2007.

[18] Neil J. Rubenking. Don't let google manage your passwords. `https://uk.pcmag.com/password-managers/145831/warning-dont-let-google-manage-your-passwords`. (Accessed on 31/01/2024).

[19] Conor Jones. Google password resets not enough to stop these info-stealing malware strains. `https://www.theregister.com/2024/01/02/infostealer_google_account_exploit/`. (Accessed on 06/03/2024).

[20] Pavan Karthick M. Compromising google accounts: Malwares exploiting undocumented oauth2 functionality for session hijacking. `https://www.cloudsek.com/blog/compromising-google-accounts-malwares-exploiting-undocumented-oauth2-functionality-for-session-hijacking`. (Accessed on 05/03/2024).

[21] Fredrik Lundh. An introduction to tkinter. *URL: www. pythonware. com/library/tkinter/introduction/index. htm*, 1999.

[22] Tom McKay. Hackers are abusing a google oauth endpoint to hijack user sessions. `https://www.itbrew.com/stories/2024/01/19/hackers-are-abusing-a-google-oauth-endpoint-to-hijack-user-sessions`. (Accessed on 05/03/2024).

[23] Microsoft. How to help keep your microsoft account safe and secure. `https://support.microsoft.com/en-us/account-billing/how-to-help-keep-your-microsoft-account-safe-and-secure-628538c2-7006-33bb-5ef4-c917657362b9`. (Accessed on 28/02/2024).

[24] Microsoft. What is sim swapping & how does the hijacking scam work? `https://www.microsoft.com/en-us/microsoft-365-life-hacks/privacy-and-safety/what-is-sim-swapping`. (Accessed on 03/02/2024).

[25] NCSC. Top tips for staying secure online. https://www.ncsc.gov.uk/collection/top-tips-for-staying-secure-online. (Accessed on 29/09/2023).

[26] Jakob Nielsen. 10 usability heuristics for user interface design. https://www.nngroup.com/articles/usability-heuristics-complex-applications/. (Accessed on 03/03/2024).

[27] Jakob Nielsen. Finding usability problems through heuristic evaluation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, page 373–380, New York, NY, USA, 1992. Association for Computing Machinery.

[28] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[29] Terence Parr. ANTLR Lab. http://lab.antlr.org, November 2022. (Accessed on 13/01/2024).

[30] Terence Parr. Antlr (another tool for language recognition). https://www.antlr.org, 2024. (Accessed on 10/01/2024).

[31] Linus Sebastian. My channel was deleted last night. https://www.youtube.com/watch?v=yGXaAWbzl5A. (Accessed on 05/03/2024).

[32] Karishma Sundaram. Cookie stealing in wordpress: Understanding the risks and consequences. https://www.malcare.com/blog/cookie-stealing. (Accessed on 26/02/24).

[33] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2007.

[34] Aranza Trevino. Are browser password managers safe? https://www.keepersecurity.com/blog/2022/11/04/are-browser-password-managers-safe/. (Accessed on 31/01/2024).

[35] Blair Walker. Graph-driven security: An interpreter for an account access language with neo4j integration, 2023. MSc Project. University of Edinburgh.

Appendix A

Background

A.1 Formal Definition of properties for Account Access Graphs

The validity of assertions is defined inductively, stating from atomic propositions [3]:

- $\langle V,E,A\rangle \vDash \text{is\_account}(v)$ if $v \in V$
- $\langle V,E,A\rangle \vDash \text{has\_access}_a(v)$ if $v \in V$ and $a \in A(v)$
- $\langle V,E,A\rangle \vDash \text{could\_access}_a(v)$ if $v \in V$ and $\exists l,u.\, l \in E(u,v) \wedge \forall x \in V.\, l \in E(x,v) \Rightarrow a \in A(x)$.
- $\langle V,E,A\rangle \vDash \text{uses\_method}_l(u,v)$ if $u,v \in V$ and $l \in E(u,v)$.
- $\langle V,E,A\rangle \vDash \phi_1 \wedge \phi_2$ if $\langle V,E,A\rangle \models \phi_1$ and $\langle V,E,A\rangle \models \phi_2$.
- $\langle V,E,A\rangle \vDash \phi_1 \vee \phi_2$ if $\langle V,E,A\rangle \models \phi_1$ or $\langle V,E,A\rangle \models \phi_2$.
- $\langle V,E,A\rangle \vDash \neg\phi$ if it does not hold that $\langle V,E,A\rangle \models \phi$.
- $\langle V,E,A\rangle \models$ true always.

