Imperial College London

BEng Individual Project

Attribution of Cyber Attacks: An Argumentation-based Reasoner

Author:
Linna Wang

Supervisor:
Dr Erisa Karafili

June 17, 2018

Contents

Abstract

As our lives become increasingly entwined with technology and the Internet, the impact of cyber attacks will be increasingly damaging. *Know yourself, know your enemy, and you shall win a hundred battles without loss*, according to ancient Chinese military strategist Sun Tzu. Knowing the 'enemy' by determining the culprit behind an attack, also called the attribution of a cyber attack, is fundamental in building preventive measures against similar culprits.

Some challenges faced by analysts and investigators performing attribution include managing the huge amounts of data available; gathering and putting together evidences from different aspects of the attack; and dealing with conflicting evidences. While existing methods of attribution rely heavily on experienced human analysts and automated tools that focus on extraction or analysis of specific technical evidences; this project is an effort to automate the attribution process by using *argumentation* to reason with both *technical* and *social* evidences.

The result is an argumentation-based reasoner (*ABR*) which, given sufficient evidence, is able to perform the attribution of an attack. *ABR* also provides a suite of tools that refines the core reasoner. Firstly, *ABR* makes the derivations of its solutions fully *transparent*. Comprehensive graph visualisations are produced, supporting the result by explaining how the result is derived. Secondly, *ABR* emphasises on the *incremental* and *iterative* nature of attribution by introducing various features that encourage further action from the its users. Lastly, *ABR* is built to be *flexible*, where all the solutions are presented to users with numerical scores to serve as guidelines, and ultimately the users have control over what kind of reasoning or solutions they prefer.

Overall, *ABR* explores the possibility of expressing attribution using a formal argumentation-based framework while achieving performance level expected of a real-time interactive application. Being the first tool of its kind that performs attribution using both technical and social evidences combined with background knowledge that models the real world, *ABR* has the potential to redefine how attribution is done in the cyber world.

Acknowledgements

I would like to take the opportunity to thank my supervisor, Dr Erisa Karafili, for her continual support and encouragement throughout the project. I feel really fortunate to have such an enthusiastic supervisor who is willing to put so much time and effort into helping me, this project would have been impossible without her efforts. Additionally, I would like to express my gratitude to Prof. Antonis C. Kakas, who was extremely helpful and provided much-needed advice on the usage of the Gorgias framework.

I would like to thank my friend, Ming Hong, for proof-reading this report, and many others for their invaluable feedback. To all my friends and family, thank you so much for the friendship and support over the years. I will miss cooking and feasting after exams, and all the random heart-to-heart sessions we have. My university life has been so enjoyable and fruitful; I have learnt so much during these three years, I could not have asked for more.

Chapter 1

Introduction

In this chapter, we first introduce the motivations for the project, before detailing the objectives and contributions made.

1.1 Motivations

It is expected that by 2021, cyber crime damage costs will hit $6 trillion annually [1]. In 2017, we have seen some of the most egregious attacks, including WannaCry, NotPetya, Wikileaks CIA Vault 7, as well as the exposure of data of 198 million US voters [2]. Moreover, the use of the *Internet of things* (IoT) and other smart devices are projected to increase exponentially from 2015 to 2025 [3]. With the sheer amount of personal information we have online and the degree of dependence on these devices, individuals are also extremely vulnerable and exposed to cyber attacks. Therefore, it is clear that organizations, countries and individuals alike have an urgent need to bolster their defences against the myriad of cyber criminals online. Although the cyber security industry have been trying to keep up by introducing new and better technology and security techniques, we believe that such a one-size-fits-all approach is insufficient.

After an attack, if we can find out the profile, motivations and methods used by the attacker using *attribution*, we can then respond in a meaningful way to it or future attacks from similar attackers. This includes strengthening the security of the systems in a targeted way, specific to the profile of the attacker. For example, certain hacker groups may only operate in certain times of the year (due to religious or national holidays), so knowing the identity of the attacker, the defender organisation can organize its defence resources accordingly [4].

However, it is imperative to ensure that the attributions are correct, as false attributions can lead to dire consequences, especially if the accused attacker is a nation-state. When a nation-state have been identified to be supporting a cyber-operation, victim nation-states often carry out severe actions against its alleged attacker. For example, after the Sony hack in 2014, the United States imposed an economic sanction against North Korea, the alleged attacker behind this attack [5]. Meanwhile, many security experts in the private sector have been doubtful of FBI's attribution [6], due to lack of conclusive evidence published by the FBI. North Korea reiterated its denial of involvement, criticizing the United States for its "hostile and repressive" move [7]. If the attribution was in fact wrong, it would have aggravated the international relations between United States and North Korea, or even spark a war.

It is thus crucial to not only solve the problem of attribution, but to solve it with considerable accuracy or at least indicate any uncertainties in attributions made.

1.2 Objective

The main goal of this project is to construct an argumentation-based reasoner (*ABR*) to aid analysts in addressing the attribution problem and produce comprehensive explanations to support the attribution. By formalising the attribution process, using both technical and social evidences, we hope to automate parts of the attribution to reduce the reliance on the knowledge and experience of forensic analysts.

Given various pieces of evidence gathered, the objective of the tool is to return the following:

Results of attribution: present a list of entities (countries or hacker groups) that could be responsible for the cyber attack.

Explanation of the result: provide a coherent explanation for the results of attribution.

Rank multiple results: when there are more than one result for a given attack, be able to rank the results by their relative strength.

Highlight gaps in reasoning: highlight to the user any assumptions or potential gaps in the reasoning used during attribution.

Path for further investigation: when there are insufficient evidence to make an attribution, suggest other information that can be given in order to successfully perform the attribution of a cyber attack.

1.3 Contributions

At the end of this project, we have an argumentation-based reasoner (*ABR*) that is able to perform the attribution of an attack using both technical and social evidences, built using Prolog.

Reasoner built using a formal logic framework Using a preference-based argumentation framework, the reasoner consists of rules and preferences split into layers according to a model proposed by Rid and Buchanan (Chapter 4). This model enables us to represent both technical and social evidences and uses both to arrive at an answer to the attribution problem. The rules in the reasoner are constructed by analysing attributions of well-known cases of cyber attacks and extracting the reasoning process used by investigators and analysts who worked on the cases.

User interface *ABR* extracts the essential information from the result returned by the reasoner and present them to the user in a much more coherent manner (than the default Prolog output) in its graphical user interface (GUI) (Chapter 6). The GUI offers many ways to interact with *ABR*, where the user can execute in *standard* or *verbose* mode, insert new rules and evidences, or execute their own custom query.

Visualisation Surrounding the core reasoner, we have built a suite of other tools and features using Java (Chapters 5 and 7). Structures that are hierarchical in nature (derivation and argument tree) are visualised by *ABR* into colour-coded graphs, which can be viewed from *ABR*'s GUI (Section 5.3). The derivation graph serves as a visual explanation of how the solution is derived; while the argument tree shows how the solution defends itself against counter arguments.

Scores To provide a guideline to users in deciding which result is stronger when faced with multiple results from *ABR*, a numerical score is assigned to each attribution result. The scoring system is constructed by analysing how humans might attempt to score a generic argument.

Forensic tool integration *ABR* integrates with some forensic tools to automate extraction of some evidence, fitting itself into the normal work-flow of analysts.

Evaluation Finally, we evaluate the performance of *ABR* by testing its correctness and performance by using both synthesized test scenarios and real cases of cyber attacks (Chapter 8). Also considered is how *ABR* scales as the number of evidences provided increases.

The result of this project is an interactive tool that, given sufficient evidence, is capable of returning the attribution result and provides comprehensive diagrams that explains its results.

1.4 Structure of report

We present the related work and background research that was used as a starting point for our project in Chapter 2. A high-level overview of *ABR* is given in Chapter 3, where we briefly describe its architecture and its various components. The main core of the *ABR* reasoner is presented in Chapter 4, where we introduce the reasoning rules and background facts used. *ABR* is composed of other key components, such as the scoring system, forensic tool integration, visualisation of key structures, as described in Chapter 5, and its user interface, introduced in Chapter 6. Details of the implementations can be found in Chapter 7. We evaluated *ABR*, showing the results and details of the test cases in Chapter 8. Finally, we conclude with our final remarks on the project and some interesting future works in Chapter 9.

As non-essential reading, we include in Appendix A further explanations of the core Gorgias rules used in *ABR*, which are extracted from the cyber attack cases as shown in Appendix B. Further details regarding some problems we faced in our project and other implementation details can be found in Appendices C and D respectively.

Chapter 2

Background

In this chapter, we consider a model proposed by Rid and Buchanan to illustrate attribution, existing means to perform attribution and their limitations. Next, we discuss the usefulness of argumentation and abduction in solving some of these problems. Finally, we introduce some key concepts that are crucial for understanding the rest of the report.

2.1 Attribution

Attribution of a cyber attack is "determining the identity or location of an attacker or an attacker's intermediary" [8]. Due to how the Internet was constructed, tracing the origin of a cyber attack can be really difficult since attackers can easily forge or obscure source information [9].

A model that deals with the problem of attribution is proposed in [10]. This model, proposed by Rid and Buchanan, is called the *Q-model* and it divides the attribution process into three distinct layers: *technical, operational* and *strategic*. It describes attribution as an *incremental process*, each of the layers build on top of each other to form the final attribution. We introduce the three layers of the Q-model below.

Technical The *technical* layer comprises the bulk of the work in digital forensics. It includes scanning and reporting of abnormal computer behaviour. The objective of this layer is to find out *how* the attackers compromised the system. Examples of evidence that belongs to this layer includes the presence of zero-day vulnerabilities, signatures or unique traits in malware, as well as traits like language indicators.

Operational The *operational* layer combines information from various sources, including derived information from the technical layer, non-technical (social) evidences, and the geopolitical context. The objective of this layer is to find out *what* attacked the systems, to develop a high-level understanding of the attacker. Such details include the motivations of the attacker and their general profile.

Strategic The *strategic* layer uses information from the technical and operational layer in attempt to draw a conclusion. To unveil the attacker's intent, analysts can consider precedents or previous attacks. Attempts to find similarities with previous cases might help to draw links between cases with similar perpetrators.

2.1.1 Case studies of attribution

To explore how analysts use technical, social evidence, and contextual knowledge for attributing attacks, we present some well-known cases of cyber attacks in this section. More case studies can be found in Appendix B.

Stuxnet Stuxnet is one of the world's first large-scale cyber attacks, even dubbed *the World's First Digital Weapon* [11]. Stuxnet was first discovered in 2010 at the uranium enrichment plant in central Iran. Its code was found to be especially complex and sophisticated, using four zero-day vulnerabilities, which was the first clue to investigators that a larger organization was behind this attack, since zero-day vulnerabilities are rare in the cyber world. A multitude of other technical evidence including the presence of two fraudulent certificates has also pointed to the direction that a large organization with substantial resources was responsible. The large proportion of infected machines that were in Iran sparked suspicions that the attack could have been a state-sponsored cyber attack [12]. Reports that the Iranian President Mahmoud Ahmadinejad was using the nuclear program to build a nuclear weapon serves as a motivation for other nation-states to attack the power plants as an act of sabotage to Iran's uranium enrichment program. The United States and Israel has been allegedly responsible for the attack.

APT1 espionage campaign Since 2004, cyber security breaches at hundreds of organizations globally were attributed to advanced threat actors commonly called the *Advanced Persistent Threat*(APT) groups. APT1 is the most prolific of those groups, and is believed to be the 2nd Bureau of the People's Liberation Army General Staff Department's 3rd Department, commonly known as Unit 61398 [13]. APT1 has stolen hundreds of tera-bytes of data from more than a hundred organizations, demonstrating "the capability and intent to steal from dozens of organizations simultaneously" [13]. This exhibits an abundance of resources of the perpetrator of this attack. Furthermore, the industries were found to "include four of the seven strategic emerging industries that China identified in its 12th Five Year Plan" [13], which serves as the motive for China to carry out the attack. On top of that, technical evidences point to Chinese involvement, since 98% of the IP addresses used by APT1 was registered in China, Shanghai. The client systems were also found be set to use Simplified Chinese language. Moreover, one of the tools used in the attack, *HUC Packet Transmit Tool*, was also found to be registered in China.

Sony hack In 2014, hackers infiltrated the Sony's computers and stole data from Sony servers. The Sony hack is an example of a case where attribution still remains controversial, years after the incident. A group called *Guardians of Peace* claimed credit for the attack, but the media and several government agencies claimed that the attack is state-sponsored by North Korea. This attribution was based on the fact that North Korea has a strong motive to attack Sony, in retaliation for Sony's then yet-to-be-released film, *The Interview*, a comedy involving a CIA plot to kill North Korean leader Kim Jong-un [14]. There were also technical evidences indicating North Korean involvement in the hack. The attackers used proxy servers to hide their origin during the attack and when sending public statements, but according to FBI Director James Comey, several times they got "sloppy", and left their tracks unhidden [15]. The FBI then traced the IP addresses to find that they were "exclusively used by the North Koreans". The FBI has also found that code in the malware used by *Guardians of Peace* in the Sony attack is similar to code used by North Korea in other attacks. However, many security experts disagree on that attribution. They commented that there is too little, and conflicting evidence to attribute the attack to North Korea; and the malware was leaked long before the hack and "any hacker anywhere in the world could have used it" [16]. Against these comments, FBI claims that they possess evidences and resources that were not made available to public.

2.2 Existing approaches to attribution

Currently, attribution involves a myriad of different approaches and facets. In the following section, we consider some of these existing approaches to attribution and the problems associated with them.

2.2.1 Forensics

Digital forensics, or sometimes called Computer forensics, is defined by the United States Computer Emergency Readiness Team (US-CERT) to be "the discipline that combines elements of law and computer science to collect and analyse data from computer systems, networks, wireless communications, and storage devices in a way that is admissible as evidence in a court of law" [17]. The main stages of digital forensic processes [18] are:

1. **Collection:** obtaining data from various data sources, taking care to isolate the device to prevent accident infection of more machines.

2. **Data examination:** evaluate and extract pertinent information from the data.

3. **Information analysis:** correlate multiple information sources to reach a conclusion regarding the people, location, tool, and events involved in the incident.

4. **Evidence reporting:** present evidence resulting from analysis, considering factors such as alternative explanations and audience.

Carrier defines the *Complexity Problem* and the *Quantity Problem* in digital forensics. The *Complexity Problem* is that "acquired data are typically at the lowest and most raw format" [19], which requires large amounts of time for highly-skilled human analysts to decode and understand. The *Quantity Problem* is caused by the sheer amount of data to analyse, that can be way too large for a human analyst to manually analyse every piece of data. To resolve these two problems, data translation tools and data reduction techniques are used respectively. Carrier also organised digital forensics into different layers of abstraction [19]. We describe what these layers mean and list some of the popular tools used in these layers below:

- **Physical media analysis:** analysis of hard disk, memory chips and recovering deleted data after it has been overwritten, e.g., commercial toolkits like FTK Toolkit.

- **File system analysis:** viewing file and directory contents, as well as recovering deleted files, e.g., Disk Drill[1].

- **Application analysis:** viewing configuration files, log files, images, documents and reverse engineering executables. In Linux there are several tools like *GNU Binutils*, *nm*, *objdump* that can be used to decompile the malware. There are also decompiler software available on other operating systems, e.g., REC Decompiler[2].

- **Network analysis:** analysis of network packets, e.g., Wireshark[3]; and intrusion detection system (IDS) alerts, e.g., Snort[4] and OSSEC[5].

- **Memory analysis:** identification of the code that a process was running on, e.g., VirusTotal[6].

Below we present two security techniques that can be used for collecting forensic evidences.

Traceback Techniques

Traceback techniques are used in Network forensics to retrace the origin of the attack. Some examples of such techniques, like storing logs and traceback queries, performing input debugging, inserting host monitor functions, are presented by Wheeler in [8].

Honeypots

Honeypots are components that "provides its value by being attacked by an adversary", and have been used in cyber security since the 1990s [20]. Honeypots are useful for detecting and recording malicious activity. Anagnostakis et al. proposed a honeypot-based detection architecture [21], where the honeypots and the rest of the system are combined, and suspicious or anomalous activity are routed to the honeypot system for further examination. The advantage of using honeypots over normal operational systems is that after detection, a honeypot can be easily disconnected from the rest of the system and isolated for further investigations since it does not actually carry out any useful function as part of the system. Forensic analysts can then retain and investigate using the *attacked* state of the honeypot.

Limitations of digital forensics

One of the main limitations of digital forensics lies in the difficulty of differentiating real evidence from forged evidence placed by the attacker intentionally. For example, focusing on network forensics specifically, source IP of the attacker can be easily forged or obscured. There are many different technologies that a hacker can use to obscure their IP, such as virtual private networks (VPNs), proxies or Tor.

[1]Disk drill: https://www.cleverfiles.com/
[2]REC Decompiler: http://www.backerstreet.com/rec/rec.htm
[3]Wireshark: https://www.wireshark.org/
[4]Snort: https://www.snort.org/
[5]OSSEC: https://www.ossec.net/
[6]Virustotal: https://www.virustotal.com/

Although there exists some spoof prevention techniques that verifies the validity of the connection back to the sender, this can only be used during the attack to prevent unauthorized connections, not in a post-attack investigation.

Furthermore, while forensic techniques in general are good for producing individual pieces of evidence from data that are agreeable with each other, it lacks the ability to deal with conflicting data or conflicting pieces of evidences from different sources. Since the attacker can easily plant false evidence to lead investigators off their trail, investigators are very likely to be in a situation with multiple pieces of conflicting evidences.

Digital forensics is also often extremely human-intensive, requiring many skilled analysts to work for weeks or even months. Nassif et al. [22] mentions that computer forensic analysis usually involves examining hundreds of thousands of files per computer, which makes it impossible for a human expert to manually analyse and interpret all the data. The problem is exacerbated by the fact that a large proportion of the data consists of unstructured text, which makes analysis by computers challenging too.

Digital forensics only uses technical evidence, but fail to consider other kinds of evidences such as geopolitical situations and social-cultural intelligence, which could also provide useful leads during investigations.

2.2.2 Cyber deception

Cyber deception techniques can be used not only for protecting the system from attackers, but also as part of the attribution process. Almeshekah et al. [20] introduced four different groups of protection mechanism, one of which includes *attribution and deception*. Deception-based mechanisms pivots on manipulating the attacker's perceptions, they are "planned actions taken to mislead attackers and to thereby cause them to take (or not take) specific actions that aid computer-security defences" [23]. Examples of such mechanisms include booby-trapped software, layered authentication, endless files, etc.

Such deceptive mechanisms can be used to entice attackers to revealing their identity and objectives. Since simply detecting the attacker's IP address is insufficient as IP addresses can be easily spoofed, Cyber deception can be used to trick attackers to allow the system to collect information on fixed artefacts such as tools, behaviours or traits.

Limitations of cyber deception

In order for deception to be successful, constant monitoring and feedback needs to be in place. Defenders need to observe the attacker's behaviour and decide whether to tone down the level of deception by including some true information, or even stopping the deception entirely. Since attackers frequently turn aggressive when they find out that they are deceived, the situation in which the attacker discovers that they were tricked should be avoided.

2.2.3 Socio-cultural modelling

Technical solutions such as adaptive defence strategies and offensive cyber-operations have some obvious shortcomings. Technical developments restricts the effectiveness of adaptive defence strategies, and the cyber-operations against the attacker might spark a cyber-war. Hence, there has been a move towards *threat intelligence*, which involves using machine learning techniques to collect and analyse digital communities such as hacker forums and chat-rooms [24].

Problems with Social-cultural Modelling

There are two major challenges in this approach. Firstly, in order to create the model, we need to identify the various persona across different digital communities, which is very challenging since digital communities are usually anonymous. Furthermore, deceptive actions by the true attacker, like constructing a false virtual persona, can deceive investigators to make a false attribution to a non-existent entity or even make another person or organization the scapegoat.

2.3 Argumentation

Two main challenges of attribution are targeted in this project are (i) incomplete information and (ii) the existence of conflicting, possibly deceptive information. To deal with these challenges, we explore the use of argumentation in the attribution process. In this section, we first introduce argumentation, then present some applications of argumentations and its strengths. Next, we look at the related studies on the use of argumentation in attribution and limitations of these studies. Lastly, we justify why argumentation should be used for attribution, and look at a few of the argumentation tools available.

Argumentation is "the act or process of forming reasons and of drawing conclusions and applying them to a case in discussion" [25]. Formal argumentation involves a set of chained arguments. Each argument has *premises* and *conclusions*.

Premise: a set of statements that supports the argument. The premise of an argument needs to be true in order to prove the conclusion.

Conclusion: a statement that the arguer is trying to convince the listener of.