A.2 Operation Definitions

The operations that can be used to modify an account access graph, as introduced by Arnaboldi et al. [3] are defined bellow.

$$\frac{\langle V,E,A\rangle \vDash \text{could\_access}_a(u)}{\langle V,E,A\rangle \overset{\text{gain\_access}_{a,v}}{\rightsquigarrow} \langle V,E,A[v \mapsto a]\rangle} \quad \text{(gain)}$$

$$\langle V,E,A\rangle \overset{\text{disc\_access}_{a,v}}{\rightsquigarrow} \langle V,E,A[v \mapsto a]\rangle \quad \text{(disc)}$$

$$\langle V,E,A\rangle \overset{\text{lose\_access}_{a,v}}{\rightsquigarrow} \langle V,E,A[v \setminus a]\rangle \quad \text{(lose)}$$

$A[v \mapsto a]$ 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 $A(v)$.

$$\langle V, E, A \rangle \overset{\text{create\_account}_{a,v}}{\rightsquigarrow} \langle V \cup \{v\}, E, A[v \mapsto a] \rangle \tag{create}$$

$$\frac{\langle V, E, A \rangle \vDash \text{has\_access}_a(v)}{\langle V, E, A \rangle \overset{\text{del\_account}_{a,v}}{\rightsquigarrow} \langle V \setminus \{v\}, E \setminus v, A|_{V \setminus \{v\}} \rangle} \tag{delete}$$

$$\frac{\langle V, E, A \rangle \vDash \text{has\_access}_a(v) \quad E'(x,y) = \begin{cases} \{l\} \text{ if } x \in \{u_1, \ldots, u_n\} \text{ and } y = v \\ \{\} \text{ otherwise} \end{cases}}{\langle V, E, A \rangle \overset{\text{add\_access}_{a,\{u_1,\ldots,u_n\},v,l}}{\rightsquigarrow} \langle V, E \uplus E', A \rangle} \tag{add}$$

$$\frac{\langle V, E, A \rangle \vDash \text{has\_access}_a(v)}{\langle V, E, A \rangle \overset{\text{rem\_access}_{a,v,l}}{\rightsquigarrow} \langle V, E \setminus \{(,v)\} \setminus \{l\}, A \rangle} \tag{remove1}$$

$$\frac{\langle V, E, A \rangle \vDash \text{has\_access}_a(v) \quad \langle V, E, A \rangle \vDash \text{uses\_method}_l(v,u)}{\langle V, E, A \rangle \overset{\text{rem\_access}_{a,u,l}}{\rightsquigarrow} \langle V, E \setminus \{(,u)\} \setminus \{l\}, A \rangle} \tag{remove2}$$

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

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

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

Appendix B

Tactics Grammar

B.1 Original Lexer Grammar

```
AND: ';' ;
OR: '||' ;
LPAR: '(' ;
RPAR: ')' ;
LBRA: '[' ;
RBRA: ']' ;
COMMA: ',' ;

TOP: 'TOP' ;
BOT: 'BOT' ;
CON: 'CON' | '^' ;
DIS: 'DIS' | 'v';
NEG: 'NEG' | '!' | '~' ;

IS_ACCOUNT: 'IS_ACCOUNT'|'is_account'
|'is_acc'|'is_ac' ;

HAS_ACCESS: 'HAS_ACCESS'|'has_access'
|'has_acc'|'has_ac' ;

COULD_ACCESS: 'COULD_ACCESS'|'could_access'
|'could_acc'|'could_ac' ;

USES_METHOD: 'USES_METHOD'|'uses_method'
|'uses_meth'|'uses_met'|'uses_me'|'uses_m' ;

CHECK: 'CHECK'|'check' ;

GAIN_ACCESS: 'GAIN_ACCESS'|'gain_access'
|'gain_acc'|'gain_ac' ;
```

```
DISC_ACCESS: 'DISC_ACCESS'|'disc_access'
|'disc_acc'|'disc_ac' ;

LOSE_ACCESS: 'LOSE_ACCESS'|'lose_access'
|'lose_acc'|'lose_ac' ;

CREATE_ACCOUNT: 'CREATE_ACCOUNT'|'create_account'
|'create_acc'|'create_ac' ;

DEL_ACCOUNT: 'DEL_ACCOUNT'|'del_account'
|'del_acc'|'del_ac' ;

ADD_ACCESS: 'ADD_ACCESS'|'add_access'
|'add_acc'|'add_ac' ;

REM_ACCESS_1: 'REM_ACCESS_1'|'rem_access_1'
|'rem_acc_1'|'rem_ac_1' ;

REM_ACCESS_2: 'REM_ACCESS_2'|'rem_access_2'
|'rem_acc_2'|'rem_ac_2' ;

STRING: '"' ~["]* '"' | '\'' ~[']* '\'' ;

WHITESPACE: [ \t\r\n]+ -> skip ;
```

B.2 Original Parser Rules

```
   tactic: t EOF ;

t:t AND t # AndOp
 |t OR t # OrOp
 | LPAR t RPAR # Parenthesized
 | t_simple # Simple
 ;

t_simple: b #Bool
 | alpha #Operation
 | CHECK LPAR proposition RPAR # CHECK
 | FOR VAR IN arg_array DO t # For
 | IF proposition THEN t ELSE t # If
 ;

 b : TOP #Top
 | BOT #Bot
 ;

 arg: STRING #StringArg
;
```

```
arg_array: '[' arg (COMMA arg)* ']' # ArgumentArray
;

alpha: GAIN_ACCESS LPAR arg COMMA arg RPAR # GainAccess
 | DISC_ACCESS LPAR arg COMMA arg RPAR # DiscAccess
 | LOSE_ACCESS LPAR arg COMMA arg RPAR # LoseAccess
 | CREATE_ACCOUNT LPAR arg COMMA arg RPAR # CreateAccount
 | DEL_ACCOUNT LPAR arg COMMA arg RPAR # DeleteAccount
 | ADD_ACCESS LPAR arg COMMA arg_array COMMA arg COMMA arg RPAR
 # AddAccess
 | REM_ACCESS_1 LPAR arg COMMA arg COMMA arg RPAR
 # RemoveAccess1
 | REM_ACCESS_2 LPAR arg COMMA arg COMMA arg RPAR
 # RemoveAccess2
 ;


proposition: NEG atomic_proposition # NegationProp
 | NEG LPAR proposition RPAR # Negation
 |proposition CON proposition # ConOp
 | proposition DIS proposition # DisOp
 | LPAR proposition RPAR # ParenthesizedProp
 | atomic_proposition # AtomicProp
 ;

atomic_proposition: IS_ACCOUNT LPAR arg RPAR # IsAccount
 | HAS_ACCESS LPAR arg COMMA arg RPAR # HasAccess
 | COULD_ACCESS LPAR arg COMMA arg RPAR # CouldAccess
 | USES_METHOD LPAR arg COMMA arg COMMA arg RPAR # UsesMethod
 ;
```

B.3 Extended Lexer Grammar

```
lexer grammar tacticLexer;

AND: ';' ;
OR: '||' ;
LPAR: '(' ;
RPAR: ')' ;
LBRA: '[' ;
RBRA: ']' ;
LSET: '{' ;
RSET: '}' ;
COMMA: ',' ;

TOP: 'TOP' ;
BOT: 'BOT' ;
```

```
CON: 'CON' | 'AND' ;
DIS: 'DIS' | 'OR' ;
NEG: 'NEG' | '!' | '~' ;

EQUAL: '=';

EQ: '==';
NEQ: '!=';
LT: '<';
LEQ: '<=';
GT: '>';
GEQ: '>=';

LET: 'let'| 'LET';
UNION: 'UNION' | 'union' ;
INTER: 'INTER' | 'inter' ;

LENGTH: 'len' | 'LEN';


IS_ACCOUNT: 'IS_ACCOUNT'|'is_account'|'is_acc'|'is_ac' ;
HAS_ACCESS: 'HAS_ACCESS'|'has_access'|'has_acc'|'has_ac' ;
COULD_ACCESS: 'COULD_ACCESS'|'could_access'|'could_acc'|'could_ac' ;
USES_METHOD: 'USES_METHOD'|'uses_method'|'uses_meth';
TRANS_ACCESS: 'transitive_access' |'TRANS_ACCESS' | 'trans_access'
| 'TRANSITIVE_ACCESS';


CHECK: 'CHECK'|'check' ;

GAIN_ACCESS: 'GAIN_ACCESS'|'gain_access'|'gain_acc'|'gain_ac' ;
DISC_ACCESS: 'DISC_ACCESS'|'disc_access'|'disc_acc'|'disc_ac' ;
LOSE_ACCESS: 'LOSE_ACCESS'|'lose_access'|'lose_acc'|'lose_ac' ;
CREATE_ACCOUNT: 'CREATE_ACCOUNT'|'create_account'|'create_acc'
|'create_ac' ;
DEL_ACCOUNT: 'DEL_ACCOUNT'|'del_account'|'del_acc'|'del_ac' ;
ADD_ACCESS: 'ADD_ACCESS'|'add_access'|'add_acc'|'add_ac' ;
REM_ACCESS_1: 'remove_access_1'|'REM_ACCESS_1'|'rem_access_1'|'rem_acc_1'
|'rem_ac_1' ;
REM_ACCESS_2: 'remove_access_2'|'REM_ACCESS_2'|'rem_access_2'|'rem_acc_2'
|'rem_ac_2' ;

// for shorthand
FORALL: 'FORALL'| 'FOR ALL' | 'forall' | 'Forall' | 'For All' | 'ForAll'
| 'for_all'| 'for all' ;
FORONE: 'FORONE'| 'FOR ONE' |'forone' | 'Forone' | 'For One' | 'ForOne'
| 'for_one'| 'for one' ;
IN: 'IN'| 'in' | 'In' ;
DO: 'DO'| 'do' | 'Do' | ':' ;
```

```
TRY: 'TRY'| 'try' | 'Try' ;
// if then else shorthand
IF : 'IF'| 'if' | 'If' ;
THEN : 'THEN'| 'then' | 'Then' ;
ELSE : 'ELSE'| 'else' | 'Else' ;
STRING: '"' ~["]* '"' | '\'' ( ~'\'' | '\'\'' )* '\'' ;


INT: [0-9]+;

// fragment DIGIT : '[0-9]';

PLUS:'+';
MINUS:'-';
MULTI:'*';
MOD: '%';
DIV: '/'|'//';
SETMINUS: '\\';


SELECT: 'SELECT';
WHERE: 'WHERE';
FROM: 'FROM';



VERTS: '$V';
EDGES: '$E';
ACCESS: '$A';
PRED: 'PRED';
SUCC: 'SUCC';
ACCESS_FROM: '$AccessFrom' | '$AF' | '$ACCESS_FROM' | '$access_from';

UNDERSCORE: '_';
VAR: [a-zA-Z0-9_]+ ;

WHITESPACE: [ \t\r\n]+ -> skip ;
```