Arguments can be *linked* or *convergent*. In linked arguments, the premises must *all* be true in order for the conclusion to follow (logical AND). In convergent arguments, *any one* of the premises itself is sufficient to prove the conclusion (logical OR), and more premises simply reinforce the conclusion [26]. When there are more than one argument, there are a few ways that one argument can attack another argument. Argument Y can:

1. Attack the premise of argument X;
2. Ask critical questions to doubt the acceptability of the argument;
3. Raise a counter-argument. A counter-argument is one that leads to the opposite conclusion (e.g., if an argument has conclusion 'CountryX is the culprit for the attack', then its counter-argument is 'CountryX is *not* the culprit for the attack').

In the case of a persuasion dialogue[7], argumentation can ultimately uncover the strongest arguments on both sides of the argument.

2.3.1 Applications of argumentation

Argumentation has been used in many real-life applications. In this section, we introduce some of them to demonstrate the advantages of argumentation.

Health industry Argumentation is used in risk assessment and communication in health care [27]. The Online Patient Education and Risk Assessment (OPERA) is based on deductive argumentation by association. By using "facts" and "truths" as premises, and risk explanations as arguments, OPERA presents its claims with an "*ex auctoritate* causal link" [28] (explaining the cause-and-effect relations in an authoritative manner). Such an approach is aimed at giving patients a personal, comprehensive explanation of the assessment while maintaining some of its authoritative nature. In OPERA, argumentation is used for the transparency that it provides to the patients. This enables the patients to understand their situation that motivates the various decision made by the medical staff, and thus be able to accept those decisions more easily.

Design industry Another example use case is an analysis of interaction between designers within a team. Stumpf et al. conducted a study [29] to understand how designers interact with each other and model the design discourse, ultimately suggesting useful techniques for better design. In this case, argumentation is used, since designers naturally tend to fall into persuasive arguments, to "explain, predict, justify and warrant their artefacts" to each other.

Transport industry Argumentation is also used in the development of an integrated flexible transport systems platform in rural areas in [30]. Velaga et al. used a resource-bounded argumentation framework, which is a traditional framework extended with a set of resource bounds. This is to model the fact that arguments (passenger trying to acquire a ticket) would only acquire resources (a seat on the train) once they are deemed acceptable. In this case, the negotiation dialogue in argumentation is used to evaluate the "conflicting choices available to both passengers and service providers" [30]. Once again, the ability to visualise easily the justifications for its decisions, for debugging purposes, was one of the main reasons why argumentation was chosen.

2.3.2 Advantages of argumentation

After looking at the examples of application of argumentation in the previous section, we review some of the key advantages of using argumentation.

Accessibility/Transparency Making decisions using argumentation is similar to how humans naturally make decisions, so the rationale for how the decision is made

[7]A persuasion dialogue is an exchange between two individuals where initial opinions are conflicting, and its goal is for one party to convince the other

(preferences between rules) can be easily accepted by humans via diagrammatic forms, achieving transparency. Furthermore, argumentation encourages evaluation of the argument, assessing relative importance of various factors when making decisions [27].

Incomplete and conflicting information Argumentation captures the fact that the final decision might change if more information is available (i.e., *non-monotonic reasoning*[8]). More information might reveal counter-arguments that could be stronger than the original winning argument [31]. In the case of conflicting information, different forms of argumentation deals with them differently. In preference-based argumentation, *preference rules* can be inserted into the knowledge base, where one rule is preferred to another when the premises of both can be satisfied.

2.3.3 Related studies on the use of argumentation in attribution

In this section, we consider some of these related works on the use of argumentation in attribution. Argumentation has been used for solving the attribution problem in [32, 33], where the work in [32] is an extension of [33].

DePL There has been research and experiments of how argumentation-based frameworks can be leveraged to improve cyber-attribution decisions. Nunes et al. show how an argumentation-based framework called *DeLP* (Defeasible Logic Programming) can be constructed [32]. DeLP is constructed using facts, strict rules and defeasible rules. Using ground truth derived from a DEFCON CTF dataset, the objective of the framework is to identify the correct team that carried out the attack.

InCA Shakarian et al. have also introduced an argumentation-based framework called the *Intelligent Cyber Attribution* (InCA) designed to aid the attribution process [33]. InCA combines argumentation-based reasoning, logic programming, and probabilistic models to attribute an operation and also to provide explanations to how the system arrives at its conclusion. InCA is constructed using two separate models, a probabilistic *environment model* (EM), and an argumentative *analytical model* (AM). The EM is used for reasoning about context information about the world, and must be in a consistent state. On the other hand, the AM is used for resolving conflicting information derived using context information from the EM. The InCA framework leverage on argumentation in the construction of the AM due to two main features of argumentation. Firstly, using argumentation, they were able to represent and resolve conflicting information in a formal, logical procedure. Secondly, the "transparency provided by the system" [33] provides the analyst with the insight required to identify incorrect inputs, calibrate the model, or collect more evidence.

[8]Non-monotonic reasoning represent *defeasible* inferences, allowing one to draw tentative conclusions that might be retracted given further information.

Limitations Despite these advances in using argumentation in attribution, there are still some shortcomings yet to be resolved. Firstly, none of the current works target the social aspect of attribution. Contextual knowledge such as ongoing conflicts between countries or rivalry between corporations can be very useful in detecting motives of potential culprits. Secondly, this project will be the first one to attempt to use the Q-model [10] to categorise evidence and rules in an argumentation-based framework, which could lead to more accurate attribution.

2.3.4 Why argumentation for attribution

We summarise below some of the key features of argumentation that makes it suitable for use in attribution.

Conflicting information Argumentation handles conflicting information elegantly, building up various arguments from the available evidences and returning the strongest ones. In preference-based argumentation, conflicting information are managed by preferences, where preferences between rules (or facts) can be added to determine the strength of the argument.

Transparency As mentioned previously, the transparency, or ability to provide self-explanatory results, is one of the key advantages of argumentation. Transparency is desirable in attribution as it provides a way to visualise the attribution process to even non-professionals. This makes the attribution more *accountable*, since the reasoning behind it can be clearly explained and understood. As analysts can understand the reasoning process taken by the argumentation-based reasoner, they are not confined to indiscriminately accepting the given solution, they are able to comprehend the reasoning process and decide if the solution is appropriate.

Formal and rigorous system Differing from the existing human-driven attribution process, using argumentation, we are able to semi-automate the process. Instead of fully relying on the experiences and instincts of human analysts, we construct a formal, rigorous proof system that aids the process of attribution.

2.3.5 Analysis of argumentation tools

In this section, we compare existing argumentation tools to determine which is the most suitable for this project. In this section, three tools are compared: Gorgias, GorgiasB and CaSAPI.

Gorgias

Gorgias[9] is a general argumentation framework that combines the preference-reasoning and abduction. The syntax for using Gorgias in SICStus Prolog are as follows:

Query `prove(Goals, Delta).`

Rule `rule(Label, Head, Body).`

[9]Gorgias: http://www.cs.ucy.ac.cy/~nkd/gorgias/

Preference `prefer(Label1, Label2).`

Conflict `conflict(Label1, Label2).`

Abducible `abducible(abduciblePredicate(_), [ ]).`

Fact `rule(Label, Fact, [ ]).`

Using Gorgias, we can create a prolog file as presented below:

```
:- compile('/PATH_TO_GORGIAS/gorgias-src-0.6d/lib/gorgias.pl').
:- compile('/PATH_TO_GORGIAS/gorgias-src-0.6d/ext/lpwnf.pl').

% Rules
rule(notGuiltyByDefault(X), (neg(isCulprit(X)), []).
rule(ipGeolocation(X), isCulprit(X), [ipGeoloc(X, IP)]).
rule(spoofedIp(X), neg(isCulprit(X)), [ipGeoloc(X, IP), spoofedIP(IP)]).

% Facts
rule(fact1, ipGeoloc(china, ip1), []).
rule(fact2, ipGeoloc(us, ip2), []).
rule(fact3, spoofedIP(ip1), []).

%Priority/Preference
rule(p1(X), prefer(spoofedIp(X), ipGeolocation(X)), []).
rule(p2(X), prefer(ipGeolocation(X), notGuiltyByDefault(X)), []).
```

To test or execute the model, we can use the enter the query `prove([neg(isCulprit(X))], D)` and `prove([(isCulprit(X))], D)` in the Prolog environment to get the following output in Listing 1.

Listing 1 Output from Gorgias

```
| ?- prove([neg(isCulprit(X))], D).
X = us,
D = [fact2,ipGeolocation(us)] ? ;
no
| ?- prove([(isCulprit(X))], D).
X = china,
D = [fact1,fact3,fact1,spoofedIp(china)] ? ;
no
```

GorgiasB

GorgiasB[10] is an extension of Gorgias, where the main difference between GorgiasB and Gorgias is the presence of a graphical user interface (GUI) in GorgiasB. To build the initial decision model, we need to perform the following steps:

[10]GorgiasB: `http://gorgiasb.tuc.gr/index.html`

1. **Add options:** options are the final decision to be made, e.g., `isCulprit(CountryA)`, `not(isCulprit(CountryA)`.

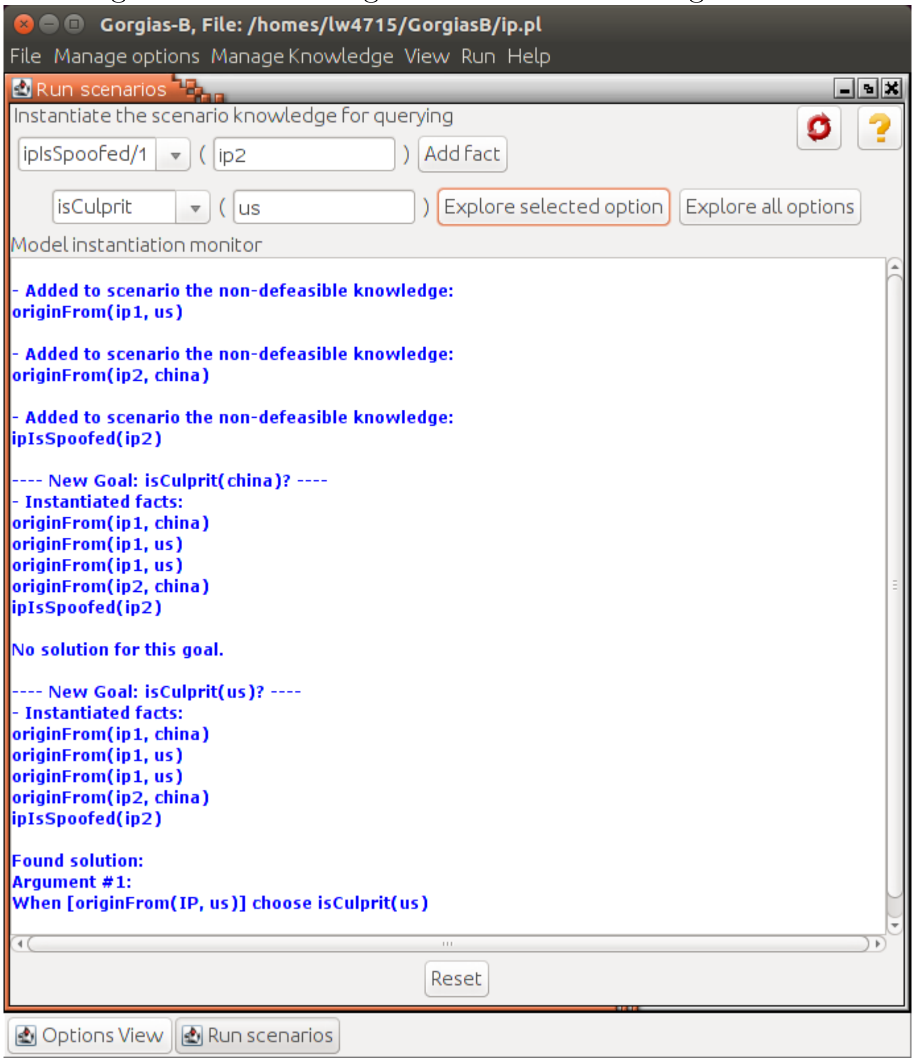
2. **Add beliefs:** beliefs are relevant knowledge that affect the decision being made, e.g., `originFrom(IP, CountryA)`.

3. **Define initial arguments:** arguments are general rules, for example `When[originFrom(IP, CountryA)] choose isCulprit(CountryA)`.

4. **Iteratively define sequences of more specific scenarios:** we increase the level of argument, by specifying scenarios where one option is stronger than the other. For example, `When[originFrom(IP, CountryA), ipIsSpoofed(IP)] choose not(isCulprit(CountryA))` is a level 2 argument.

5. **Execute:** finally, we execute the model to test it.

Figure 2.1: Executing decision model in GorgiasB GUI


Gorgias-B, File: /homes/lw4715/GorgiasB/ip.pl
File Manage options Manage Knowledge View Run Help
Run scenarios
Instantiate the scenario knowledge for querying
ipIsSpoofed/1
ip2
Add Fact
isCulprit
us
Explore selected option
Explore all options
Model instantiation monitor
Reset
Options View
Run scenarios

```
- Added to scenario the non-defeasible knowledge:
originFrom(ip1, us)

- Added to scenario the non-defeasible knowledge:
originFrom(ip2, china)

- Added to scenario the non-defeasible knowledge:
ipIsSpoofed(ip2)

---- New Goal: isCulprit(china)? ----
- Instantiated facts:
originFrom(ip1, china)
originFrom(ip1, us)
originFrom(ip1, us)
originFrom(ip2, china)
ipIsSpoofed(ip2)

No solution for this goal.

---- New Goal: isCulprit(us)? ----
- Instantiated facts:
originFrom(ip1, china)
originFrom(ip1, us)
originFrom(ip1, us)
originFrom(ip2, china)
ipIsSpoofed(ip2)

Found solution:
Argument #1:
When [originFrom(IP, us)] choose isCulprit(us)
```

Although the output from GorgiasB (Figure 2.1) is more comprehensible compared to the output from Gorgias (Listing 1), the use of GUI is a fundamental disadvantage of GorgiasB. Since we plan to build our entire project using the argumentation tool, using the GUI will be too slow, difficult to maintain, and hard to keep track of different versions.

CaSAPI

Credulous and Sceptical Argumentation, Prolog Implementation (CaSAPI)[11] is a hybrid argumentation system implemented in Prolog that combines abstract and assumption-based argumentation. In CaSAPI, the user starts the argumentation process with the command `run/3` that takes 3 arguments: derivation type (GB, AB, or ID), output mode (noisy or silent), and number of solutions. The three derivation types differ in the level of scepticism of the proponent agent [34]:

Grounded beliefs (GB) derivations Agent is not prepared to take any chances and is completely sceptical in the presence of seemingly equivalent alternatives.

Ideal beliefs (IB) derivations Agent is wary of alternatives, but is prepared to accept common ground between them.

Admissible beliefs (AB) derivations Agent will adopt any alternative that is capable of counter attacking all attacks without attacking itself.

One advantage of CaSAPI is that it can output the tuples $< P_i, O_i, A_i, C_i >$ at each step i in the argumentation, where P and O are the set of sentences held by the proponent and opponent respectively, A is the set of assumptions generated by the proponent and O is the set of assumptions in attacks generated by the opponent. These information can be useful for debugging purposes, to check that our model is performing as intended.

However, unlike GorgiasB and Gorgias, CaSAPI does not explicitly differentiate between normal rules and preference rules, which might make it less readable and thus confusing during construction of the model. For example, given the below CaSAPI rule from [35], it is considerably hard for a human to read and understand what the rule actually means.

```
myRule(def(t(villa,villa)),[not(def(t((villa,villa),(villa,villa)))),
        not(def(r3(villa,villa)))]).
```

Choice of argumentation tool

We decided to use Gorgias for this project. The advantages of using Gorgias can be summarised as follows:

Scalable Compared to GUI-based tools such as GorgiasB, Gorgias allows us to build our own interface that allow for more automation and uploading large number of rules instead of being restricted to inputting one rule at a time manually.

Flexible Since Gorgias is implemented as a library to Prolog, it can be embedded in our own application using another language[12].

[11]CaSAPI: https://www.doc.ic.ac.uk/~ft/CaSAPI/

[12]http://www.swi-prolog.org/pldoc/man?section=embedded

Comprehensive Gorgias labels each rule, which makes the program more readable and maintainable compared to other tools like CaSAPI.

2.4 Abduction

In this section, we describe how and why abduction is usually used, followed by advantages of using abduction in attribution.

Abduction is a form of 'logical inference'. Given an observation, it seeks to find the 'simplest and most likely explanation' for it [36]. This is done by *abducing* all possible explanations and considering competing hypotheses. When something is abduced, it can be assumed to be true. Using an example from [37], if we know that a car is not working, and we know that there are two possible causes for this: (i) the battery is flat ($flatBattery(car)$) or (ii) the car is out of fuel ($outOfFuel(car)$). Then in this case we say that the abducibles (Ab) are the two possible explanations for the observation:

$$Ab = \{flatBattery(X), outOfFuel(X)\}$$

All abducible predicates comes with a set of *integrity constraints*. Integrity constraints are rules that ensures the consistency of the abduction system. They are often written in the form of denials (rules with empty head). In order for the denial to be true, there must be at least one body predicate that is false. All abducibles have the following integrity constraint:

$$\leftarrow abduciblePred, \neg abduciblePred$$

This constraint ensures that the abducible predicate cannot be both true and false at the same time. While this must be true in order for the system to be consistent, integrity constraints can also contain other contextual rules that specified by the system's designer. For example, back to our previous example, a possible context-based integrity constraint can be:

$$\leftarrow flatBattery(X), lightsOn(X)$$

This integrity constraint represents the knowledge that if the lights of a car are on, we know that the car battery is not flat.

Next, to perform abduction, we try to prove the observation (O) using a set of background rules (B) by abducing any of the abducible predicates (Ab), while abiding by the integrity constraints (IC). Formalising the previous example, we have:

$$\begin{aligned}
O &= notWorking(car) \\
B &= \{notWorking(car) \leftarrow flatBattery(X). \\
&\quad\;\; notWorking(car) \leftarrow outOfFuel(X).\} \\
Ab &= \{flatBattery(X), outOfFuel(X)\} \\
IC &= \{\leftarrow flatBattery(X), lightsOn(X). \\
&\quad\;\; \leftarrow flatBattery(X), \neg flatBattery(X). \\
&\quad\;\; \leftarrow outOfFuel(X), \neg outOfFuel(X).\}
\end{aligned}$$

The abduction task is to find a set of abduced predicates ($\triangle$) that satisfies the following:

$$\begin{aligned}
\triangle \cup B &\vDash O \\
\triangle \cup B &\vDash IC \\
\triangle \cup B &\nvDash \bot
\end{aligned}$$

For this example, the solution is:

$$\begin{aligned}
\triangle_1 &= \{flatBattery(car), outOfFuel(car)\} \\
\triangle_2 &= \{outOfFuel(car)\} \\
\triangle_3 &= \{flatBattery(car)\}
\end{aligned}$$

2.4.1 Advantages of abduction

In the context of attribution, we use abduction to fill in the *knowledge gaps* in our reasoning. When we have insufficient evidence, we would like to see if we can make some assumptions so that we are able to make the attribution. By abducing some unavailable knowledge, we are able to make plausible deductions to reason about who might be the culprit of an attack. Adding this abductive step into our project means that it will be potentially fallacious, or reach a conclusion based on a false belief. However, in the real world, it is extremely unlikely that we are able to obtain all the required evidences and have no gaps in our knowledge before making the attribution for any case. Thus, the ability to work with incomplete information makes abduction suitable to use in the attribution process.

2.5 Foundation knowledge

In this section, we introduce the argumentation framework used by *ABR*, then give a brief overview of the syntax of Prolog and Gorgias, which will be helpful to understand the explanations on how *ABR* works during later discussions in coming sections.

2.5.1 Argumentation framework

In this section, we present the argumentation framework that we use in *ABR*, which is adapted from [38].

Argumentation theory defines the axioms in our framework. It is pair of *argument rules* and *preference rules*, $(\mathcal{T}, \mathcal{P})$.

Argument rules $\mathcal{T}$ is a set of labelled clauses in the form $rule_i : \mathcal{L} \leftarrow \mathcal{L}_1, ..., \mathcal{L}_n$, where $\mathcal{L}, \mathcal{L}_1, ..., \mathcal{L}_n$ are ground literals, and $rule_i$ is the label, or rule name, of the rule.

Conclusion of an argument rule $rule_i : \mathcal{L} \leftarrow \mathcal{L}_1, ..., \mathcal{L}_n$ is $\mathcal{L}$.

Premise of an argument rule $rule_i : \mathcal{L} \leftarrow \mathcal{L}_1, ..., \mathcal{L}_n$ is $\{\mathcal{L}_1, ..., \mathcal{L}_n\}$. It is the set of conditions required for the conclusion of the rule to be true.

Preference rules $\mathcal{P}$ is a set of clauses. A preference rule can either be *static* or *dynamic*. Static preference rules only has a head in the form $rule_1 < rule_2$, where $rule_1, rule_2$ are labels of rules defined in $\mathcal{T}$. $rule_1 < rule_2$ means that $rule_2$ has a higher priority over $rule_1$. Dynamic preference rules are defined similarly to static preference rules, but with the addition of a body. For example, the rule $(rule_1(X) < rule_2(X)) \leftarrow pred(X)$ means that $rule_1(X)$ will only be preferred to $rule_2(X)$ if $pred(X)$ is true.

Argument is a set of argument rules and preference rules. It takes the form (T, P), where $T \subseteq \mathcal{T}$ and $P \subseteq \mathcal{P}$.

Conclusion of an argument Given an argument (T, P), and a set of literals B, a conclusion of the argument is a literal that can be derived from T (given that the literals in B are true) and supported by P.

Premise of argument (T, P), is a set of all premises of T, i.e., $arg\_premise = \bigcup_{T_i \in T} premise(T_i)$.

Conflicting argument is an argument. An argument (T', P') is a *conflicting argument* of another argument (T, P) when the two arguments derive $\mathcal{L}$ and $\neg\mathcal{L}$ respectively, and rules used in the counter-argument (T', P') are at least 'as strong' as that in (T, P) in terms of priority between rules. This pair of arguments is also called *conflicting* arguments.

Derivation of an argument is a list containing - (i) labels $(rule_i)$ of argument rules applied (T) (ii) labels of preference rules applied (P) and (iii) labels of facts that proves the premises of the argument rules.

Non-monotonic reasoning

Preference-based argumentation allows us to handle *non-monotonic reasoning* in attribution, where the introduction of new evidence (literals) might change the result of the attribution (due to conflicting arguments). For example, given the argument pair (T, P):

$$\begin{aligned} T = \{ & rule_1 : \ attackOrigin(X, Attack) \leftarrow attackSourceIP(IP, Attack) \wedge ipGeoloc(X, IP), \\ & rule_2 : \neg attackOrigin(X, Attack) \leftarrow attackSourceIP(IP, Attack) \wedge ipGeoloc(X, IP) \\ & \qquad \wedge spoofedIP(IP)\} \end{aligned}$$

$$P = \{pref_1 : rule_2 > rule_1\}$$

If we only had the evidences:

$$E = \{attackSourceIP(ip1, attack1), ipGeoloc(countryX, ip1)\}$$

we will get $attackOrigin(countryX, attack1)$ as the conclusion of the argument. However, if we had the evidences:

$$E = \{attackSourceIP(ip2, attack2), ipGeoloc(countryX, ip2), spoofedIP(ip2)\}$$

then the conclusion will be $\neg attackOrigin(countryX, attack2)$ instead.

Abduction

By introducing a set of *abducible* predicates Ab, abduction allows us to make assumptions $\triangle$ ($\triangle \subseteq Ab$), for reaching a conclusion $\mathcal{L}$; as long as the assumptions $\triangle$ satisfy the *integrity constraints*. This gives us the power to make assumptions to reason with incomplete evidence, and give analysts a potential lead to follow and carry out further investigations. We describe in Section 4.6 how abducibles are used in *ABR*.

2.5.2 Logic programming

The core reasoner of *ABR* is built in Prolog, so we briefly go through the basics of first-order logic and programming in Prolog.

First-order logic

Prolog programs follow the first-order logic (FOL) style, so before we start on Prolog, we give a short introduction on FOL.

In FOL, sentences or concepts can be represented by *predicate symbols* and *constants*. For example, the concept 'United States imposed sanctions on Iran in 2012 Feb' can be represented as the literal $imposedSanctions(united\_states, iran,$ [2012, 2]). In this case, $imposedSanctions/3$ is the predicate symbol, '`/3`' denotes that the predicate symbol takes 3 arguments. $united\_states, iran$ and $[2012, 2]$ are constants. We can also use quantifiers in FOL. '$\exists$' is read as 'there is', denoting that there exists at least one substitution of the variable makes the formula true. $\forall$ is read as 'for all', denoting that for all substitutions of the variable, the formula must be true. By using quantifiers and propositional logic connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$), we can build more complicated formulas such as $\forall X, Att.hasMotive(X, Att), hasCapability(X, Att) \rightarrow isCulprit(X, Att)$.

Prolog

Prolog is one of the most popular declarative logic programming languages. It has three basic constructs: facts, rules and queries.

Facts Facts are literals that are universally true. They are written as a single literal with a '.' at the end.

Rules Rules denote information that is conditionally true. Prolog rules has the same meaning as a logic implication with exactly one head. They are written in the following format: $H\ :\ -B_1, \ldots, B_N.$. It can be rewritten into the following FOL formula: $H \leftarrow B_1 \wedge \ldots \wedge B_N$, read as: if $B_1, \ldots, B_N$ are all true, then H is true.

Queries A query asks the Prolog knowledge base (collection of facts and rules) if a specific formula is true. Queries can be made by typing the formula in the Prolog environment.

2.5.3 Gorgias

Gorgias is a preference framework built on top of Prolog. As mentioned in Section 2.3.5, the Gorgias syntax adds another layer on top of the basic Prolog syntax. All Gorgias constructs can be written in the following form: `rule(Label, Head, Body)` where `Label` is a string, `Head` is a single literal and `Body` is a list of literals. Below we show how the Prolog facts, rules and queries translate into its equivalent form in Gorgias.

```
% Fact
% Prolog style:
claimedResponsibility(guardiansOfPeace, sonyhack)
% Gorgias style:
rule(case5_f1(),claimedResponsibility(guardiansOfPeace,
 ↪  sonyhack),[]).

% Rule
% Prolog stye:
isCulprit(C,Att) :- hasMotive(C,Att), hasCapability(C,Att).
% Gorgias style:
rule(r_str__motiveAndCapability(C,Att),  isCulprit(C,Att),
 ↪  [hasMotive(C,Att), hasCapability(C,Att)]).
```

```
% Query (to be typed into Prolog environment)
% Prolog style:
| ?- isCulprit(X,Att).
% Gorgias style:
| ?- prove([isCulprit(X, Att)], D).
```

A caveat to keep in mind is that Prolog and Gorgias not only have different syntax, but are fundamentally different as well. If we use Prolog system predicates might not work when placed inside the Gorgias `rule/3` predicate. For example, the following query will fail when run with Gorgias-Visual on SWI-Prolog, even though in the literal is clearly true.

```
| ?- prove([member([a, [a,b,c]])], D).
false.
```

This is why in some of our rules, we use Prolog definitions for conditions outside of the Gorgias rules, like the following:

```
rule(bg8(),goodRelation(X,Y),[]) :-
 ↪  goodRelationList(X,L),member(Y,L).
```

In general, when using Prolog system predicates as part of the body, we put them outside of Gorgias rules using Prolog conditions, and when using predicates defined using Gorgias rules, we put them inside the Gorgias rules.

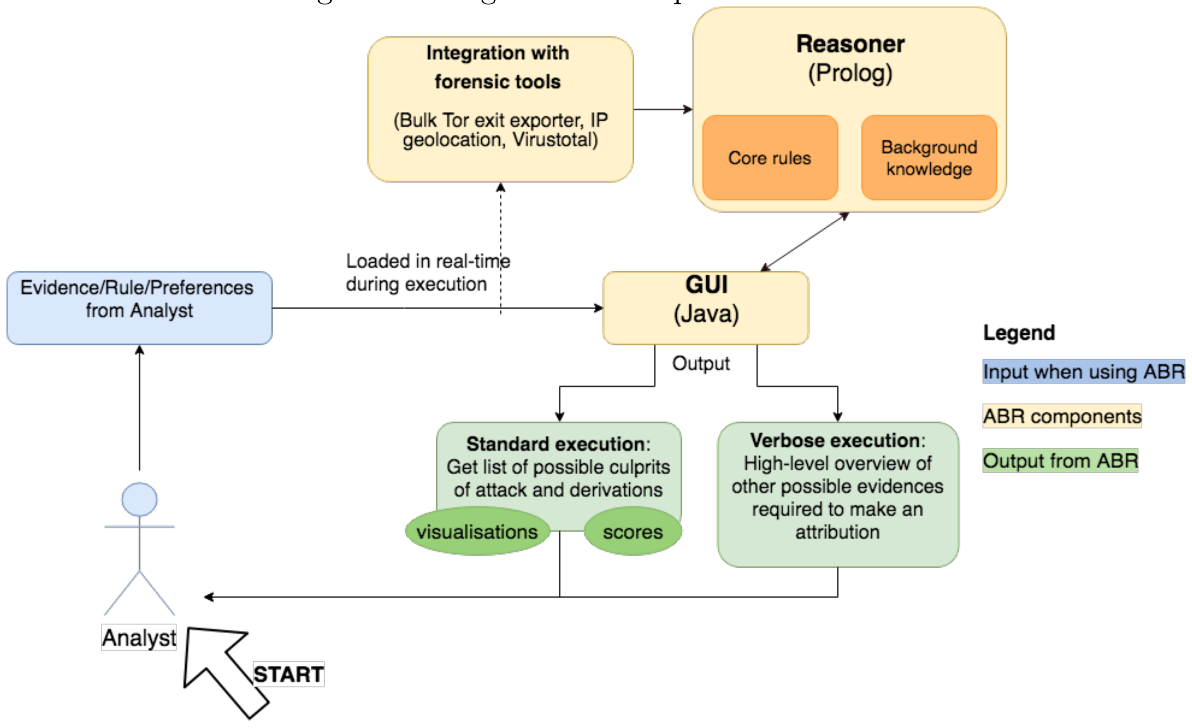
Chapter 3

ABR Overview

3.1 Overall architecture

In this chapter we outline how the *ABR* works on a high-level, showing the high-level architecture and providing a brief summary of the key components involved. Details regarding the various components can be found in Chapters 4 and 5. At the end of this chapter, we also summarise the key driving motivations that shaped how *ABR* was constructed.

Figure 3.1: High-level description of *ABR*


Integration with forensic tools
(Bulk Tor exit exporter, IP geolocation, Virustotal)
Reasoner
(Prolog)
Core rules
Background knowledge
Loaded in real-time during execution
Evidence/Rule/Preferences from Analyst
GUI
(Java)
Output
Legend
Input when using ABR
ABR components
Output from ABR
Standard execution: Get list of possible culprits of attack and derivations
visualisations
scores
Verbose execution: High-level overview of other possible evidences required to make an attribution
Analyst
START

The overall flow of information in *ABR* is illustrated by Figure 3.1. The user (analyst) interacts with the GUI, which returns and displays the results to the user. *ABR* is designed to be used in an iterative process, the user can add more evidences, rules or preferences after evaluating the results produced by *ABR*.

The users will input a set of evidences (technical and social) as well as rules and preferences through the GUI, which can trigger the integrated forensic tools to automatically extract some evidence. These evidences/rules/preferences are then used by the *ABR* reasoner, composed of the core rules and background knowledge, to execute the requested query; the result of which is returned to the GUI.

The result can take two different forms, depending on which mode of execution (*standard* or *verbose*) the *ABR* is executed in. Standard execution returns the list of possible culprits; while verbose execution, run when standard execution does not return any result, returns suggestions of what other evidences are required in order to get a result in standard execution.

3.1.1 Reasoner

The reasoner is the main component in *ABR*. It is constructed from two parts, (i) core Gorgias rules and (ii) background facts. The rules and background facts are used to analyse the given evidence and to produce a result (attribution of cyber attack) that is returned to the user.

Core Gorgias Rules The core rules of the *ABR* are built from well-known attribution cases. We study a full range of cyber attack cases in detail, extracted the reasoning used by investigators and converted them into rules. The details of which rules are extracted from which cases can be found in Appendix B.

Background facts Apart from the core Gorgias rules extracted from past cases, we have also compiled pertinent facts and rules as background facts, or background knowledge. The background knowledge contains information in two broad categories:

1. General knowledge that models the common sense and knowledge that analysts use during investigation, e.g., which countries use English as their first language.

2. Domain-specific knowledge that models past experiences and knowledge of analysts, e.g., what are the prominent Advanced Persistent Threat (APT) groups and what are their profiles.

Details of the rules and background facts used in *ABR* can be found in Sections 4.3 and 4.4.

3.1.2 Numerical scoring of solutions

Each derivation returned by *ABR* has a numerical score. The score can serve as a guideline for users to decide which derivation is stronger. This is useful especially when *ABR* attributes an attack to different groups or countries, since the user can decide, at a glance, which is the strongest attribution. We cover in detail how the score is calculated and its implementation details in Section 5.1.

3.1.3 Integration with forensic tools

In order to streamline the attribution process when using *ABR*, we have integrated it with some useful forensic tools. *ABR* is able to automate network-based intrusion detection system (NIDS) log parsing, IP geolocation and checking IPs against output from the Bulk Tor Exit Exporter. More details are covered in Section 5.2.

3.1.4 Visualisation

ABR supports the visualisation of three different structures: (i) derivation, (ii) attack-and-defense argumentation tree and (iii) overall hierarchy of rules. The derivation and the argumentation tree are returned by the Gorgias query in string form, we have implemented a visualisation component that transforms the string into a colour-coded, labelled graph, which is much easier to read.

The overall hierarchy of rules refers to how all the rules in *ABR* are ranked, according to the preference rules. This hierarchy graph shows which rules are stronger and which rules are weaker, and more importantly, also identify any cycles that were accidentally added.

3.1.5 User input

ABR users are able to input facts (evidences), rules, and preferences. We have implemented a variety of features that enhances this process, we go into the details of what actions can be performed by the user in Chapter 6.

3.2 Motivations for design

In this section we emphasise the key motivations behind the design of *ABR*. These are the motivations that shaped the design and development of various features in *ABR*.

Iterative Being able to model the iterative or incremental nature of attribution is one of the key features of *ABR*. In the real world, attribution is hardly a single atomic operation. Due to its complex nature, attribution is often a process, consisting of many different teams of people with different expertise, working over weeks or even months investigating and hypothesising. Therefore, to effectively fit into the existing work-flow of analysts, *ABR* aims to provide support alongside existing methods. Rather than merely returning True or False to whether a country or group was responsible for an attack, *ABR* provides possible information for further investigations, and compel its users to consider possible gaps in the reasoning process.

Transparency In order to fit into the iterative process of attribution, we need transparency. Only if users understand the current output, can they analyse it and either continue to build on top of it, or steer the reasoner in another direction.

Flexible We recognise that the *ABR* will not give the correct answer all the time, as the cases and evidences provided can differ greatly. Therefore, we aim not to build a perfect tool, but a flexible one, that gives users the ability to choose what kind of solutions they prefer.

Chapter 4

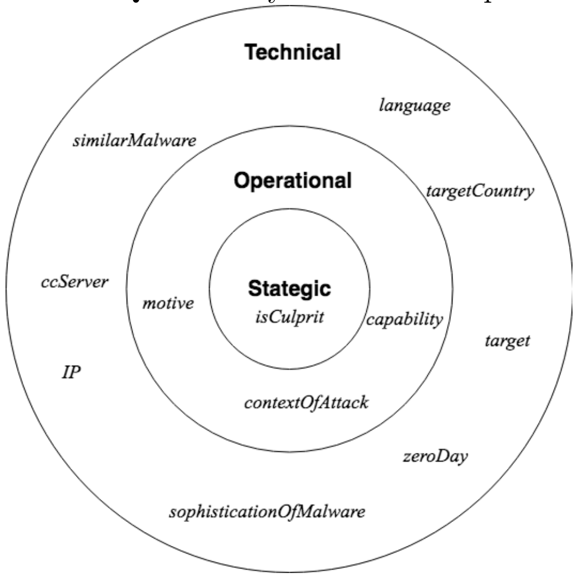
ABR Reasoner

The reasoner is a fundamental part of this project. Its execution permits the attribution to cyber attacks. In this chapter, we cover the various elements that compose the reasoner. The *ABR* reasoner is constructed based on the Q-model, that we introduce in the following section; followed by some key definitions and terms that will be used in explanations; then the core rules and background knowledge that constitutes the central part of the reasoner. Lastly, we discuss the use of preferences and abducibles in the reasoner.

4.1 Q-model

Attribution involves both technical and non-technical (or better called social) evidences. Our ABR reasoner works with both types of evidences. To represent and reason with these evidences, we based the *ABR* reasoner upon a model from Rid's work [10], called the *Q-model*.

Figure 4.1: Q-model layers with *ABR* predicates



The Q-model consists of three main layers: *technical*, *operational* and *strategic*. In Figure 4.1, we show the three layers of the Q-model with some of the ABR predicates associated to each layer. The combination of information in these three layers permits the attribution of a cyber attack. To illustrate, we show below some examples of rules for each of the three layers:

Technical

$$requireHighResource(Att) \leftarrow usesZeroDay(Att).$$

If the attack (Att) uses zero-day vulnerabilities ($usesZeroDay(Att)$), then we say that it requires a lot of resources ($requireHighResource(Att)$).

Operational

$$hasCapability(X, Att) \leftarrow hasResources(X), requireHighResource(Att).$$

If the attack (Att) required a large amount of resources ($requireHighResource(Att)$), and an entity X has (large amounts of) resources ($hasResources(X)$), then X has the capability to carry out the attack ($hasCapability(X, Att)$).

Strategic

$$isCulprit(X, Att) \leftarrow hasCapability(X, Att), hasMotive(X, Att).$$

If X has both the motive ($hasMotive(X, Att)$) and the capability ($hasCapability(X, Att)$) for carry out the attack, then X is the culprit ($isCulprit(X, Att)$).

4.2 Key definitions

Before we explain the rules in the reasoner, we introduce some definition that will be used throughout the explanations later in this chapter. For the first three definitions, we refer to the following example Gorgias rule:

$$rule(str\_rule\_1, isCulprit(C, Att), [hasMotive(C, Att), hasCapability(C, Att)]).$$

Rulename is the label or id for the rule, it appears as the first argument in the Gorgias rule. *str_rule_1* is the rulename of the example rule.

Head/head predicate is the second argument in the Gorgias rule. *isCulprit(C, Att)* is the head of the example rule.

Body predicates are the literals that occur between the square brackets, as the last argument of the Gorgias rule. *hasMotive(C, Att), hasCapability(C, Att)* are the body predicates of the example rule.

Prove When we say a literal L is *proved* (by a rule), we mean that L is the head of a rule, i.e., $rule(\_, L, [\_])$ and all the body predicates of the rule are satisfied (true).

Disprove When we say a literal L is *disproved* (by a rule), we mean that $\neg L$ is the head of a rule, i.e., $rule(\_, \neg L, [\_])$ and all the body predicates of the rule are satisfied (true).

Fact are heads of rules with no body predicates. Any literal L in a rule in the form *rule(_, L, [])* is a *fact*.

Background fact is a fact that is defined as part of the background knowledge (see Section 4.4 for details).

Evidence In the context of *ABR*, evidences are facts.

Technical evidence are evidence that are obtained from digital forensic processes, relating to *what* the attack was about and *how* it was carried out.

Operational evidence are non-technical evidence that relates to the geopolitical situation, relating to *who* performed the attack.

Base evidence are evidence that need to be input by the user directly, there are no rules that proves either the predicate or its negation.

Derived evidence In contrast with a *base evidence*, a *derived evidence* is a piece of evidence that is proved by another rule (all *heads* of a rules with a non-empty body are *derived evidences*).

Next, we describe the two components that forms the *ABR* reasoner, (i) the core rules and (ii) the background knowledge.

4.3 Core rules

The *core rules* are the Gorgias rules that performs the reasoning behind the attribution. These rules are crafted by studying various attribution cases and extracting the reasoning that investigators employed to make the attribution and translate them into argumentation rules (described in detail in Appendix B).

All three layers together, technical, operational and strategic, form the core rules in the reasoner. In this section, we give an overview of some of the strategic rules of the reasoner, then delve into one of the core rules of the reasoner to explain how the layers work together.

4.3.1 Strategic layer overview

We show below some of the rules from the strategic layer[1]. These rules describe some circumstances where we can prove an entity to be a culprit ($isCulprit(X, Att)$) or not ($\neg isCulprit(X, Att)$).

[1]For the sake of understandability and presentation purposes, we simplified the rule names used in this section. The actual rule names are parametrized as shown in Section 5.4.

$$\begin{aligned}
str\_rule\_1 :&\ isCulprit(X, Att) \leftarrow existingGroupClaimedResponsibility(X, Att).\\
str\_rule\_2 :&\ isCulprit(X, Att) \leftarrow hasMotive(X, Att), hasCapability(X, Att).\\
str\_rule\_3 :&\ isCulprit(X, A1) \leftarrow malwareUsedInAttack(M1, A1),\\
&\qquad similar(M1, M2), malwareLinkedTo(M2, X),\\
&\qquad notFromBlackMarket(M1),\\
&\qquad notFromBlackMarket(M2).\\
str\_rule\_4 :&\ \neg isCulprit(X, Att) \leftarrow \neg attackOrigin(X, Att).\\
str\_rule\_5 :&\ \neg isCulprit(X, Att) \leftarrow \neg hasCapability(X, Att).\\
str\_rule\_6 :&\ \neg isCulprit(X, Att) \leftarrow target(X, Att).
\end{aligned}$$

4.3.2 Rule walk-through

Since the layers in *ABR* are closely linked, instead of explaining each layer separately, we start from the strategic layer, then move forward to the operation and technical layer while explaining the rules. To illustrate how the rules in the different layers are connected, we present[2], in detail, one rule from start to end, from the strategic layer, to operational layer, to technical layer. Explanations of more rules can be found in Appendix A.