B.4 Definition of evaluation for SELECT expressions

Definition 7. Evaluation of Set operations. The evaluation of the S rules are defined inductively by the following rules.

$$\frac{\langle\sigma\rangle S_1 \Downarrow \{a_1, a_2, \ldots, a_n\}\langle\sigma'\rangle \ \langle\sigma\rangle S_2 \Downarrow \{a_1, a_2, \ldots, a_i, b_1, \ldots, b_j\}\langle\sigma''\rangle}{\langle\sigma\rangle S_1 \cap S_2 \Downarrow \{a_1, \ldots, a_i\}\langle\sigma'\rangle} \quad \text{(INTER)}$$

$$\frac{\langle\sigma\rangle S_1 \Downarrow \{a_1, a_2, \ldots, a_n\}\langle\sigma'\rangle \ \langle\sigma\rangle S_2 \Downarrow \{a_1, a_2, \ldots, a_i, b_1, \ldots, b_j\}\langle\sigma''\rangle}{\langle\sigma\rangle S_1 \setminus S_2 \Downarrow \{a_{i+1}, \ldots, a_n\}\langle\sigma'\rangle} \quad \text{(MINUS)}$$

Definition 8. Evaluation of exp on the integers . The evaluation of the numerical expressions is once again defined inductively by the following rules.

$$\frac{\langle\sigma\rangle exp_1 \Downarrow n_1\langle\sigma'\rangle \quad \langle\sigma\rangle exp_2 \Downarrow n_2\langle\sigma''\rangle}{\langle\sigma\rangle exp_1 + exp_2 \Downarrow n_1 + n_2\langle\sigma'\rangle} \quad \text{(PLUS)}$$

$$\frac{\langle\sigma\rangle exp_1 \Downarrow n_1\langle\sigma'\rangle \quad \langle\sigma\rangle exp_2 \Downarrow n_2\langle\sigma''\rangle}{\langle\sigma\rangle exp_1 - exp_2 \Downarrow n_1 - n_2\langle\sigma'\rangle} \quad \text{(MINUS)}$$

$$\frac{\langle\sigma\rangle exp_1 \Downarrow n_1\langle\sigma'\rangle \quad \langle\sigma\rangle exp_2 \Downarrow n_2\langle\sigma''\rangle}{\langle\sigma\rangle exp_1 * exp_2 \Downarrow n_1 * n_2\langle\sigma'\rangle} \quad \text{(TIMES)}$$

$$\frac{\langle\sigma\rangle exp_1 \Downarrow n_1\langle\sigma'\rangle \quad \langle\sigma\rangle exp_2 \Downarrow n_2\langle\sigma''\rangle}{\langle\sigma\rangle exp_1 \ / \ exp_2 \Downarrow \lfloor\frac{n_1}{n_2}\rfloor\langle\sigma'\rangle} \quad \text{(DIVIDE)}$$

$$\frac{\langle\sigma\rangle exp_1 \Downarrow n_1\langle\sigma'\rangle \quad \langle\sigma\rangle exp_2 \Downarrow n_2\langle\sigma''\rangle}{\langle\sigma\rangle exp_1 \ \% \ exp_2 \Downarrow n_1 \bmod n_2\langle\sigma'\rangle} \quad \text{(MOD)}$$