The rule we use in this example is the following rule from the strategic layer:

$$str\_rule\_2 : isCulprit(C, Att) \leftarrow hasMotive(C, Att), hasCapability(C, Att).$$

Rule *str_rule_2* uses the predicates $hasMotive/2$ and $hasCapability/2$; $hasMotive/2$ is a derived predicate proved in the operational layer and $hasCapability/2$ is a derived predicate proved in the technical layer. As there are many rules that proves these two predicates, we break them down into smaller parts and explain each part separately.

hasMotive(X, Att)

First, we cover the explanations for $hasMotive/2$.

Rule 1 We show below one of the operational rules used to derive $hasMotive/2$.

$$\begin{aligned}
op\_rule\_1 : hasMotive(C, Att) \leftarrow &target(T, Att), industry(T),\\
&hasEconomicMotive(C, T),\\
&contextOfAttack(economic, Att),\\
&specificTarget(Att).
\end{aligned}$$

This rule can be read as: if a country/group has an economic motive to attack the industry that was targeted in the attack, and the context of the attack was economic,

[2]For presentation purposes, rules in this section are presented in the actual Gorgias syntax, we instead prepend the rule with a label and use conventional logic operators like '←' for implication and ',' for logical AND.

and the attack had a specific target, then the country/group has motive to carry out the attack.

The predicates used in this rule are summarised as below:

target(T, Att) is a base evidence, it means T is the target of the attack Att.

industry(T) is one of the background facts, it is true when T is an industry (see Table 4.1 for the full list of background facts).

hasEconomicMotive(C, T) is also a base evidence. When $hasEconomicMotive(countryX, industryY)$ is true, it means $countryX$ will benefit economically from attacking $industryY$. For example, if countryX has identified industryY as a strategic emerging industry in official public documents such as white papers or other government reviews, we say that $hasEconomicMotive(countryX, industryY)$ is true.

specificTarget/1 An attack is assumed to be targeted unless more than one country is targeted ($targetCountry/2$)[3], or there are technical evidence that show that the malware was constructed specifically for attacking that particular organization or country ($specificConfigInMalware(M)$).

contextOfAttack/2 can be proven if the target is a 'normal' industry. A 'normal' industry, as opposed to a 'political' industry, are industries that are not closely related to a country's national interests (refer to Table 4.1 for more details).

Rule 2 Another operational rule used to derive $hasMotive/2$ is given below:

$$\begin{aligned} op\_rule\_2 : hasMotive(C, Att) \leftarrow & targetCountry(T, Att), attackPeriod(Att, Date1), \\ & hasPoliticalMotive(C, T, Date2), \\ & dateApplicable(Date1, Date2), \\ & contextOfAttack(political, Att), \\ & specificTarget(Att). \end{aligned}$$

This rule can be interpreted as: if a country has a political motive (we will explain later what this means) to attack the target country, and the order of events fits into the general timeline (more on $dateApplicable/2$ later), and the attack was a targeted attack, then the country has motive to perform the attack.

The predicates used in this rule can be summarised below:

targetCountry(T, Att) is a base evidence, denoting that the country T is the country of the target of the attack Att.

attackPeriod(X, Date) is also a base evidence, it denotes the date of the attack. $Date$ is a list, in the format $[YYYY, MM]$[4].

[3]Since $targetCountry/2$ is a base evidence, it can be omitted at the user's discretion, if the user deem the number of infections in another country is too insignificant.

[4]We exclude the day since in many cases even though the malware is discovered on a certain day, we are unsure of when the systems were actually infiltrated.

hasPoliticalMotive(C,T,Date2) is a derived predicate. We have one rule that derives $hasPoliticalMotive(C, T, Date2)$. If the target country has imposed sanctions on country C, then as a form of retaliation, country C has political motive to attack the target country T.

dateApplicable(Date1, Date2) is an auxiliary rule used to ensure that the triggering event (in this case, the imposing of sanctions) takes place before the attack, and occurred shortly before the attack. Alternatively, if the event is long-term and ongoing, we use the constant *ongoing* in place of the date of the event in the $[YYYY, MM]$ format, $dateApplicable(\_, ongoing)$ is always true.

contextOfAttack(political, Att) can be proved if either the target is a country, or the target is a 'political' industry (industries that are closely related to a country's national interests, such as the military or energy sector).

We show below the operational rules used to derive the above mentioned predicates.

$$\begin{aligned}
&op\_rule\_3 : hasPoliticalMotive(C, T, Date) \leftarrow imposedSanctions(T, C, Date).\\
&op\_rule\_4 : contextOfAttack(political, Att) \leftarrow target(T, Att), country(T).\\
&op\_rule\_5 : contextOfAttack(political, Att) \leftarrow target(T, Att), industry(Ind, T),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad politicalIndustry(Ind).
\end{aligned}$$

Rule 3 We move on to the next operational rule to derive $hasMotive/2$.

$$\begin{aligned}
op\_rule\_6 : hasMotive(X, Att) \leftarrow\; & target(T, Att), attackPeriod(Att, Date1),\\
& news(Event, T, Date2),\\
& dateApplicable(Date1, Date2),\\
& causeOfConflict(X, T, Event),\\
& specificTarget(Att).
\end{aligned}$$

The general idea of this rule is that if (i) an incident occurred in the target country and was publicized, and (ii) that incident is the cause of international conflict or tension with another country shortly before the attack, then the other country has motive to attack the target country.

To better explain this rule, we use a real-world example. The Sony Pictures hack in 2014 was attributed to North Korea, that allegedly attacked Sony Pictures in retaliation for the upcoming North Korean-based comedy "The Interview". The relevant evidences are as follows:

$$\begin{aligned}
&target(sony, sonyhack).\\
&attackPeriod(sonyhack, [2014, 11]).\\
&news(theInterview, sony, [2013, 10]).\\
&causeOfConflict(north_korea, sony, theInterview).
\end{aligned}$$

We know that the scandal revolving "The Interview" was publicized in October 2013 ($news(theInterview, sony, [2013, 10])$), and this sparked conflict between

North Korea and Sony Pictures ($causeOfConflict(north\_korea, sony, theInterview)$). The target of the attack was Sony Pictures ($target(sony, sonyhack)$) and the attack occurred in November 2014 ($attackPeriod(sonyhack, [2014, 11])$), which was shortly before the attack occurred. Using these evidences and the above rules, we are able to arrive at the conclusion that North Korea has motive to perform the Sony Pictures hack.

hasCapability(X, Att)

Next, we briefly cover the explanations for $hasCapability/2$, which is defined in both the operational and technical layer. We show below one of the rules in the operational layer that defines $hasCapability(X, Att)$.

$$\begin{aligned} op\_rule\_7 : hasCapability(X, Att) \leftarrow\ & requireHighResource(Att), \\ & hasResources(X). \end{aligned}$$

The predicate $requireHighResource(Att)$ can be derived from the technical layer. We show two such rules that derives $requireHighResource(Att)$:

$$\begin{aligned} tech\_rule\_1 : requireHighResource(Att) \leftarrow\ & target(T, Att), highSecurity(T). \\ tech\_rule\_2 : requireHighResource(Att) \leftarrow\ & highVolumeAttack(Att), \\ & longDurationAttack(Att). \end{aligned}$$

The predicates $highSecurity(T)$, $highVolumeAttack(Att)$ and $longDurationAttack(Att)$ are all base evidences. They have the following definitions:

highSecurity(T) (Organisation or company) T has high-security.

highVolumeAttack(Att) The attack had a high volume.

longDurationAttack(Att) The attack was performed over a long duration (few months or even years).

4.4 Background knowledge

The other part of the reasoner comprises of the background knowledge or background facts. Not all evidences or facts are given by the user, the background knowledge consists of other non-case-specific information. There are two main types of facts compiled in the background knowledge - (i) general knowledge, and (ii) domain-specific knowledge. We show in Table 4.1 the full list of predicates in the background file used by *ABR*.

4.4.1 General knowledge

The general knowledge consists of information about the (i) characteristics of countries, such as the first language used in the country and a measure of how advanced the country's cyber security scene is; (ii) international relations between nations; (iii) classification of the types of industry. These facts can be used together with evidences given by the user to make the attribution. Below we illustrate how the predicates are used in the rules in the various layers.

Language indicators Language indicators in malware can provide useful clues regrading the possible origin of attacks [10]. There are two types of language artefacts: (i) default system language settings and (ii) names used in code. In the technical layer, the following rules makes use of these language artefacts to deduce the origin of the attack:

$$\begin{aligned} lang1 : attackPossibleOrigin(X, Att) &\leftarrow sysLanguage(L, Att), \\ &\quad firstLanguage(L, X). \\ lang2 : attackPossibleOrigin(X, Att) &\leftarrow languageInCode(L, Att), \\ &\quad firstLanguage(L, X). \end{aligned}$$

If it is found that the general configuration language settings of the attacker's machine was language L, ($sysLanguage(L, Att)$), and the first language of country X is L, ($firstLanguage(L, X)$), then the attack might have originated from country X.

Otherwise, if names of variables or folders in the code are words from a certain language or reference a specific cultural reference from a specific language ($languageInCode(L, Att)$), then similarly, a country that uses L as its first language is a possible candidate for attack origin.

Capability of nation The capability of a nation limits the level of attacks it can possibly sponsor or carry out. We classify the amount of resources owned by a country into three different groups. A country can (i) have large amount of resources ($hasResources/1$) (ii) not have large amount of resources ($\neg(hasResources/1)$) (iii) have no resources ($hasNoResources/1$).

To estimate a country's cyber capabilities, we use the Global Cybersecurity Index (GCI) Groups 2017. The GCI is an integrated index that assesses countries based on their commitment to five core pillars: legal, technical, organisational, capacity-building and cooperation [39]. There are three GCI groups: *leading*, *maturing* and *initiating*. We represent a country's GCI group using the predicate $gci\_tier/2$. In the *operational* layer, we have rules for whether a country is capable of carrying out an attack ($hasCapability/2$):

$$\begin{aligned} hasResources1 &: hasResources(X) \leftarrow gci\_tier(X, leading). \\ hasResources2 &: hasResources(X) \leftarrow cybersuperpower(X). \\ hasResources3 &: hasNoResources(X) \leftarrow gci\_tier(X, initiating). \end{aligned}$$

We classify a country as $hasResources/1$ if it is either in the 'leading' GCI group or is recognised as one of the cyber superpowers [40]. Countries in the 'initiating' GCI group are classified as $hasNoResources/1$.

International relations Good international relations between two countries can indicate that a state-sponsored attack is unlikely to happen. While it is difficult to accurately portray relations between all countries, we used some simple rules and statistics [41, 42, 43] to compile a list of facts regarding whether two countries have a good relationship ($goodRelation/2$).

Information about relations between countries are incorporated in the *operational* layer to narrow down the countries that might have a motive to carry out the attack.

$$\begin{aligned}geopolitics1 : \neg hasMotive(C, Att) \leftarrow\ & targetCountry(T, Att), country(T),\\ & country(C), goodRelation(C, T).\end{aligned}$$

It should be recognized that this is an overly simplistic approach for classifying international relations. Public opinions on international relations could be different on the actual situation between two countries, or outdated, or it might not even be possible to simply classify a relation as simply 'good' or not. Therefore, even though we are able to prove that a country likely does not have motive to carry out an attack based on the estimated relation status, we have added preferences to prefer other rules when there is a conflict between the other rules and the rules that uses the background information on geopolitical situation.

4.4.2 Domain-specific knowledge

Domain-specific knowledge consists of information about (i) prominent groups and (ii) past attacks. These facts are primarily used in the *strategic* and *technical* layer.

Prominent APT groups Advanced Persistent Threat (APT) groups are well-organised hacker groups that usually pursue their objectives over months or years [44]. Due to their scale, many prominent APT groups are receive instructions and backing from nation states. We have extracted information on prominent APT groups from FireEye's report on APT groups [44] and Martin's article [45]. Each prominent APT group has (where available) the following facts:

- Name or ID of the group;
- Country of origin of group;
- Countries/organisations targeted by the group in the past;
- Malware linked to the group (malware suspected to be made by group).

We show below some example evidences regarding prominent APT groups that can be found in the background knowledge. These information are used in both the *strategic* and *operational* layer.

```
% information for few prominent APT groups shown,
% bg.pl contain other prominent groups that are omitted here.
rule(bg26(),prominentGroup(lazarusGrp),[]).
rule(bg27(),groupOrigin(lazarusGrp ,north_korea),[]).
rule(bg28(),malwareLinkedTo(backdoorDuuzer ,lazarusGrp),[]).
rule(bg29(),malwareLinkedTo(backdoorDestover ,lazarusGrp),[]).
rule(bg30(),malwareLinkedTo(infostealerFakepude ,lazarusGrp),[]).
rule(bg31(),malwareLinkedTo(backdoorContopee ,lazarusGrp),[]).
...
```

Since prominent APT groups are APT groups that are active and have past records of conducting long attacks on other organizations, they, by definition, have large amounts of resources, and are thus capable of carrying out any attack (denoted by the rule below):

$$str\_rule\_1 : hasCapability(X, Att) \leftarrow prominentGroup(X).$$

We can also link the APT group to its country of origin. If we are able to attribute the attack to an APT group and the country of origin of the APT group has motive to carry out the attack, we attribute the attack to the country (denoted by the rule below):

$$\begin{aligned} str\_rule\_2 : isCulprit(C, Att) \leftarrow\ & prominentGroup(Group), \\ & groupOrigin(Group, C), country(C), \\ & isCulprit(Group, Att), hasMotive(C, Att). \end{aligned}$$

Similarity to an APT-linked malware might also indicate that the culprit might be the same APT group (denoted by the rule below):

$$\begin{aligned} str\_rule\_3 : isCulprit(X, A1) \leftarrow\ & malwareUsedInAttack(M1, A1), \\ & similar(M1, M2), malwareLinkedTo(M2, X), \\ & notFromBlackMarket(M1), \\ & notFromBlackMarket(M2). \end{aligned}$$

In the *operational* layer, APT groups that have attacked the same targets before has motive to attack them again. While this alone is definitely insufficient to make the attribution, combined with other evidences, it can steer us towards the culprit of an attack (denoted by the rule below):

$$\begin{aligned} op\_rule : hasMotive(Group, Att) \leftarrow\ & target(T, Att), prominentGroup(Group), \\ & pastTargets(Group, Ts), member(T, Ts). \end{aligned}$$

Past attacks Information like the command and communication (C&C) server (*ccServer*/2) its domain registration details (*domainRegisteredDetails*/3) are included in the background file. We show an example of such details that can be found in `bg.pl`:

```
% ccServer(+DomainName, +MalwareName)
rule(bg79(), ccServer(gowin7, flame),[]).
% domainRegisteredDetails(+DomainName, +Name, +Address)
rule(bg80(),domainRegisteredDetails(gowin7, adolph_dybevek,
 ↪  prinsen_gate_6),[]).
```

These information can help us to spot similarities between past attacks and the current attack in the *technical* layer. We use the predicate $similar(M1, M2)$ to denote that two malware are similar to each other. This similarity can stem from both malwares using the same C&C server, or the C&C server registered under the same name or address.

Table 4.1: Background facts, divided into general knowledge (yellow) and domain-specific knowledge (orange)

Predicate example	Explanation
industry(infocomm)	Type predicate for industries[5]
politicalIndustry(military)	Political industries are industries that are closely related to a country's national interests
normalIndustry(infocomm)	Normal industries are industries that are not political industries
country(united_states)	Type predicate for countries
cybersuperpower(united_states)	List of cyber superpowers[6]
gci_tier(afghanistan,initiating)	
gci_tier(poland, maturing)	Global Cybersecurity Index (GCI)[7] group the country falls in
gci_tier(russian_federation,leading)	
firstLanguage(english, united_states)	First language used in the country
goodRelation(united_states, australia)	
prominentGroup(fancyBear)	Type predicate for prominent hacker groups
groupOrigin(fancyBear, russian_federation)	Country of origin of a group
pastTargets(fancyBear, [france,...,poland])	Past target (countries or organizations) of a hacker group
malwareLinkedTo(trojanMiniduke,cozyBear)	Past attribution of malware to hacker group/country
malwareUsedInAttack(flame, flameattack)	
ccServer(gowin7 ,flame)	Command and communication server of malware
domainRegisteredDetails(gowin7, adolph_dybevek, prinsen_gate_6)	Domain registration details of C&C server

4.5 Use of preference-based argumentation

In the next two sections we cover how preferences and abducible predicates were used in *ABR*.

ABR uses preference-based argumentation, where preferences are used to specify relative strength between conflicting rules[8]. In this section, we highlight the different ways in which preferences are used in the reasoner and justify why preferences are not used in some situations.

4.5.1 Conflicting rules with preferences

Some conflicting rules have preferences. When the bodies of both rules are satisfied, only one can be proven by Gorgias. There are two different ways in which we use preferences.

Negation as failure Firstly, we use this to model negation-as-failure (NAF)[9]. Since Gorgias facts does not have NAF by default, preferences are used to model NAF in the reasoner.

Contextual knowledge Secondly, we add preferences between conflicting rules that, using contextual knowledge, we determine that there is more reason to follow one rule and disregard the other. An example of such rules are the rules involving spoofed IP addresses.

$$\begin{aligned}
r\_t\_srcIP1(X, Att):\quad & attackPossibleOrigin(X, Att) \leftarrow attackSourceIP(IP, Att),\\
& \qquad ipGeoloc(X, IP).\\
r\_t\_spoofIP(X, Att):\ & \neg attackPossibleOrigin(X, Att) \leftarrow attackSourceIP(IP, Att),\\
& \qquad spoofedIP(IP, Att),\\
& \qquad ipGeoloc(X, IP).\\
prefer(r\_t\_spoofIP(X, Att), & r\_t\_srcIP1(X, Att)).
\end{aligned}$$

Normally, if we geolocated the source IP of the attack to a country, we derive that the attack originated from that country (*attackPossibleOrigin(X, Att)*). However, if we were also able to derive that the IP was spoofed, then we should instead arrive at that conclusion $\neg attackPossibleOrigin(X, Att)$.

4.5.2 Conflicting rules without preferences

We also left some conflicting rules without preferences. Such are for the situations where it might not be clear-cut whether to disregard one of the rules, and we leave the decision to the user. Since we run the negative derivation *neg(isCulprit(X,Att))* for every positive derivation *isCulprit(X,Att)* (refer to Section 6.4 for more details), if a derived evidence could be disproved (or the negation of the derived evidence

[8]To recap, conflicting rules are rules in which their heads are complementary to each other. So conflicting rules are any two rules with the form: $rule1 : L \leftarrow \ldots$ and $rule2 : \neg L \leftarrow \ldots$.

[9]In negation-of-failure (NAF), we say that $\neg p$ can be derived when we fail to derive p.

could be proved), then it will show up under the negative derivations. For example, given the following two conflicting technical rules:

$$\begin{aligned}
tech\_rule1(X, Att): &\quad attackOrigin(X, Att) \leftarrow attackPossibleOrigin(X, Att).\\
tech\_rule2(X, Y, Att): &\quad \neg attackOrigin(X, Att) \leftarrow attackPossibleOrigin(X, Att),\\
&\qquad attackPossibleOrigin(Y, Att),\\
&\qquad country(X), country(Y),\\
&\qquad X \backslash = Y.
\end{aligned}$$

We deliberately leave out the preference for these two rules in order to give the user more flexibility in deciding which rule to use. For example, if we managed to prove the following predicates:

$$\begin{aligned}
&f1 : attackPossibleOrigin(country1, attack).\\
&f2 : attackPossibleOrigin(country2, attack).\\
&f3 : country(country1).\\
&f4 : country(country2).
\end{aligned}$$

We are able to prove $attackOrigin(country1, attack)$ and $attackOrigin(country2, attack)$ by using the first rule, and we are also able to prove $\neg attackOrigin(country1, Att)$ and $\neg attackOrigin(country2, Att)$ by using the second rule. This is the desired result since we want to leave the user with some flexibility in which rules they think is stronger in the specific use case.

4.6 Use of abducibles

In this section, we describe the use of abducibles[10] in the *ABR* reasoner. After declaring a predicate as abducible, it can be used like normal predicates in the Gorgias rules.

There are two abducible predicates used in *ABR*, $specificTarget/1$ and $contextOfAttack/2$. These two abducibles are used in different ways. For $specificTarget/1$, we have rules to disprove the abducible predicate:

$$\begin{aligned}
\neg specificTarget(Att) \leftarrow\ & targetCountry(T1, Att), targetCountry(T2, Att),\\
& T1 \backslash = T2.
\end{aligned}$$

These rules can also be thought of as the *integrity constraints*. This models how some facts are assumed to be true unless proven otherwise.

On the other hand, for *contextOfAttack/2*, we have rules to prove the abducible predicates:

$$\begin{aligned}
contextOfAttack(economic, Att) &\leftarrow target(T, Att), industry(Ind, T),\\
&\qquad normalIndustry(Ind).\\
contextOfAttack(political, Att) &\leftarrow target(T, Att), country(T).\\
contextOfAttack(political, Att) &\leftarrow target(T, Att), industry(Ind, T),\\
&\qquad politicalIndustry(Ind).
\end{aligned}$$

[10]Abducible predicates are predicates that *can be* assumed to be true.

The abducible is then used as part of other rules. In this case, the attribution will go through even without proving $contextOfAttack/2$. It will be flagged up by the ABR as abduced predicates, prompting the user to look for more evidence in order to prove the $contextOfAttack/2$. This models the situation where there are incomplete information to fully make an attribution, but the analyst chooses to make some assumptions in order to put together other pieces of evidences. Later on, the analyst might go back and try to find more evidence to prove that the assumptions made during investigation were indeed true.

Chapter 5

Other Key Components

In the previous chapter, we introduced the *ABR* reasoner and details of its construction. Let us now explain the other key components that form *ABR*. We start with the (i) scoring system, followed by (ii) forensic tool integration, then (iii) visualisation. At the end, we present a short section on standardisation of rule names in *ABR*.

5.1 Scoring system

In *ABR*, on top of implementing the reasoner, we also devised a scoring system to compare different derivations. In this section, we cover the motivation for the scoring system, the intuition as to how the system works, and the actual implementation of the scoring system.

5.1.1 Motivation

When using *ABR*, if we have large amounts of evidence, our tool will likely produce several attributions, or different results for who was behind the attack. To reach a meaningful conclusion, it is essential that we construct a technique to rank or score different derivations.

5.1.2 Intuition

To uncover the intuition behind our scoring system, we first examine two different cases of comparisons between rules.

Case 1 Let us consider the two following rules, where both of them have $isCulprit($ $C, Att)$ as head, which is the conclusion of the rule.

$$rule1 : isCulprit(C, Att) \leftarrow hasMotive(C, Att), attackOrigin(C, Att), country(C).$$
$$rule2 : isCulprit(C, Att) \leftarrow attackOrigin(C, Att), country(C).$$

It is clear in the above example that $rule1$ should be stronger[1] than $rule2$ since the set of C and Att that satisfies the former rule is a proper subset of that for the latter rule. In other words, a more 'specific' rule is stronger than a 'generic' rule.

[1]'Stronger' here denotes the strength of argument in a colloquial sense.

Case 2 While the previous case is a straightforward comparison, since the body predicates of one rule are a subset of the body predicates of the other rule, there are also less obvious cases. For example, when comparing the following two rules:

$$rule1 : isCulprit(C, Att) \leftarrow hasMotive(C, Att), attackOrigin(C, Att), country(C).$$
$$rule3 : isCulprit(X, Att) \leftarrow existingGroupClaimedResponsibility(X, Att).$$

It is hard to spot any definite relation between the results from both rules. However, intuitively, we would say that *rule*1 is stronger than *rule*3, because *rule*3's body predicate simply a single piece of evidence (the presence of a group that claimed responsibility of an attack) while *rule*1 reached its conclusion by analysing the geographical origin of the attack and motives of a group or country.

Insight The above examples gives insight into how we intuitively decide if a rule is stronger. Generally, we say that a rule is stronger if its supporting evidences are more numerous and/or stronger. One simple way of scoring the derivations is to count the number of evidences used. Since there are two types of evidences: (i) case-specific evidence input by the user and (ii) background evidence, one possible refinement is to score case-specific evidence and background evidence differently. Case-specific evidence should weigh more than background evidence, since attributions based on more case-specific evidences should be stronger than attributions based mostly on background evidence. This alleviates the problem of always attributing attacks to enemy nations, regardless of case evidences, since the user will be able to see that the score of the derivation is low.

5.1.3 Implementation

When executing `prove([isCulprit(X,A)], D)`, the result of `D` is the derivation, which is a list of all the rule names of rules used to prove the predicate. This include both rules and facts[2]. We have chosen to assign each case-specific evidence a score of 3, and each background evidence a score of 1.

Since users input evidences in Prolog syntax (`Head :- Body1, Body2, ...`) and all Gorgias rules are created automatically, we have full control over the rule names. The mapping of the *ABR* rulenames to their rule types and the files where they are located in is shown in Table 5.1. Case-specific evidences have rule names starting with '`case_`' and background evidences have rule names starting with '`bg_`'

To demonstrate the scoring system at work, we show the below example.

```
D = [ass(notFromBlackMarket(flame)), case3_f13(), case3_f12(), r_t_bm(gauss),
↪  bg82(), bg86(), case3_f17(), r_t_simCC1(gauss, flame), r_t_similar(gauss,
↪  flame), case3_f2(), r_str__linkedMalware(equationGrp, gaussattack)]
```

The above derivation yields a score of 14, since there are 4 case-specific evidences {`case3_f13(), case3_f12(), case3_f17(), case3_f2()`} and 2 background evidences {`bg82(), bg86()`) used $(4 * 3 + 2 = 14\}$. The scores of all solutions are displayed to the user in the GUI, the user can decide if the score is to be taken into consideration when comparing different solutions for the attack.

[2]Facts are a special case of rules, they are rules with no body predicates.

5.2 Forensic tool integration

Integration with forensic tools is another important component of *ABR*. In this section we consider the motivations, followed by the implementation of the integration with each of the forensic tools used in *ABR*.

5.2.1 Motivation

Integrating with some forensics tools can help to incorporate *ABR* into the normal work-flow of analysts. Moreover, some information such as the geolocation and domain resolution of an IP address are very easy to find, but can be cumbersome to deal with when handling many IP addresses. *ABR* integrates with 4 different tools: (i) Snort, (ii) Tor Exit Exporter, (iii) ip2nation IP geolocation and (iv) Virustotal IP domain resolution. We give an overview of how all the tool integrations work together with *ABR*, before presenting the implementation details for each of the tools.

5.2.2 Implementation

The tool integrations are implemented in Java. The results are written into file as Gorgias facts and passed into the reasoner during the execution. The architecture diagram of the forensic tool integration is shown in Figure 5.1. It illustrates (i) the order in which the processing is done, (ii) the auxiliary predicates that each process scans for, (iii) the predicates produced by each process and (iv) the files produced and imported by each process. Uploading the Snort alert log is the only integration that requires user action. All the other integrations are started automatically whenever a user tries to execute a query; beginning by scanning the `user_evidence.pl` file for the auxiliary predicates described in the next paragraph.

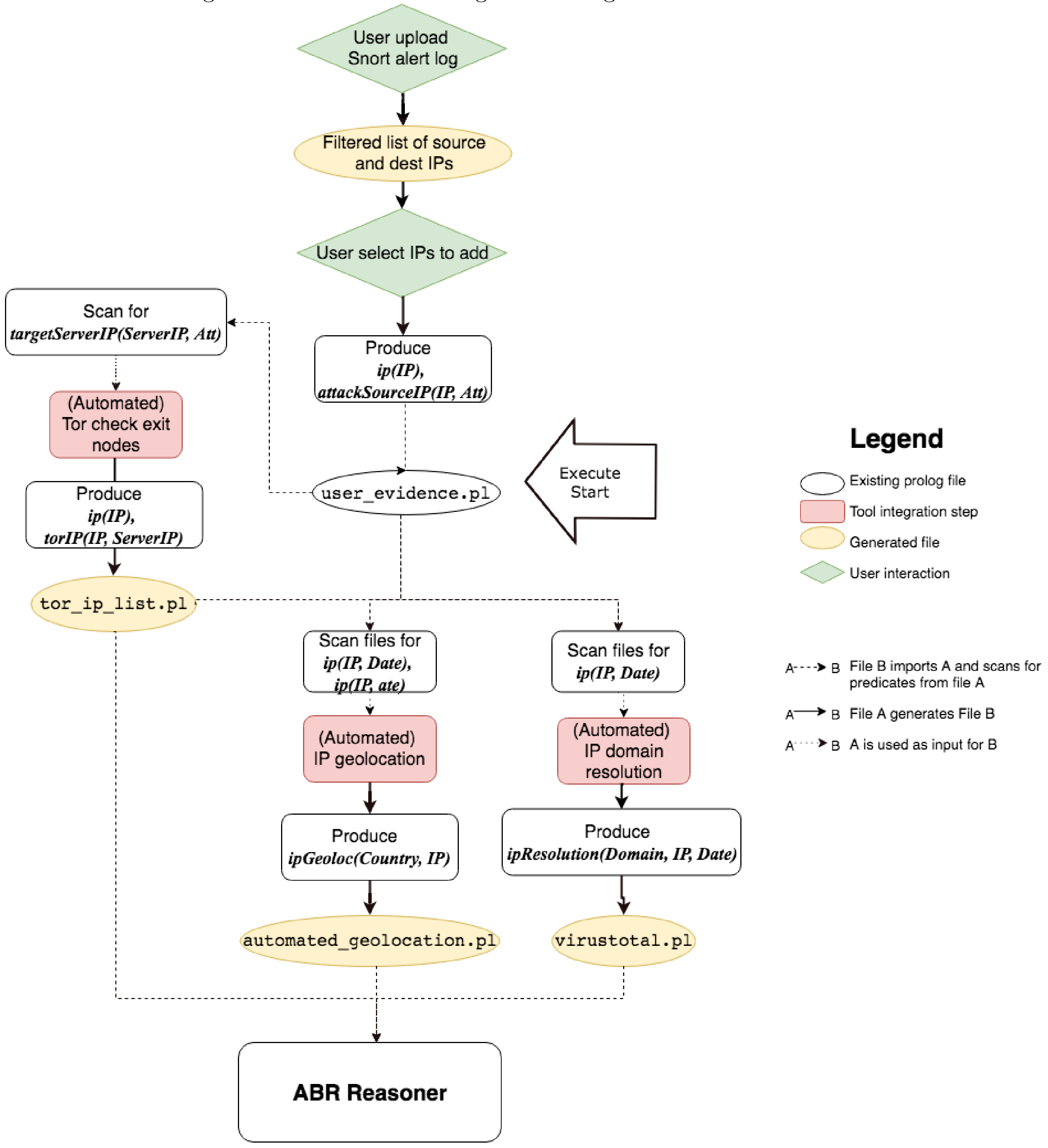
Some auxiliary predicates are used to mark an IP address for automatic processing for IP geolocation, IP resolution and checking for Tor exit nodes, indicated in Figure 5.1 as "Scan for *predicate*". These auxiliary predicates are summarised as below:

targetServerIP(ServerIP, Att) Given the IP address of the server targetted during an attack, for example $targetServerIP([72, 111, 1, 30], tor\_ex)$, Tor exit node exporter is triggered using the IP 72.111.1.30. The predicates $torIP([103, 1, 206, 100], [72, 111, 1, 30])$, ... are generated (one for each exit node returned by the exporter).

ip(IP) Given the predicate $ip/1$, for example $ip([8, 8, 8, 8])$, IP geolocation for 8.8.8.8 is triggered. The predicate $ipGeoloc(united\_states, [8, 8, 8, 8])$ is generated.

ip(IP, Date) Given the predicate $ip/2$, for example $ip([8, 8, 8, 8], [2018, 5])$, IP geolocation for 8.8.8.8 is triggered. Additionally, the Virustotal API is also triggered to find the domain resolution for 8.8.8.8 in May 2018. The predicates *ipGeoloc(united_states, [8,8,8,8])* and *ipResolution('00027.hk', [8,8,8,8], [2018,5])* are generated.

Figure 5.1: Architecture diagram of integration with forensic tools


User upload Snort alert log
Filtered list of source and dest IPs
User select IPs to add
Produce ip(IP), attackSourceIP(IP, Att)
user_evidence.pl
Execute Start
Scan for targetServerIP(ServerIP, Att)
(Automated) Tor check exit nodes
Produce ip(IP), torIP(IP, ServerIP)
tor_ip_list.pl
Scan files for ip(IP, Date), ip(IP, ate)
(Automated) IP geolocation
Produce ipGeoloc(Country, IP)
automated_geolocation.pl
Scan files for ip(IP, Date)
(Automated) IP domain resolution
Produce ipResolution(Domain, IP, Date)
virustotal.pl
ABR Reasoner
Legend
Existing prolog file
Tool integration step
Generated file
User interaction
A B File B imports A and scans for predicates from file A
A B File A generates File B
A B A is used as input for B

Next, we proceed to describe how the integration with each tool is implemented.

Tor exit node exporter

The Tor network[3] is often utilized by attackers to avoid surveillance via traffic analysis. Tor distributes the user's transactions over several different paths via different Tor nodes instead of taking a direct path to the target server of the requested page [46].

When attackers are using Tor to communicate with the victim's machine, we cannot determine the actual origin IP of the attacker. However, given the victim's server IP address, we can obtain a list of IP addresses of Tor exit nodes[4], using the Bulk Tor Exit Exporter tool[5]. Below we show an example output from the Bulk Tor Exit Exporter tool using 72.111.1.30 as target server's IP address.

```
# This is a list of all Tor exit nodes from the past 16 hours that can
# contact 72.111.1.30 on port 80 #
# You can update this list by visiting
# https://check.torproject.org/cgi-bin/TorBulkExitList.py?ip=72.111.1.30 #
# This file was generated on Wed May  2 19:04:11 UTC 2018 #
103.1.206.100
103.234.220.195
103.234.220.197
103.236.201.110
...
```

The process begins when the user clicks the 'Execute' button from the GUI. A series of steps will then be carried out, summarised as below:

1. Obtain server IP addresses of target machines by processing evidence file (`user_evidence.pl`) for predicate `targetServerIP/2` e.g., `targetServerIP([173,194,36,104],attackName)`.

2. Make request to https://check.torproject.org/ to get list of Tor nodes that might contact given target server IP addresses.

3. Add rule `torIP/2` relating Tor node to IP of target server, and `ip/1` to mark IP for geolocation later.

An example of the `tor_ip_list.pl` file generated if the predicate `targetServerIP( [173,194,36,104], attackName)` was present in the evidence file is the following:

```
:- multifile rule/3.
rule(case_torCheck0(), torIP([103,1,206,100], [173,194,36,104]),
 ↪  []).
rule(case_torCheck10(), ip([103,1,206,100]), []).
rule(case_torCheck1(), torIP([103,234,220,195], [173,194,36,104]),
 ↪  []).
```

[3]Tor project: https://www.torproject.org/about/overview.html.en

[4]Tor exit nodes are the nodes, or machines, that directly connects to requested pages outside the Tor network

[5]Bulk Tor Exit Exporter tool: https://check.torproject.org/cgi-bin/TorBulkExitList.py

```
rule(case_torCheck11(), ip([103,234,220,195]), []).
...
```

These predicates can then be used as part of the following rule from the technical layer:

$$\begin{aligned} spoofedIP(IP, Att) \leftarrow\ & malwareUsedInAttack(M, Att), attackSourceIP(IP, M), \\ & targetServerIP(TargetServerIP, Att), \\ & torIP(IP, TargetServerIP). \end{aligned}$$

Snort

ABR allows users to upload Snort alert logs and process them to allow users to select any suspicious IP addresses to mark them as an attack source IP (using the predicate `attackSourceIP(IP, Att)`).

Snort[6] is a popular network-based intrusion detection system (NIDS). NIDS are used to monitor the network for potential malicious activity [47]. When the IDS detects a possible malicious event, it then alerts security teams with useful information, e.g., sender and receiver IP addresses, timestamp, return code. Analysts can then look at the alerts and note down possible attack sources.

One of the main problems when using IDS is the large number of false positives generated [48], leading to security teams spending long hours reading through the alerts. Integrating the IDS output with the *ABR* allows analysts to make use of information obtained from the IDS without needing to read through every one of them.

While we are unable to fully automate the process of log analysis since there could potentially be thousands of alerts generated, and there are no strict rules to determining which alerts are suspicious and which are just false-alarms, we have implemented a log processor that filters the log and only displays part of the information. A snort alert has the following format [49] as shown below (we have highlighted the fields for alert message, priority level, source IP and destination IP, which we are interested in).

```
<timestamp> <hostname> <process_id>: [<gen_id>:<sig_id>:<version>] <alert_msg>
[Classification: <category_name>] [Priority: <priority>]
<network_interface> <ip_prot> <src_IP> -> <dest_IP>
```

The overview of the *ABR* Snort log processor is presented in Figure 5.2. We filter the alerts that satisfies the following conditions: (i) has *level* $>= 3$, (ii) message contains one of the keywords (see Figure 5.2 for list of keywords) and (iii) source IP is one of the 5 most occurring source IPs in the entire file. Using this filtering method, from the original 77680 alerts in the `tg_snort_full.7z` alert file from SecRepo[7], *ABR* only displays 155 alerts.

We show in Figure 5.3 the result shown to the user after uploading a Snort alert log. From this list, the user can click any IP to select it as a possible attack source. Once the user clicks an IP, say `ipAddr1`, then `attackSourceIP(ipAddr1)`

[6]Snort can be downloaded from https://www.snort.org/.

[7]The alert file from SecRepo can be downloaded from http://www.secrepo.com/

Figure 5.2: Processing of Snort alert logs

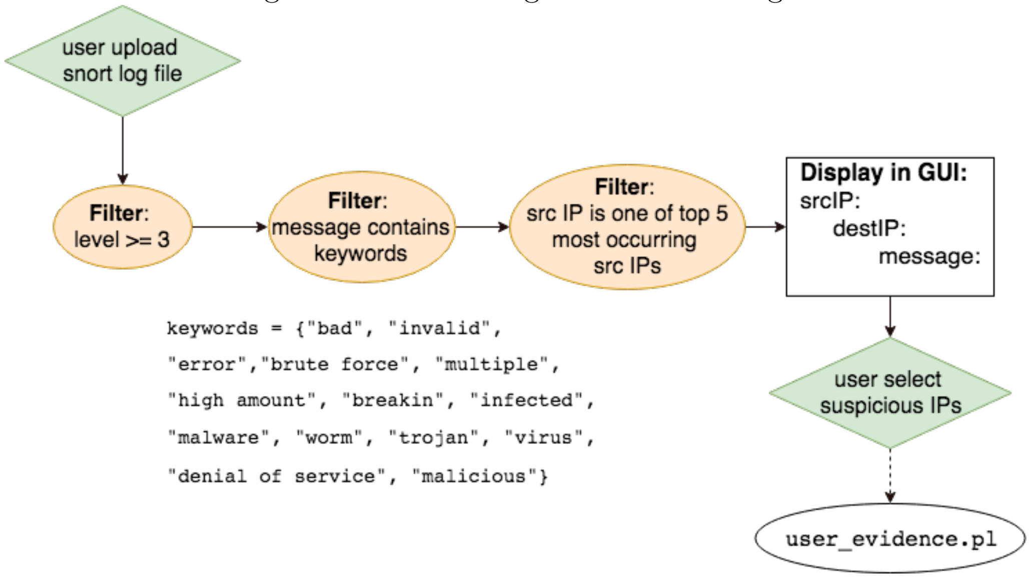
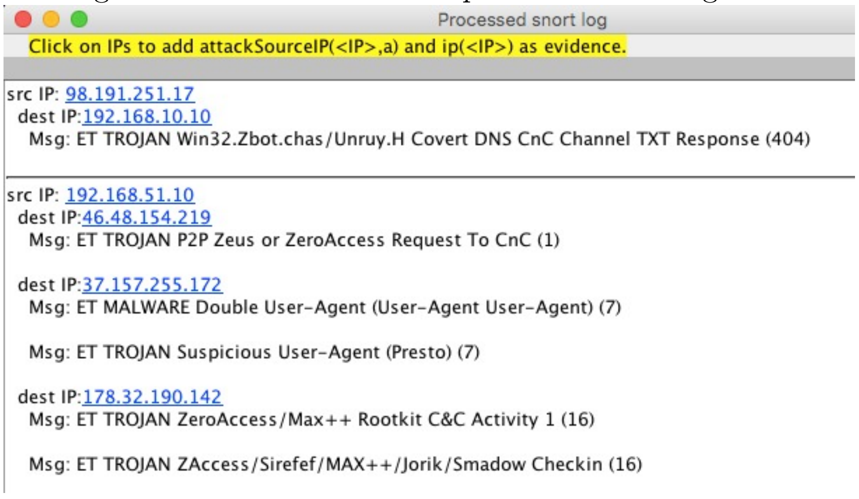

user upload snort log file
Filter: level >= 3
Filter: message contains keywords
Filter: src IP is one of top 5 most occurring src IPs
Display in GUI: srcIP: destIP: message:
keywords = {"bad", "invalid", "error","brute force", "multiple", "high amount", "breakin", "infected", "malware", "worm", "trojan", "virus", "denial of service", "malicious"}
user select suspicious IPs
user_evidence.pl

Figure 5.3: Screenshot of output from Snort log filter



Click on IPs to add attackSourceIP(<IP>,a) and ip(<IP>) as evidence.

```
src IP: 98.191.251.17
 dest IP:192.168.10.10
  Msg: ET TROJAN Win32.Zbot.chas/Unruy.H Covert DNS CnC Channel TXT Response (404)

src IP: 192.168.51.10
 dest IP:46.48.154.219
  Msg: ET TROJAN P2P Zeus or ZeroAccess Request To CnC (1)

 dest IP:37.157.255.172
  Msg: ET MALWARE Double User-Agent (User-Agent User-Agent) (7)

  Msg: ET TROJAN Suspicious User-Agent (Presto) (7)

 dest IP:178.32.190.142
  Msg: ET TROJAN ZeroAccess/Max++ Rootkit C&C Activity 1 (16)

  Msg: ET TROJAN ZAccess/Sirefef/MAX++/Jorik/Smadow Checkin (16)
```

and `ip(ipAddr1)` (auxiliary predicate) are added to `user_evidence.pl`. The predicate `attackSourceIP/1` is used by the following rule in the technical layer:

$$attackPossibleOrigin(X, Att) \leftarrow attackSourceIP(IP, Att), ipGeoloc(X, IP).$$

IP geolocation

ABR also incorporate automatic IP geolocation for any IP address marked in `user_evidence.pl`, `virustotal.pl` or `tor_ip_list.pl` files, using a free IP to country service ip2nation[8]. In this case, a http request is sent to `http://ip2c.org/<ip_address>`, which returns the name of country where the IP is geolocated. Country names are then converted in a standard way into prolog atoms, converting everything to lower case then replacing spaces with underscore. For example, if `ip([66,135,192,123])` appears in one of the prolog evidence files,

[8]ip2nation can be accessed from `https://about.ip2c.org/`

`ipGeoloc(united_states,[66,135,192,123])` will be generated in `automated_geolocation.pl`.