B.5 Extended Parser Rules

```
parser grammar tacticParser;

options {
    tokenVocab = tacticLexer;
}

// Define the entry point for the parser
start: tactic EOF | search EOF | exp EOF;

tactic: t EOF
;

t:t AND t # TOp
 |t OR t # TOp
 | LPAR t RPAR # Parenthesized
 | t_simple # Simple
 ;

t_simple: b #Bool
 | alpha #Operation
 | CHECK LPAR proposition RPAR # CHECK
```

```
 | FORALL VAR IN search DO t # ForAll
 | FORONE VAR IN search TRY t # ForOne
 | LET VAR EQUAL search # SearchAssign
 | IF proposition THEN t ELSE t # If
 ;

 b : TOP
 | BOT
 ;

 arg: STRING
 | VAR
;

arg_array: '[' (arg (COMMA arg)*)? ']' # ArgumentArray
| LSET (arg (COMMA arg)*)? RSET # ArgumentArray
;

alpha: GAIN_ACCESS LPAR arg COMMA arg RPAR #GainAccess
 | DISC_ACCESS LPAR arg COMMA arg RPAR #DiscAccess
 | LOSE_ACCESS LPAR arg COMMA arg RPAR #LoseAccess
 | CREATE_ACCOUNT LPAR arg COMMA arg RPAR #CreateAccount
 | DEL_ACCOUNT LPAR arg COMMA arg RPAR #DeleteAccount
 | ADD_ACCESS LPAR arg COMMA search COMMA arg COMMA arg RPAR #AddAccess
 | REM_ACCESS_1 LPAR arg COMMA arg COMMA arg RPAR #RemoveAccess1
 | REM_ACCESS_2 LPAR arg COMMA arg COMMA arg RPAR #RemoveAccess2
 ;


proposition: proposition (CON | DIS) proposition # PropOp
 | NEG LPAR proposition RPAR #PropOp
 | LPAR proposition RPAR # ParenthesizedProp
 | NEG atomic_proposition #PropOp
 | atomic_proposition # AtomicProp
 ;

atomic_proposition: IS_ACCOUNT LPAR arg RPAR #IsAccount
 | HAS_ACCESS LPAR arg COMMA arg RPAR #HasAccess
 | COULD_ACCESS LPAR arg COMMA arg RPAR #CouldAccess
 | USES_METHOD LPAR arg COMMA arg COMMA arg RPAR #UsesMethod
 | TRANS_ACCESS LPAR arg COMMA arg RPAR #TransAccess
 | comp # ComparisonProp
 ;

search : search (UNION | INTER | SETMINUS) search # UnionInterMinus
| search_simple # SearchSimple
;

search_simple: VAR # Var
```

```
 | FROM search SELECT VAR WHERE proposition # Select
 | EDGES LPAR (arg | UNDERSCORE) COMMA (arg | UNDERSCORE) RPAR # Edges
 | ACCESS LPAR (arg | UNDERSCORE) RPAR #Access
 | VERTS #Vertices
 | SUCC LPAR arg COMMA arg RPAR #Succ
 | PRED LPAR arg COMMA arg RPAR #Pred
 | ACCESS_FROM LPAR search RPAR #AccessFrom
 | arg_array # ArgArray
 ;


exp : exp (PLUS | MINUS | MULTI | MOD | DIV) exp # ExpOp
 | LPAR exp RPAR # ParenthesizedExp
 | num # Number
 ;

num : INT # Int
 | LENGTH LPAR search RPAR # Length
 ;

comp : exp (EQ | NEQ | LT | GT | LEQ | GEQ) exp # Comparison
 | LPAR comp RPAR # ParenthesizedComparison
 | arg IN search # In
 | search (LT | GT | EQ | LEQ | GEQ | EQ | NEQ) search # SetComp
 ;
```

Appendix C

Evaluation and Case Studys

Example of command used to test tactic evaluation time:

```
LET U = FROM $V SELECT v WHERE (
FROM $E(_,v) SELECT l WHERE (
    LEN(PRED(l,v)) ==
    LEN (FROM PRED(l,v) SELECT u WHERE has_access('attacker', u)) + 1)
    != {});

FOR ONE v IN U TRY (
    FOR ONE l IN FROM $E(_, v) SELECT l2 WHERE (
        LEN(PRED(l2,v)) ==
        LEN (FROM PRED(l2,v) SELECT u WHERE has_access('attacker',u)) + 1)
    TRY(disc_access('attacker', FROM PRED(l,v) SELECT x WHERE
    !has_access('attacker',x)[0]);
        gain_access('attacker', v)))
```