Below we show an example of `automated_geolocation.pl` generated:

```
rule(case_autogen_geolocation0(), ipGeoloc(china,[123,123,123,102]), []).
rule(case_autogen_geolocation1(), ipGeoloc(australia,[103,1,206,109]), []).
rule(case_autogen_geolocation2(), ipGeoloc(united_states,[69,195,124,58]), []).
...
```

IP geolocation is used to identify where the attack originated from, according to the following rule (taken from the technical layer):

$$attackPossibleOrigin(X, Att) \leftarrow attackSourceIP(IP, Att), ipGeoloc(X, IP).$$

Virustotal IP report

To get domain resolution of an IP on a certain date, we require the `ip/2` predicate in the form `ip([ip1,ip2,ip3,ip4], [YYYY, MM])`. For example, `ip([69,195,124,58], [2018,5])` will automatically generate the domain resolution for IP 69.195.124.58 in May 2018.

We show below an example output using Virustotal public API[9] using the IP address `69.195.124.58`.

```
___IP Report__
...
Resolutions
Host Name : 0556.info
Last Resolved : 2015-11-22 00:00:00
Host Name : 0564.info
Last Resolved : 2015-11-22 00:00:00
...
```

The IP report generated from Virustotal includes all the domain resolutions until the current date. To find out which domain the IP was registered to on the given date, we need to do some post-processing to the data. We first sort the resolutions according to its last resolved date, then iterate through the list to find the domain that the IP is registered to during the given date.

After obtaining the $ipResolution/3$ predicate, the following technical rule is used to deduce the command and control server (ccServer) used by the malware.

$$\begin{aligned} IPdomain1(S, M) : ccServer(S, M) \leftarrow\ & malwareUsedInAttack(M, Att), \\ & attackSourceIP(IP, Att), \\ & ipResolution(S, IP, \_D). \end{aligned}$$

5.3 Visualisation

ABR supports the visualisation of three different structures: (i) derivation, (ii) attack-and-defense argumentation tree and (iii) overall hierarchy of rules. In this

[9]Virustotal public API V2.0 adapted from https://kdkanishka.github.io/Virustotal-Public-API-V2.0-Client/

section, we detail the motivations, then the implementation details for the visualisation of each of the three structures.

5.3.1 Motivation

Without the graph visualisations, the original text form of the derivation and argumentation tree are rather hard to read and comprehend. On the other hand, without the visualisation of preference hierarchy, there is no way of spotting errors in preferences. Below we go into details for each of the structures, explaining the motivation for the visualisations.

Derivation

We show below an example derivation string, which is the list of rules used to derive the result of a query, in Listing 2.

Listing 2 Example derivation string

```
[r_op_notTargetted(example2b), case_example2b_f2b(), case_example2b_f2(),
 ↪  p4a_t(), bg1(), case_example2b_f10(), case_example2b_f9(),
 ↪  r_t_srcIP1(yourCountry, example2b), r_t_attackOrigin(yourCountry,
 ↪  example2b), case_example2b_f8(), r_str__motiveAndLocation(yourCountry,
 ↪  example2b)]
```

Although the derivation string contains all the information required to understand how the prove was constructed, users have to manually search for each of the rulenames to find out what the rules are.

Argument tree

We also show below the argument tree printed to standard output when we execute the query `visual_prove([isCulprit(X, Att)], D)` into the Prolog console, in Listing 3. The first line of the argument tree shows the *winning argument*, or

Listing 3 Example argument tree

```
[bg1(), case_example2b_f10(), case_example2b_f9(), r_t_srcIP1(yourCountry,example2b),
r_t_attackOrigin(yourCountry,example2b), case_example2b_f8(), r_str__motiveAndLocation(yourCountry,example2b)]  {DEFENSE}
|___[r_t_nonOrigin(yourCountry,example2b), r_t_noLocEvidence(yourCountry,example2b), p3_t()]
|   |___[r_t_srcIP1(yourCountry,example2b), case_example2b_f10(), case_example2b_f9(), p4a_t()]  {DEFENSE}
|___[r_str__targetItself2(yourCountry,example2b), case_example2b_f2(), p22d(), ass(specificTarget(example2b))]
    |___[r_op_notTargetted(example2b), case_example2b_f2b(), case_example2b_f2()
```

the derivation of the solution of the query. Subsequently, derivations of attack and defense (marked by `{DEFENSE}`) arguments are shown. For each defense argument, Gorgias attempts to find counter-arguments that are at least as strong as the given defense argument, and vice-versa. Similar to that for derivations, the default argument tree was also fairly hard to comprehend, the hierarchy of the various attack and defense arguments are not immediately obvious.

Rule hierarchy

Prior to implementing the visualisation of the rule hierarchy, there was no way of visualising the relationship between rules or preferences in ABR. Given the large number of preferences in the reasoner and the fact that the user can add more preferences, deadlocks or cyclic preferences can be easily introduced.

We show below a series of cyclic preferences, created by extracted part of the preferences used in the technical layer (preferences $p1$ to $p7$) and adding one additional preference ($p8$) that forms a cycle.

$$\begin{aligned}
&p1 : prefer(r\_t\_srcIP1(X, Att), r\_t\_noLocEvidence(X, Att)).\\
&p2 : prefer(r\_t\_srcIP2(X, Att), r\_t\_noLocEvidence(X, Att)).\\
&p3 : prefer(r\_t\_lang1(X, Att), r\_t\_noLocEvidence(X, Att)).\\
&p4 : prefer(r\_t\_lang2(X, Att), r\_t\_noLocEvidence(X, Att)).\\
&p5 : prefer(r\_t\_infra(X, Att), r\_t\_noLocEvidence(X, Att)).\\
&p6 : prefer(r\_t\_domain(X, Att), r\_t\_noLocEvidence(X, Att)).\\
&p7 : prefer(r\_t\_spoofIP(X, Att), r\_t\_srcIP1(X, Att)).\\
&p8 : prefer(r\_t\_noLocEvidence(X, Att), r\_t\_spoofIP(X, Att)).
\end{aligned}$$

It can be seen that it is quite difficult to spot cyclic preferences, especially when it is a large cycle. When there are cyclic preferences, ABR can get stuck, or not be able to prove anything, since there are no winning arguments if the rules are stuck in a deadlock.

5.3.2 Implementation

We now present a high-level description of the implementation of the various visualisations.

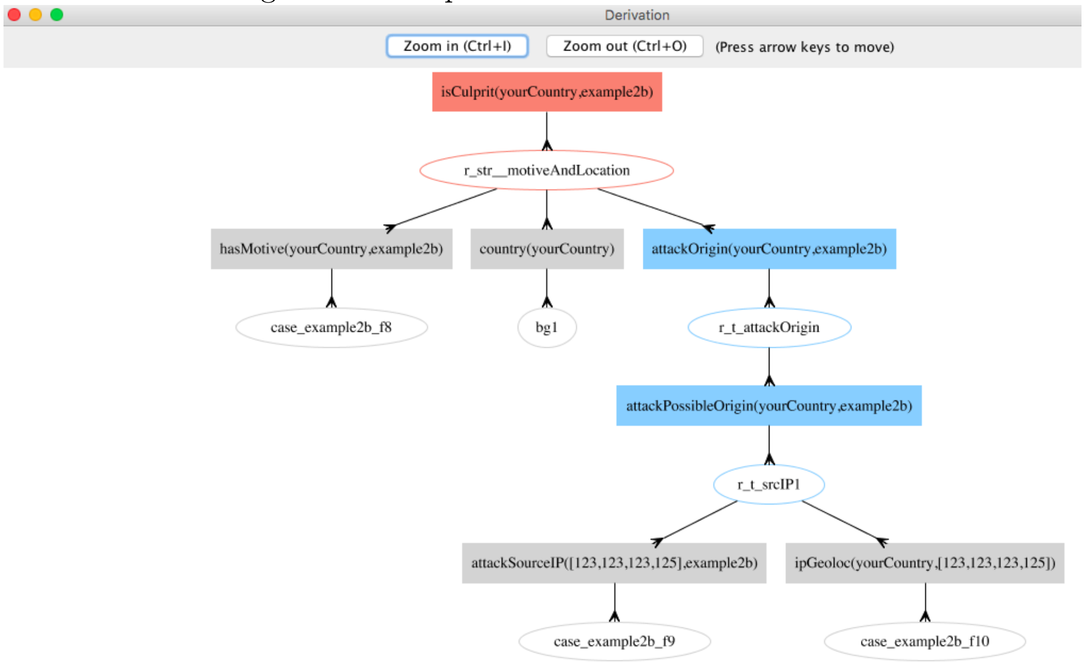
Derivation

The visualisation of the derivation string from Listing 2 is shown in Figure 5.4. This diagram clearly shows the hierarchy of rules applied, with the rulenames in the ellipses and evidences in the rectangles. It is also colour-coded to show which layer the evidence, or derived evidence had come from. Red denotes the strategic layer, yellow denotes the operational layer, blue denotes the technical layer and grey denotes either a base evidence or background knowledge.

Each rulename in the derivation corresponds to a rule, that has a head and a list of body predicates. The hierarchy of the derivation graph can be illustrated by the following example: A node ($n1 : H \leftarrow B_1, ..., B_N$) is a child of another node ($n2 : H_1 \leftarrow H, ...$) since the head of $n1$ is one of the body predicates of $n2$. This means that $n2$ uses the derived evidence from $n1$ during derivation.

The general idea is to create a tree structure from the list of rulenames in the derivation string. While iterating through the derivation, we either (i) push the rule onto a stack or (ii) pop rules from the stack and add them as the current rule's children if they derive the body predicates of the current rule, before pushing the parent onto the stack. The details of the algorithm are shown in Section 7.3.4.

Figure 5.4: Graph visualisation for derivation


Derivation
Zoom in (Ctrl+I)
Zoom out (Ctrl+O)
(Press arrow keys to move)
isCulprit(yourCountry,example2b)
r_str__motiveAndLocation
hasMotive(yourCountry,example2b)
country(yourCountry)
attackOrigin(yourCountry,example2b)
case_example2b_f8
bg1
r_t_attackOrigin
attackPossibleOrigin(yourCountry,example2b)
r_t_srcIP1
attackSourceIP([123,123,123,125],example2b)
ipGeoloc(yourCountry,[123,123,123,125])
case_example2b_f9
case_example2b_f10

Argument tree

The visualisation of the argument tree string from Listing 3 is shown in Figure 5.5. From this diagram, the relationship between attack and defense arguments are much clearer. We also show the preference rule applied that makes one argument stronger than the other. Like the derivation graph, the argument tree graph is also colour-coded. Green nodes denote defense arguments and red nodes denote attack arguments.

The hierarchy in an argument tree depends on the 'level' of the argument. Reviewing the argument tree previously mentioned, we see that there are 'branches' indicated by the "`|___[`" branch in the string. We refer to the depth of the branch as the 'level' of the argument.

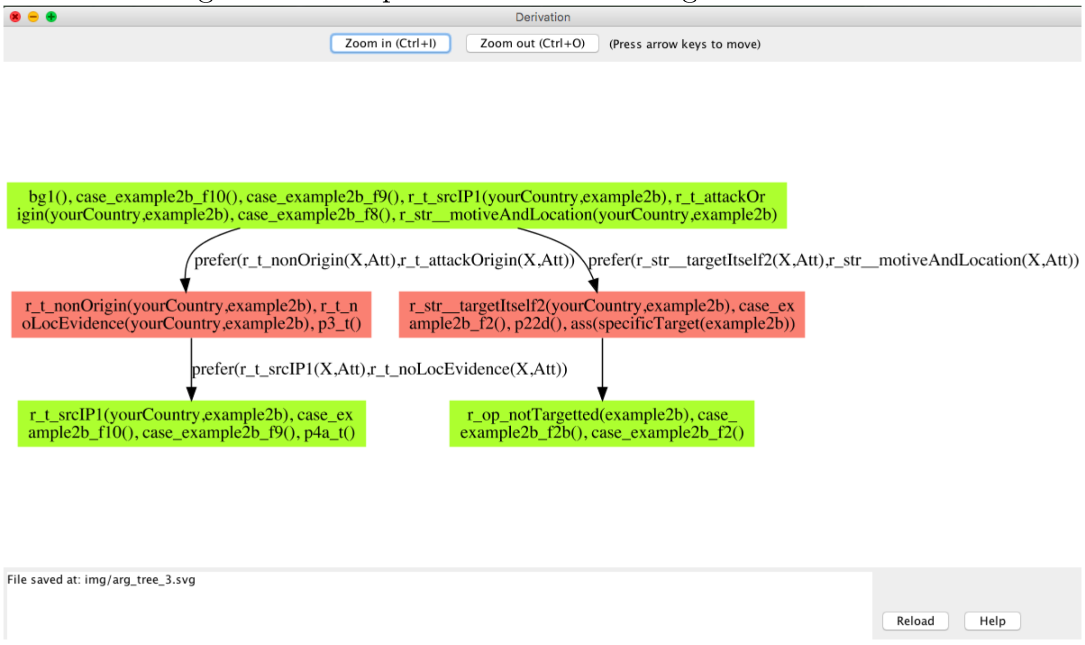
```
[bg1(), case_example2b_f10(), case_example2b_f9(), r_t_srcIP1(yourCountry,example2b),
r_t_attackOrigin(yourCountry,example2b), case_example2b_f8(), r_str__motiveAndLocation(yourCountry,example2b)]  {DEFENSE}
|___[r_t_nonOrigin(yourCountry,example2b), r_t_noLocEvidence(yourCountry,example2b), p3_t()]
|   |___[r_t_srcIP1(yourCountry,example2b), case_example2b_f10(), case_example2b_f9(), p4a_t()]  {DEFENSE}
|___[r_str__targetItself2(yourCountry,example2b), case_example2b_f2(), p22d(), ass(specificTarget(example2b))]
    |___[r_op_notTargetted(example2b), case_example2b_f2b(), case_example2b_f2()
```

An argument (with $level = n$) is a child of the closest argument above it that has $level = n - 1$. The algorithm for creating the graph from the argument tree string is similar to that for creating a graph from the derivation string. Starting from the last line of the string, we either (i) push nodes onto the stack or (ii) pop the children argument off and push the parent argument onto the stack. The details are described in Section 7.3.4.

Rule hierarchy

ABR can visualise the hierarchy of all the preferences in the reasoner and all the preferences added by the user. The generated graph highlights rules that are involved in cycles, so erroneous preferences can be spotted from a glance. Visualising the preference rules mentioned previously, Figure 5.6 shows the graph for the cyclic

Figure 5.5: Graph visualisation for argumentation tree


Derivation
Zoom in (Ctrl+I)
Zoom out (Ctrl+O)
(Press arrow keys to move)
bg1(), case_example2b_f10(), case_example2b_f9(), r_t_srcIP1(yourCountry,example2b), r_t_attackOrigin(yourCountry,example2b), case_example2b_f8(), r_str__motiveAndLocation(yourCountry,example2b)
prefer(r_t_nonOrigin(X,Att),r_t_attackOrigin(X,Att))
prefer(r_str__targetItself2(X,Att),r_str__motiveAndLocation(X,Att))
r_t_nonOrigin(yourCountry,example2b), r_t_noLocEvidence(yourCountry,example2b), p3_t()
r_str__targetItself2(yourCountry,example2b), case_example2b_f2(), p22d(), ass(specificTarget(example2b))
prefer(r_t_srcIP1(X,Att),r_t_noLocEvidence(X,Att))
r_t_srcIP1(yourCountry,example2b), case_example2b_f10(), case_example2b_f9(), p4a_t()
r_op_notTargetted(example2b), case_example2b_f2b(), case_example2b_f2()
Reload
Help

File saved at: img/arg_tree_3.svg

nodes and Figure 5.7 shows the graph if we remove the last preference ($p8$) that was creating the loop. With the rule hierarchy graph, users can check it after adding a new preference to ensure that the preferences are added correctly, no cycles are formed with existing preferences inside the ABR reasoner or any other previously defined preferences.

To create the rule hierarchy diagram, we first perform a scan for all the preferences used across the various files, then draw each pair of rules in a preference as nodes in a directed graph, with an edge from the stronger rule to the weaker rule. Special care was to be taken when detecting cycles. When drawing the graph, we iterate through the graph, drawing the parent (p) followed by its children ($\{c_1, ..., c_n\}$). Meanwhile, we keep track of the set of nodes that are seen. Starting from one node, we recursively traverse its children nodes. If we see a node the second time, we know that there is a cycle. All the nodes in that set form part of the cycle and are highlighted as ‘recursive preferences’ in the diagram.

Figure 5.6: Part of a usual preference graph

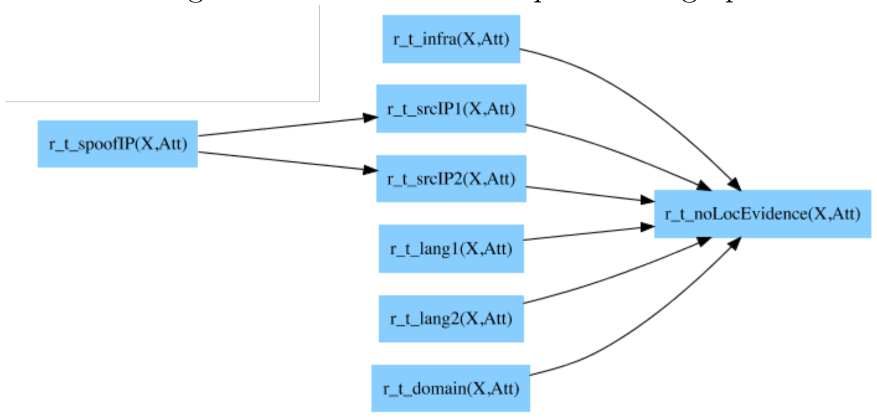
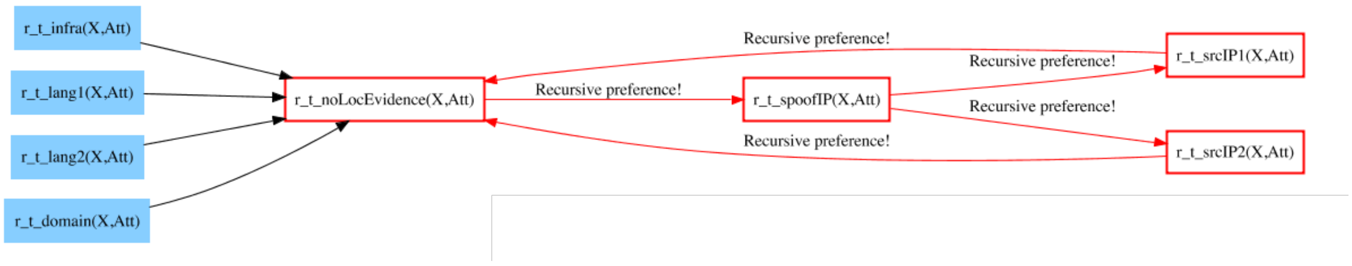


Figure 5.7: Recursive preference after adding `prefer(r_t_noLocEvidence(X,Att), r_t_spoofIP(X,Att))`



5.4 Standardised rule names

In *ABR*, we allow users to use Prolog syntax (as opposed to Gorgias syntax) as far as possible when inserting rules and facts. We made this choice for several reasons.

Accessibility While Prolog is a well-known logical programming language, Gorgias is much more niche, hence we do not expect our users to know the exact syntax for Gorgias rules and preferences.

Convenience The conversion of a Prolog rule (`Head :- Body1, ..., BodyN`) to a Gorgias (`rule( rulename, Head, [Body1, ..., BodyN])`) is very easy and can be easily automated. Since Prolog rules are shorter, allowing users to input rules in Prolog style saves them time in typing the longer, more cumbersome Gorgias rules.

Maintainability Constraining users to use Prolog rules also gives us complete control over the rule names. Having standardised rules for rule names makes it possible to determine the type of rule (preference rule or normal rule) and the file that the rule is located in, making the parsing and searching for the rules much easier. Table 5.1 summarises the format of the rule names used in *ABR*. Additionally, this is also a fundamental requirement that allows our scoring system (Section 5.1) to work.

Table 5.1: Standardised rule names

Rule name	Rule type	Location
r_t_<String>()	Normal	tech.pl
p<Int/String>_t()	Preference	tech.pl
r_op_<String>()	Normal	op.pl
p<Int/String>_op()	Preference	op.pl
r_str_<String>()	Normal	str.pl
p<Int/String>()	Preference	str.pl
case_user_f<Int>()	Fact	user_evidence.pl
p_user_<Int>()	Preference	user_evidence.pl
bg<Int>()	Background fact	bg.pl
case_torCheck<Int>()	Case-specific fact	toolIntegration/tor_ip_list.pl
case_virustotal_res<Int>()	Case-specific fact	toolIntegration/virustotal.pl
case_autogen_geolocation_<Int>()	Case-specific fact	toolIntegration/automated_geolocation.pl
case_<String/Int>()	Case-specific fact	evidence.pl

Chapter 6

ABR User Interface

ABR is a tool that is able to perform a myriad of different functions. Therefore, exposing and presenting these functions to the user in coherent manner is a major part in building *ABR*. In this chapter, we present the different ways in which the user can interact with the *ABR* GUI, and why the features are useful to the user.

In Figure 6.1 we show an annotated screen-shot that highlights the various options that the user can select in the *ABR* main screen. These options are illustrated in Figure 6.2. We present below a detailed description of the features of *ABR* for each option.

Figure 6.1: Annotated *ABR* main screen

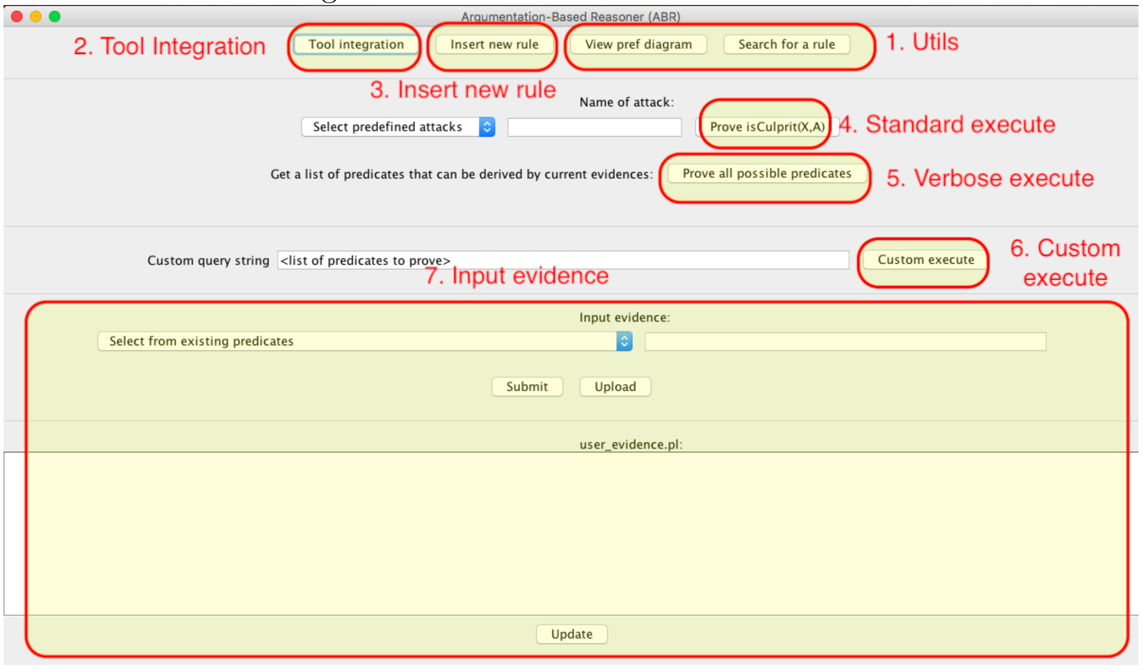
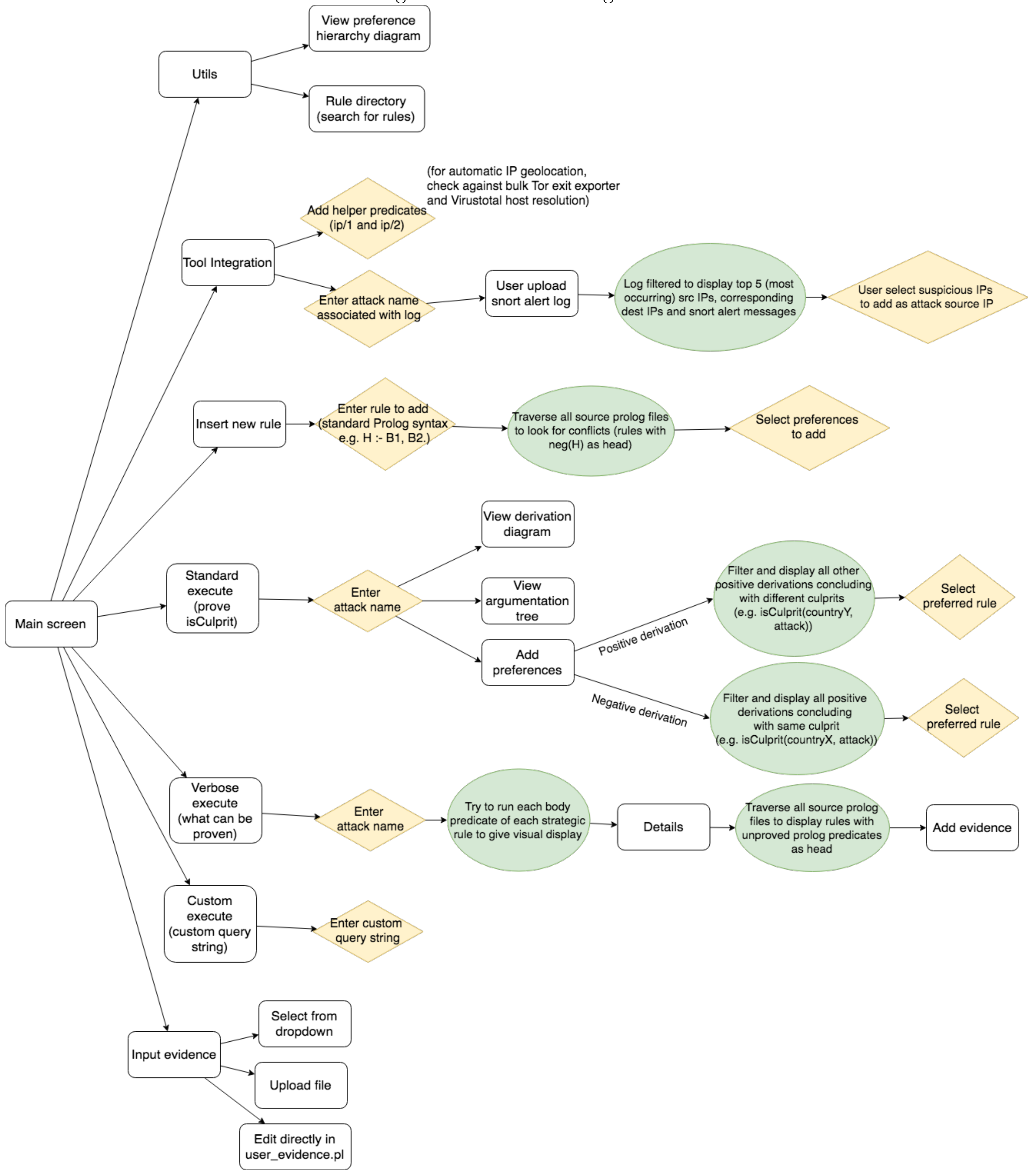


Figure 6.2: User flow diagram



Main screen
Utils
View preference hierarchy diagram
Rule directory (search for rules)
Tool Integration
Add helper predicates (ip/1 and ip/2)
(for automatic IP geolocation, check against bulk Tor exit exporter and Virustotal host resolution)
Enter attack name associated with log
User upload snort alert log
Log filtered to display top 5 (most occurring) src IPs, corresponding dest IPs and snort alert messages
User select suspicious IPs to add as attack source IP
Insert new rule
Enter rule to add (standard Prolog syntax e.g. H :- B1, B2.)
Traverse all source prolog files to look for conflicts (rules with neg(H) as head)
Select preferences to add
Standard execute (prove isCulprit)
Enter attack name
View derivation diagram
View argumentation tree
Add preferences
Positive derivation
Filter and display all other positive derivations concluding with different culprits (e.g. isCulprit(countryY, attack))
Select preferred rule
Negative derivation
Filter and display all positive derivations concluding with same culprit (e.g. isCulprit(countryX, attack))
Select preferred rule
Verbose execute (what can be proven)
Enter attack name
Try to run each body predicate of each strategic rule to give visual display
Details
Traverse all source prolog files to display rules with unproved prolog predicates as head
Add evidence
Custom execute (custom query string)
Enter custom query string
Input evidence
Select from dropdown
Upload file
Edit directly in user_evidence.pl

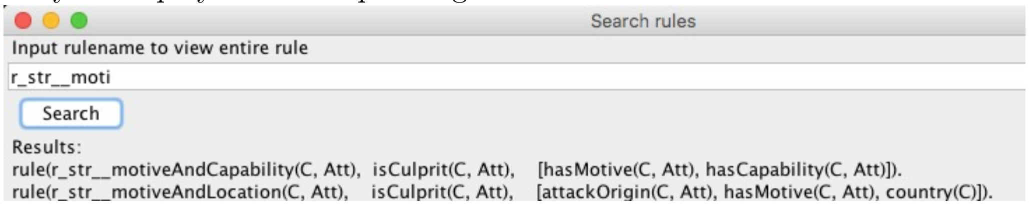
6.1 Utils

Users can view the rule hierarchy diagram and use the rule directory to search for the rule by its rule name. Given the large number of rules in *ABR* itself and numerous rules or preferences inserted by the user, without these tools, the user will quickly get lost in all the rules and preferences.

Rule hierarchy diagram The rule hierarchy diagram visualises the relative strength of rules by processing all the preferences used in *ABR*. Details of the rule hierarchy visualisation are covered in Section 5.3.

Rule directory The rule directory serves as a convenient way for users to look up what rule corresponds to specific rule names, as shown in Figure 6.3. In both the derivations and argumentation trees, only the rule names are displayed. While adding all the rules (e.g., `rule(rulename, head, [body1, ..., bodyn])`) will make it difficult to read the derivations and argumentation trees, having no rules at all and only the rule names makes the derivations incomprehensible. Having a rule directory is a good solution that allows the users to understand which rule corresponds to the rulenames, without cluttering the visualisations.

Figure 6.3: Screenshot of *ABR*, the name of the rule can be input into the rule directory to display the corresponding rules.



```
Results:
rule(r_str__motiveAndCapability(C, Att),  isCulprit(C, Att),   [hasMotive(C, Att), hasCapability(C, Att)]).
rule(r_str__motiveAndLocation(C, Att),    isCulprit(C, Att),   [attackOrigin(C, Att), hasMotive(C, Att), country(C)]).
```

6.2 Tool integration

By using the tool integration, users can insert the helper predicates (`ip/1, ip/2` and `targetServerIP/2`) required for *ABR* to generate (i) the IP geolocation, (ii) resolved domain name for the IP, and (iii) the list of Tor exit nodes contacting the target server. Users can also upload a snort NIDS alert log, where *ABR* filters the log and display the top 5 source IP addresses where users can select which addresses could potentially be the attack source IP for that attack. Further details regarding tool integration can be found in Section 5.2.

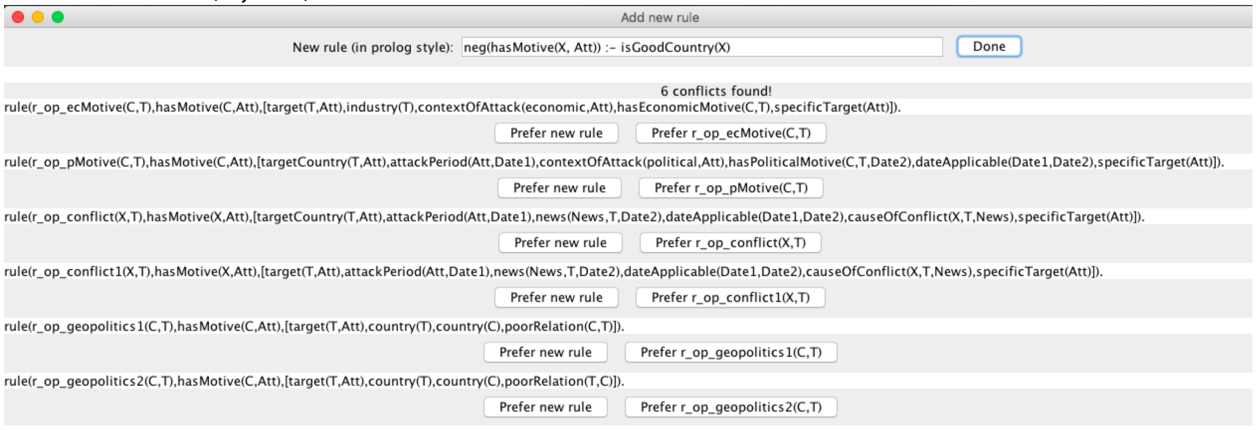
6.3 Insert new rule

When a new rule is inserted by the user, *ABR* automatically traverse through all the Prolog source files to look for conflicting rules[1] and lists them, as shown in Figure 6.4. The user can then add preferences between the new rule or the old rule.

[1]Conflicting rules are rules with heads that are complements of each other. In Gorgias, by default, `complement(neg(L), L)` is true for any literal `L`.

Adding preferences is essential for pruning out weak or invalid arguments, especially when large amounts of relevant evidences are present and many derivations are generated. Therefore, it is important to highlight counter-rules to the user to add in the necessary preferences.

Figure 6.4: Screenshot of *ABR*, where inserting new rule `neg(hasMotive(X,Att)) :- isGoodCountry(X)` displays all rules with `hasMotive(X,Att)` as head.


Add new rule
Done
Prefer new rule
Prefer r_op_ecMotive(C,T)
Prefer r_op_pMotive(C,T)
Prefer r_op_conflict(X,T)
Prefer r_op_conflict1(X,T)
Prefer r_op_geopolitics1(C,T)
Prefer r_op_geopolitics2(C,T)

New rule (in prolog style): neg(hasMotive(X, Att)) :- isGoodCountry(X)

6 conflicts found!

```
rule(r_op_ecMotive(C,T),hasMotive(C,Att),[target(T,Att),industry(T),contextOfAttack(economic,Att),hasEconomicMotive(C,T),specificTarget(Att)]).
rule(r_op_pMotive(C,T),hasMotive(C,Att),[targetCountry(T,Att),attackPeriod(Att,Date1),contextOfAttack(political,Att),hasPoliticalMotive(C,T,Date2),dateApplicable(Date1,Date2),specificTarget(Att)]).
rule(r_op_conflict(X,T),hasMotive(X,Att),[targetCountry(T,Att),attackPeriod(Att,Date1),news(News,T,Date2),dateApplicable(Date1,Date2),causeOfConflict(X,T,News),specificTarget(Att)]).
rule(r_op_conflict1(X,T),hasMotive(X,Att),[target(T,Att),attackPeriod(Att,Date1),news(News,T,Date2),dateApplicable(Date1,Date2),causeOfConflict(X,T,News),specificTarget(Att)]).
rule(r_op_geopolitics1(C,T),hasMotive(C,Att),[target(T,Att),country(T),country(C),poorRelation(C,T)]).
rule(r_op_geopolitics2(C,T),hasMotive(C,Att),[target(T,Att),country(T),country(C),poorRelation(T,C)]).
```

6.4 Standard execute

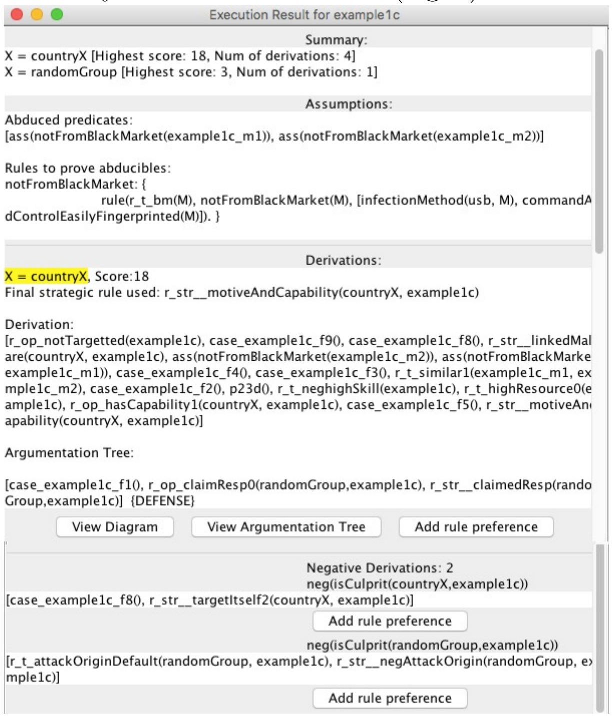
Executing in *standard mode* executes the query `visual_prove([isCulprit(X, <attackName>)], D)` using Gorgias for the given attack name. The standard execution returns the following information, as illustrated by Figure 6.5:

1. Who is the culprit;
2. Score of derivation;
3. The derivation that led to the conclusion;
4. Argument tree showing counter-arguments against the winning argument (the derivation);
5. Any negative derivations for the same culprit;
6. Abduced predicates used in derivation;
7. Rules that derives the abduced predicates.

By running the `visual_prove/2` query, Gorgias returns the derivation (`D`) that led to the conclusion, and also an argument tree. *ABR* processes the derivation and argument tree to show a graph diagram, which is much more comprehensible than the original output. The details regarding this visualisation is covered in Section 5.3.

ABR also shows negative derivations for each proved culprit (`neg(isCulprit(X, Att))` for each `X` in `isCulprit(X, Att)`). This shows which are the possible conflicting arguments, so users can then decide whether or not to add a preference to choose one argument or the other.