C.1 Google Account Case Study

Commands to Generate Initial State:

```
(for all x in ['Google PWD', 'Outlook',
'Google PWD Manager', 'Google full', 'MS Password',
'Google open', 'Google Chrome', 'Laptop', 'Locked Laptop',
'Laptop PIN', 'Phone', 'Locked Phone', 'Phone PIN']
DO create_account('user', x));

add_access('user', ['Google PWD'], 'Google full', 'PWD');
add_access('user', ['Outlook'], 'Google full', 'recovery');
add_access('user', ['Google full'], 'Google open', 'subset');
add_access('user', ['MS Password', 'Locked Laptop']
, 'Laptop', 'PWD');
add_access('user', ['Laptop PIN', 'Locked Laptop']
, 'Laptop', 'PIN');
add_access('user', ['Google open']
```

```
, 'Google PWD Manager', 'subset');
add_access('user', ['Google PWD Manager']
, 'MS Password', 'contains');
add_access('user', ['Laptop'], 'Google Chrome', 'runs');
add_access('user', ['Google Chrome'], 'Google open', 'session');
add_access('user', ['MS Password'], 'Outlook', 'PWD');
add_access('user',['Phone'], 'Google Chrome', 'session');
add_access('user', ['Phone PIN'], 'Phone', 'PIN');
add_access('user', ['Locked Phone'],'Phone','PIN');

(for all x in [ 'Outlook',
'Google PWD Manager', 'Google full',
'Google open', 'Google Chrome'] DO lose_access('user', x))
```

C.1.1 Commands for modeling of Attack

Commands to Gain Initial Access:

```
(lose_access('user', 'Laptop');
lose_access('user', 'Locked Laptop');
disc_access('attacker', 'Laptop')) ||
(lose_access('user','Phone');disc_access('attacker', 'Phone'))
```

Commands to Copy Password and for User to Return:

```
(for all x in ['Google Chrome', 'Google open',
'Google PWD Manager', 'MS Password']
do gain_access('attacker', x));
lose_access('attacker', 'Laptop');
lose_access('attacker', 'Google Chrome');
lose_access('attacker', 'Google open');
disc_access('user','Locked Laptop');
gain_access('user', 'Laptop')
```

Commands for when Attacker Regains Access to Password Manager:

```
for all x in ['Outlook', 'Google full', 'Google open',
'Google PWD Manager'] do gain_access('attacker', x)
```

C.1.2 Intermediate Figures for Attack

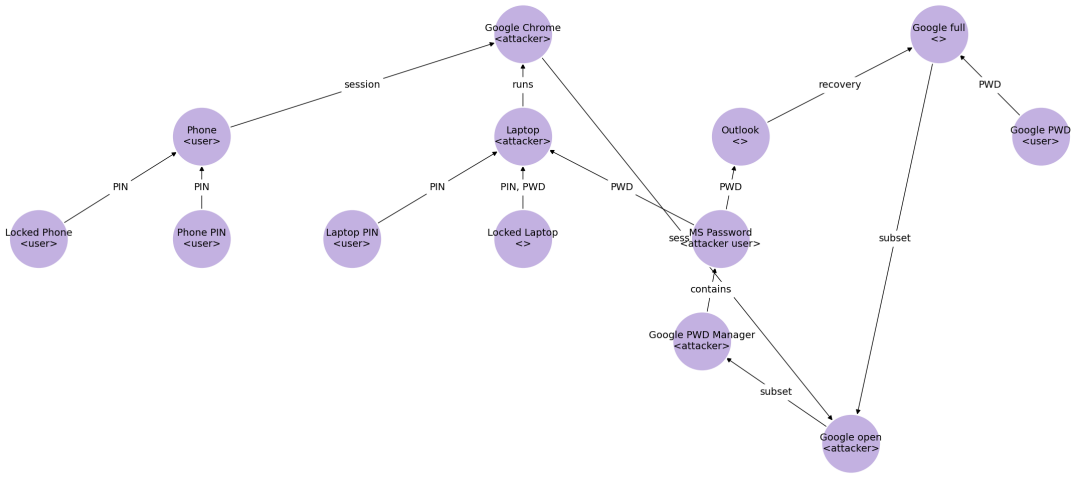


Figure C.1: Graph after the first attack stage.

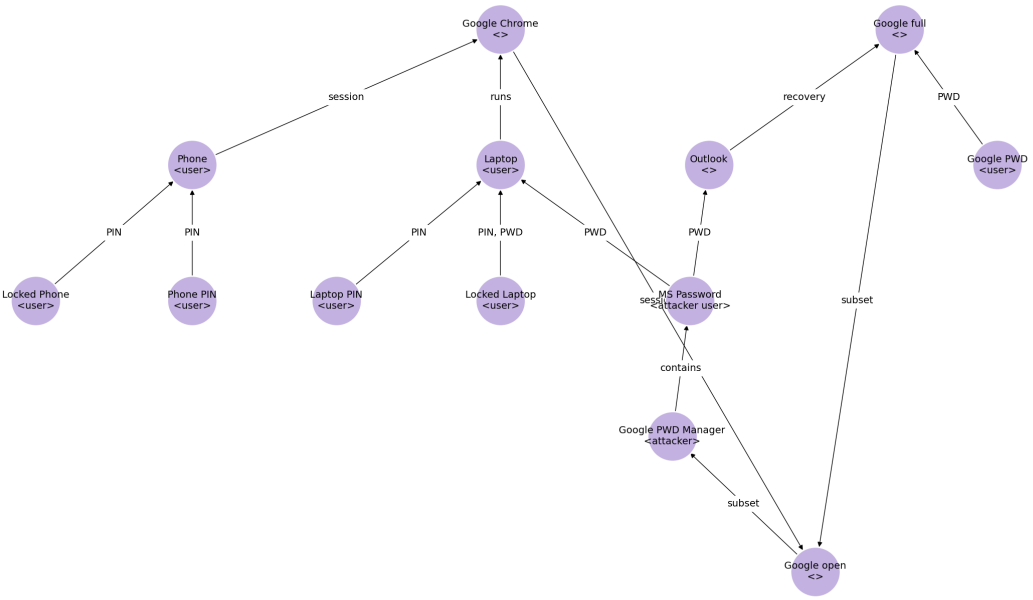


Figure C.2: Graph after the second attack stage.

C.1.3 Commands to Implement 2FA

Adding 2FA to the outlook account:

```
gain_access('user','Outlook');
create_account('user','2FA code');
add_access('user', {'2FA code'}, 'Outlook', 'PWD');
add_access('user', {'Phone'}, '2FA code', 'SMS');
lose_access('user', 'Outlook')
```

Attacker attempting to gain access to the recovery email following addition of 2FA:

```
gain_access('attacker', 'Outlook')
```

C.2 OAuth Case Study

Commands to Generate Initial State:

```
(for all x in ['Chrome Files', 'Encryption Key', 'Encrypted Token',
'GAIA ID', 'Decrypted Token', 'Session Cookie', 'Google Account',
'Google PWD', '2FA Code', 'YouTube', 'Cloud Storage', 'Gmail',
'MultiLogin']DO create_account('user', x));
add_access('user', {'MultiLogin', 'Decrypted Token', 'GAIA ID'},
'Session Cookie', 'GEN');
add_access('user', {'Encrypted Token', 'Encryption Key'},
'Decrypted Token', 'Decrypt');
add_access('user', {'Chrome Files'}, 'Encrypted Token','Exfiltrate');
add_access('user', {'Chrome Files'}, 'Encryption Key','Exfiltrate');
add_access('user', {'Chrome Files'}, 'GAIA ID','Exfiltrate');
add_access('user', {'Session Cookie'}, 'Google Account', 'Session');
add_access('user', {'Google PWD', '2FA Code'}, 'Google Account', 'PWD');
add_access('user', {'Google Account'}, 'YouTube', 'OAuth');
add_access('user', {'Google Account'}, 'Gmail', 'OAuth');
add_access('user', {'Google Account'}, 'Cloud Storage', 'OAuth');
disc_access('attacker', 'Chrome Files');
disc_access('attacker', 'MultiLogin')
```

C.2.1 Commands for modeling of Attack

Commands for Theft of Chrome Files and to Decrypt the Token:

```
(FOR ALL x IN {'Encryption Key', 'GAIA ID', 'Encrypted Token'}
DO gain_access('attacker', x));
gain_access('attacker', 'Decrypted Token')
```

Commands to Generate New Cookie then Access all Related accounts:

```
gain_access('attacker', 'Session Cookie');
gain_access('attacker', 'Google Account');
```

```
FOR ALL x IN SUCC('OAuth','Google Account') INTER
$AccessFrom({'Google Account'}) DO gain_access('attacker',x)
```

C.2.2 Commands for Account Recovery

Reset the tokens associated with the account.

```
lose_access('attacker', 'Chrome Files');
create_account('user', 'New Token');
rem_access_1('user', 'Session Cookie', 'GEN');
lose_access('attacker', 'Session Cookie');
add_access('user', {'MultiLogin', 'New Token',
'GAIA ID'}, 'Session Cookie', 'GEN')
```

Attacker attempts to regenerate cookie.

```
lose_access('attacker', 'Google Account');
gain_access('attacker', 'Session Cookie')
```

Changing the Google Account Password.

```
create_account('user', 'New PWD');
rem_access_1('user', 'Google Account', 'PWD');
add_access('user', {'New PWD', '2FA Code'}, 'Google Account', 'PWD')
```

C.3 Heuristic Evaluation of Usability

C.3.1 Improved UI

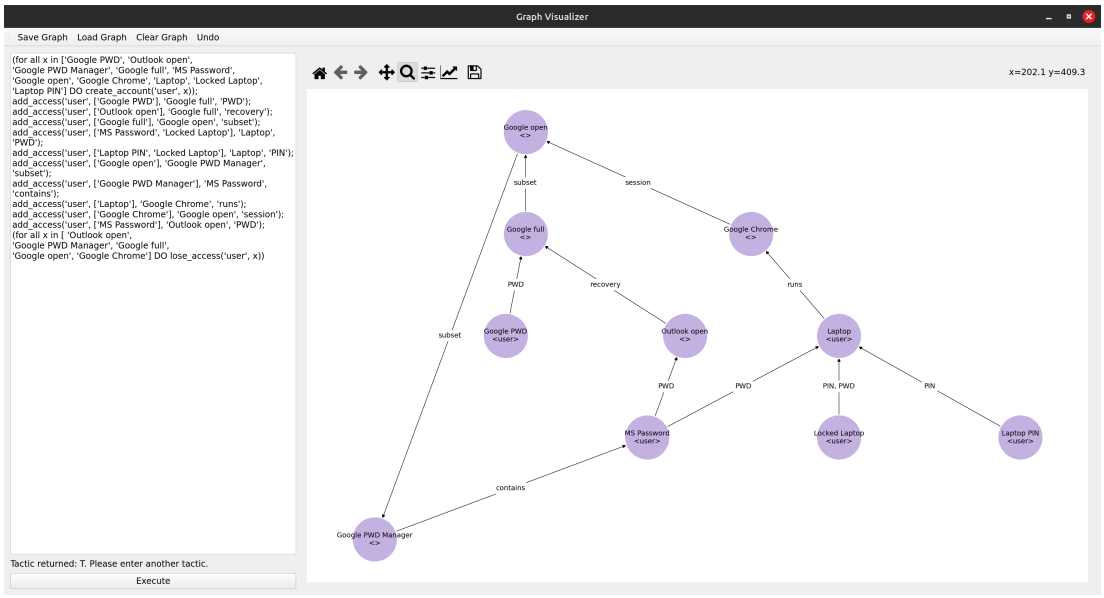


Figure C.3: A Screenshot of the updated interface following improvements made as a result of the evaluation.

Table C.1: Heuristic Evaluation of UI (PART 1)- Rule Descriptions as stated by Jakob Nielsen [26]

Rule	*Description*
Visibility of System Status	*The design should always keep users informed about what is going on, through appropriate feedback within a reasonable amount of time.*
Match between System and Real World	*The design should speak the users' language. Use words, phrases, and concepts familiar to the user, rather than internal jargon. Follow real-world conventions, making information appear in a natural and logical order.*
User Control and Freedom	*Users often perform actions by mistake. They need a clearly marked "emergency exit" to leave the unwanted action without having to go through an extended process.*
Consistency and Standards	*Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform and industry conventions.*
Error Prevention	*Good error messages are important, but the best designs carefully prevent problems from occurring in the first place. Either eliminate error-prone conditions, or check for them and present users with a confirmation option before they commit to the action.*
Recognition Rather than Recall	*Minimize the user's memory load by making elements, actions, and options visible. The user should not have to remember information from one part of the interface to another. Information required to use the design (e.g. field labels or menu items) should be visible or easily retrievable when needed.*
Flexibility and Efficiency of Use	*Shortcuts — hidden from novice users — may speed up the interaction for the expert user so that the design can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.*
Aesthetic and Minimalist Design	*Interfaces should not contain information that is irrelevant or rarely needed. Every extra unit of information in an interface competes with the relevant units of information and diminishes their relative visibility.*

Table C.2: Heuristic Evaluation of UI (CONT.)- Rule Descriptions as stated by Jakob Nielsen [26]

Rule	*Description*
Help Users Recognize, Diagnose, and Recover from Errors	*Error messages should be expressed in plain language (no error codes), precisely indicate the problem, and constructively suggest a solution.*
Help and Documentation	*It's best if the system doesn't need any additional explanation. However, it may be necessary to provide documentation to help users understand how to complete their tasks.*

Table C.3: Heuristic Evaluation of UI - Scores found using the framework

Rule	Score (out of 5)	Notes
Visibility of System Status	**4**	For large graphs, due to the long runtime of the positioning algorithm, it can take up to a minute before an updated graph state is displayed.
Match between System and Real World	**3**	The tactic commands, as implemented, are close to those within the abstract notation. Though there are some changes in how arguments are provided.
User Control and Freedom	**1**	There is no clear way to undo an action. The closest mechanism is for the user to preemptively save then reload the graph.
Consistency and Standards	**5**	Standard Language is used in Button labeling, and keywords are consistent with the formal tactics language.
Error Prevention	**2**	The use of a single line input for tactics makes it easy to make typographical errors. The removal of types makes it easier for incorrect tactics to be entered and executed.
Recognition Rather than Recall	**4**	Labels and relevant information is always visible to the user. Potentially help prompts could be added.
Flexibility and Efficiency of Use	**4**	Shorter versions of many keywords are supported within the implemented language. Though there is no support for macros.
Aesthetic and Minimalist Design	**3**	No excessive information is provided. However the UI is constructed of basic visual elements, thus is not especially aesthetic compared to other software.
Help Users Recognize, Diagnose, and Recover from Errors	**4**	Meaningful error messages are provided to the user when errors occur during tactic evaluation.
Help and Documentation	**2**	While there is no specific documentation (aside from this report), the interface is relatively intuitive. Details on the tactics laguage can also be found within the source paper.