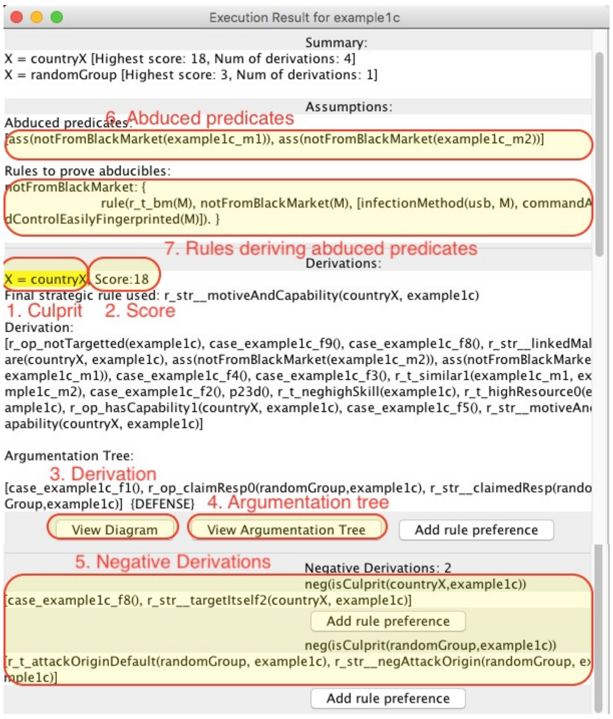
Figure 6.5: Original results screen (left) and annotated result screen highlighting key information shown (right).



```
Execution Result for example1c

Summary:
X = countryX [Highest score: 18, Num of derivations: 4]
X = randomGroup [Highest score: 3, Num of derivations: 1]

Assumptions:
Abduced predicates:
[ass(notFromBlackMarket(example1c_m1)), ass(notFromBlackMarket(example1c_m2))]

Rules to prove abducibles:
notFromBlackMarket: {
                rule(r_t_bm(M), notFromBlackMarket(M), [infectionMethod(usb, M), commandA
dControlEasilyFingerprinted(M)]). }

Derivations:
X = countryX, Score:18
Final strategic rule used: r_str__motiveAndCapability(countryX, example1c)

Derivation:
[r_op_notTargetted(example1c), case_example1c_f9(), case_example1c_f8(), r_str__linkedMal
are(countryX, example1c), ass(notFromBlackMarket(example1c_m2)), ass(notFromBlackMarke
example1c_m1)), case_example1c_f4(), case_example1c_f3(), r_t_similar1(example1c_m1, ex
mple1c_m2), case_example1c_f2(), p23d(), r_t_neghighSkill(example1c), r_t_highResource0(e
ample1c), r_op_hasCapability1(countryX, example1c), case_example1c_f5(), r_str__motiveAn
apability(countryX, example1c)]

Argumentation Tree:

[case_example1c_f1(), r_op_claimResp0(randomGroup,example1c), r_str__claimedResp(rando
Group,example1c)]  {DEFENSE}
```

```
Negative Derivations: 2
neg(isCulprit(countryX,example1c))
[case_example1c_f8(), r_str__targetItself2(countryX, example1c)]
```

```
neg(isCulprit(randomGroup,example1c))
[r_t_attackOriginDefault(randomGroup, example1c), r_str__negAttackOrigin(randomGroup, ex
mple1c)]
```



```
Execution Result for example1c

Summary:
X = countryX [Highest score: 18, Num of derivations: 4]
X = randomGroup [Highest score: 3, Num of derivations: 1]

Assumptions:
Abduced predicates:
[ass(notFromBlackMarket(example1c_m1)), ass(notFromBlackMarket(example1c_m2))]

Rules to prove abducibles:
notFromBlackMarket: {
                rule(r_t_bm(M), notFromBlackMarket(M), [infectionMethod(usb, M), commandA
dControlEasilyFingerprinted(M)]). }

Derivations:
X = countryX, Score:18
Final strategic rule used: r_str__motiveAndCapability(countryX, example1c)

Derivation:
[r_op_notTargetted(example1c), case_example1c_f9(), case_example1c_f8(), r_str__linkedMal
are(countryX, example1c), ass(notFromBlackMarket(example1c_m2)), ass(notFromBlackMarke
example1c_m1)), case_example1c_f4(), case_example1c_f3(), r_t_similar1(example1c_m1, ex
mple1c_m2), case_example1c_f2(), p23d(), r_t_neghighSkill(example1c), r_t_highResource0(e
ample1c), r_op_hasCapability1(countryX, example1c), case_example1c_f5(), r_str__motiveAn
apability(countryX, example1c)]

Argumentation Tree:
[case_example1c_f1(), r_op_claimResp0(randomGroup,example1c), r_str__claimedResp(rando
Group,example1c)]  {DEFENSE}

Negative Derivations: 2
neg(isCulprit(countryX,example1c))
[case_example1c_f8(), r_str__targetItself2(countryX, example1c)]
neg(isCulprit(randomGroup,example1c))
[r_t_attackOriginDefault(randomGroup, example1c), r_str__negAttackOrigin(randomGroup, ex
mple1c)]
```

To aid the user in evaluating and improving on the attribution, *ABR* shows a list of the abduced predicates[2]. By showing all the possible rules that can prove the abducible predicate, we highlight to the user what are the possible paths to investigate further.

6.5 Verbose execute

When `isCulprit/2` cannot be proved in *standard* mode, users can start executing in *verbose* mode. Executing in *verbose* mode attempts to prove each body predicate in all the rules with `isCulprit/2` as head, i.e., for each rule:

$$rule(r1, isCulprit(X, Att), [B_1, \dots, B_N]).$$

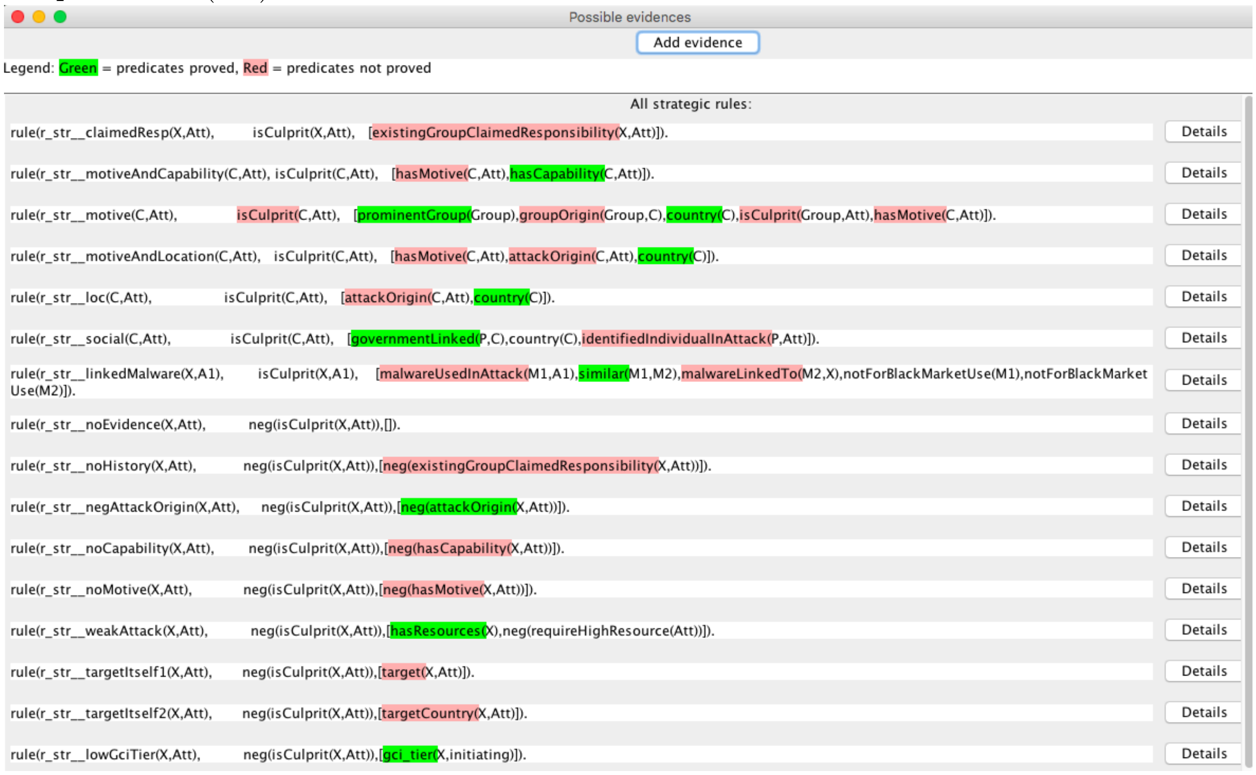
$\forall B \in \{B_1, \dots, B_N\}$, execute $prove([B], D)$. The results are then processed to give users a visual overview of what predicates are missing to make an attribution. From Figure 6.6, users can easily see what additional evidences are required to make an attribution and thus investigate further in that direction and add more evidence to prove `isCulprit/2`.

6.6 Custom execute

Other than executing the query `prove([isCulprit(X, Att)], D)`, users can also execute their own custom query (execute `prove([<customQuery>], D)`) from *ABR*. This could be useful when trying to test if a derived evidence can be proved. For

[2]Abduced predicates are assumed to be true, but not explicitly proven.

Figure 6.6: Verbose execution result visualising derivable (green) and non-derivable predicates (red)


Possible evidences

Add evidence

Legend: Green = predicates proved, Red = predicates not proved

All strategic rules:

```
rule(r_str__claimedResp(X,Att),        isCulprit(X,Att),   [existingGroupClaimedResponsibility(X,Att)]).
rule(r_str__motiveAndCapability(C,Att), isCulprit(C,Att),   [hasMotive(C,Att),hasCapability(C,Att)]).
rule(r_str__motive(C,Att),             isCulprit(C,Att),   [prominentGroup(Group),groupOrigin(Group,C),country(C),isCulprit(Group,Att),hasMotive(C,Att)]).
rule(r_str__motiveAndLocation(C,Att),  isCulprit(C,Att),   [hasMotive(C,Att),attackOrigin(C,Att),country(C)]).
rule(r_str__loc(C,Att),                isCulprit(C,Att),   [attackOrigin(C,Att),country(C)]).
rule(r_str__social(C,Att),             isCulprit(C,Att),   [governmentLinked(P,C),country(C),identifiedIndividualInAttack(P,Att)]).
rule(r_str__linkedMalware(X,A1),       isCulprit(X,A1),    [malwareUsedInAttack(M1,A1),similar(M1,M2),malwareLinkedTo(M2,X),notForBlackMarketUse(M1),notForBlackMarket
Use(M2)]).
rule(r_str__noEvidence(X,Att),         neg(isCulprit(X,Att)),[]).
rule(r_str__noHistory(X,Att),          neg(isCulprit(X,Att)),[neg(existingGroupClaimedResponsibility(X,Att))]).
rule(r_str__negAttackOrigin(X,Att),    neg(isCulprit(X,Att)),[neg(attackOrigin(X,Att))]).
rule(r_str__noCapability(X,Att),       neg(isCulprit(X,Att)),[neg(hasCapability(X,Att))]).
rule(r_str__noMotive(X,Att),           neg(isCulprit(X,Att)),[neg(hasMotive(X,Att))]).
rule(r_str__weakAttack(X,Att),         neg(isCulprit(X,Att)),[hasResources(X),neg(requireHighResource(Att))]).
rule(r_str__targetItself1(X,Att),      neg(isCulprit(X,Att)),[target(X,Att)]).
rule(r_str__targetItself2(X,Att),      neg(isCulprit(X,Att)),[targetCountry(X,Att)]).
rule(r_str__lowGciTier(X,Att),         neg(isCulprit(X,Att)),[gci_tier(X,initiating)]).
```

example, to check if malware `m1` and `m2` are similar, users can execute the query `prove([similar(m1, m2)], D))`.

6.7 Input evidence

There are several modes of input for insertion of evidence. Users can select from the dropdown menu of existing predicates, input the evidence directly, or upload a Prolog file. The evidence is then processed into a Gorgias rule. For example, the evidence `fact1` will be processed to `rule(case_user_f0, fact1, [])`.

Chapter 7

Implementation Details

In this chapter we explain the implementation details of the *ABR* components that were written in Java.

7.1 Summary of *ABR* functionalities

The functionalities of the Java component of *ABR* is summarised as follows:

- Standard execution: Extract derivations, argument tree and abduced predicates from Prolog query result.
- Verbose execution: Try to prove all predicates in strategic rules.
- Utility: Parse Gorgias rules to get head and body predicates.
- Forensic tool integrations.

7.2 Technology stack

The primary development language of the *ABR* back-end is Java. Besides the standard java libraries, we also use the `java-graphviz` and `jpl` libraries for graph visualisation and interfacing with SWI-Prolog respectively. To connect with forensic tools, the Virustotal public API client[1] was used.

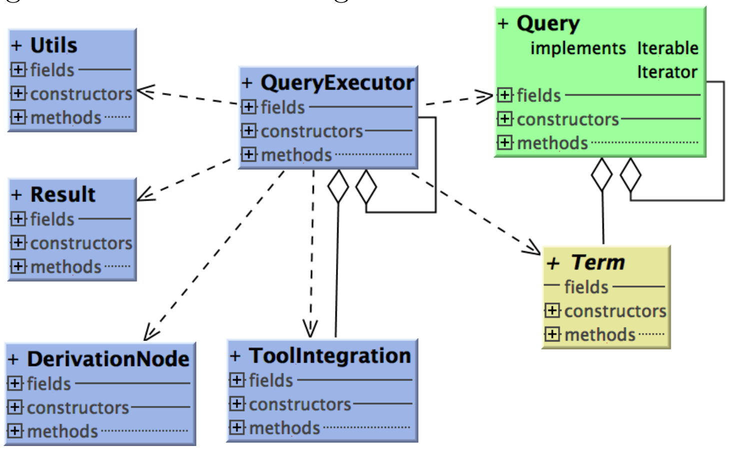
7.3 Implementation details

The following classes make up the Java back-end of *ABR*, represented in Figure 7.1:

- QueryExecutor.java: Main class that interacts with `JPL` package to execute Prolog queries.
- Result.java: Data structure to hold all information relating to the result of a query execution.
- Utils.java: Contains String constants and utility methods that aid in parsing of rules.

[1] http://kdkanishka.github.io/Virustotal-Public-API-V2.0-Client/

Figure 7.1: UML class diagram of Java back-end of *ABR*



- DerivationNode.java: Data structure used in generation of graph diagrams using `java-graphviz` package.
- ToolIntegration.java: Helper class containing all the methods relating to forensic tool integration.

Below we describe the above components in detail.

7.3.1 QueryExecutor.java

The QueryExecutor class handles execution of Prolog queries. In *ABR*, users can perform two different types of executions, *standard* and *verbose*. These two types involve different implementations, in this section we cover the implementation details for both of them.

Standard execution

When executing in standard mode, *ABR* executes the query `visual_prove([isCulprit(X, <attackName>)], D)`, then extracts the following information from the result:

1. `X`: Who is the culprit;
2. `D`: The derivation that led to the conclusion;
3. Any negative derivations for the same `X`;
4. Abduced predicates used in derivation;
5. Rules with the abduced predicates as head;
6. Argumentation reasoning tree that was printed to standard output.

The culprit and derivation of each solution can be obtained from the query result directly, by calling `Term culprit = map.get("X")` and `Term derivation = map.get("D")` respectively on the result map returned by the JPL method. To obtain the negative derivations, we execute `prove([neg(isCulprit(X, Att)], D))` for each of the culprits `X` obtained from the positive derivation. Abduced predicates can be extracted from the derivation by filtering for rulenames starting with "`ass(`", since the abduced predicates appear as `ass(abduciblePred)` in the derivation.

Verbose execution

The verbose query mode is used when we have insufficient evidence to make the attribution and there are no solutions during standard execution. When executing the verbose query, we would like to see a visual overview of what predicates can be derived with the available evidence, as shown in Figure 6.6. The verbose execution algorithm involves variable mapping, which we explain below, and is summarised in Algorithm 1. Given the name of the attack (*attackName* in Algorithm 1) and all strategic rules used in *ABR* (*allStrategicRules*), the algorithm returns two lists, one containing all derivable predicates (*derivable*), and one containing all the non-derivable predicates (*nonDerivable*). The output can be used as hints by the user to what evidence can be further provided to *ABR*.

Variable mapping The verbose execution result displayed only shows the 'derivable' or 'not derivable' predicates for one specific set of constants[2], or fully ground predicates. To understand how we classify the predicates as 'derivable' and 'not derivable', we go into details on how the Prolog variables are mapped from one body predicate of a rule to the next body predicate.

As shown in Algorithm 1, *ABR* will try to derive each body predicate, going from left to right[3], while keeping track of all the mappings from variable to constants (we refer to this map as the argument map). Before executing the query `prove([bodyPredicate(X1, ..., XN)],D)` for each body predicate `bodyPredicate(X1, ..., XN)`, we ground the predicate by applying the substitutions in the argument map, i.e., to replace `X1, ..., XN` with its corresponding constants.

The derivable and non-derivable predicates are generated in a very specific way, that might lead to unexpected results in some cases. In short, *ABR* will always display *only the first result* returned by the reasoner, and use the constants in that particular result in all further queries on other body predicates to its right. This means that the results returned by the verbose execution will change depending on the order of evidence given. We justify why we chose this approach later in this chapter, after going through an example to further illustrate this limitation.

Example We use the following strategic rule (i.e., take *allStrategicRules* to be the list containing only the one rule given below) and evidence (written in this specific order) throughout this example to explain how the variables are mapped:

$$\begin{aligned} isCulprit(X, A1) \leftarrow & malwareUsedInAttack(M1, A1), similar(M1, M2), \\ & malwareLinkedTo(M2, X), notFromBlackMarket(M1), \\ & notFromBlackMarket(M2). \end{aligned}$$

[2]In Prolog, variables in rules can be unified with constants. *Variables* are arguments starting with an upper-case character (e.g., Att, X) and *constants* are arguments starting with a lower-case character or in quotes (e.g., china, "malwareA").

[3]Since Prolog's internal unification mechanism proceeds from left to right [50], for efficiency purposes, the rules in *ABR* are written such that the variables are ground from left to right. Hence, in verbose execution, we also map the arguments from left to right.

Algorithm 1 Verbose execution

```
1: procedure VERBOSEEXECUTE($attackName, allStrategicRules$)
2:     $derivable \leftarrow \{\}$
3:     $nonDerivable \leftarrow \{\}$
4:     for $rule \in allStrategicRules$ do
5:         $bodies \leftarrow getBodiesFromRule(rule)$    ▷ The variables “Att”, “A” and “A1” are reserved for the attack name
6:         $argumentMap \leftarrow \{“Att” = attackName, “A'' = attackName, “A1'' = attackName\}$
7:         for $body \in bodies$ do
8:             $body \leftarrow replaceVarWithConst(argumentMap, body)$
9:             $map \leftarrow executeQueryString(body)$
10:            $vars \leftarrow getAllVariables(body)$
11:            if $m = null$ then
12:                $nonDerivable \leftarrow m + nonDerivable$
13:            else
14:                for $var \in vars$ do
15:                    $constant \leftarrow m.get(var)$
16:                    $argumentMap.put(var, constant)$
17:                    $groundBody \leftarrow replaceVarWithConst(argumentMap, body)$
18:                end for
19:            end if
20:        end for
21:    end forreturn $derivable, nonDerivable$
22: end procedure
```

$$malwareUsedInAttack(“malwareA”, “attackName”).$$
$$similar(“malwareA”, “malwareB”).$$
$$similar(“malwareA”, “malwareC”).$$
$$malwareLinkedTo(“malwareC”, “someoneElse”).$$

When executing in verbose mode for the attack “*attackName*”, *ABR* will try to prove the body predicates from left to right. The first predicate that *ABR* attempts to prove is $malwareUsedInAttack(M1, “attackName”)$. The predicate is derivable, only one result (`M1 = ‘‘malwareA’’`) is returned, which is added to the argument map. Then, we try to derive $similar(“malwareA'', M2)$. The first result returned will be `M2 = ‘‘malwareB’’` because the evidence $similar(“malwareA”, “malwareB”)$ was defined above $similar(“malwareA”, “malwareC”)$. At this point, the argument map, list of derivable and non-derivable predicates will contain the following:

```
argumentMap: {A1="attackName", M1="malwareA", M2="malwareB"}
derivable: {malwareUsedInAttack("malwareA", "attackName"),
 ↪  similar("malwareA", "malwareB")}
nonDerivable: {}
```

ABR then tries to derive the predicate $malwareLinkedTo(“malwareB”, X)$, which will fail. Similarly, the predicates $notFromBlackMarket(“malwareA”)$ and

$notFromBlackMarket(\text{“}malwareB\text{”})$ will also fail. The final state of the variables (argument map, derivable and non-derivable predicates) will be:

```
argumentMap: {A1="attackName", M1="malwareA", M2="malwareB"}
derivable: {malwareUsedInAttack("malwareA", "attackName"),
↪  similar("malwareA", "malwareB")}
nonDerivable: {malwareLinkedTo("malwareB", X),
↪  notFromBlackMarket("malwareA"), notFromBlackMarket("malwareB")}
```

Limitation of verbose execution Looking back at the evidences used in this example,

$$\begin{aligned}
&malwareUsedInAttack(\text{“}malwareA\text{”}, \text{“}attackName\text{”}).\\
&similar(\text{“}malwareA\text{”}, \text{“}malwareB\text{”}).\\
&similar(\text{“}malwareA\text{”}, \text{“}malwareC\text{”}).\\
&malwareLinkedTo(\text{“}malwareC\text{”}, \text{“}someoneElse\text{”}).
\end{aligned}$$

it is clear that the following solution should also be possible.

```
argumentMap: {A1="attackName", M1="malwareA", M2="malwareC"}
derivable: {malwareUsedInAttack("malwareA", "attackName"),
↪  similar("malwareA", "malwareC"), malwareLinkedTo("malwareC",
↪  "someoneElse")}
nonDerivable: {notFromBlackMarket("malwareA"),
↪  notFromBlackMarket("malwareC")}
```

One might even say that such a solution is “better” since it has more derivable predicates than the one produced by *ABR* (3 instead of 2). However, to be able to produce such a solution, we need to find all possible solutions for every combination of arguments for every predicate, and then choose the one with the most derivable predicates.

To evaluate the complexity of such an operation, let’s consider a dummy rule with 5 body predicates, each with 2 arguments:

$$H \leftarrow B_1(X_1, X_2), B_2(X_2, X_3), B_3(X_3, X_4), B_4(X_4, X_5), B_5(X_5, X_6).$$

Take for example we have 3 solutions to each body predicate, each with a different second argument (e.g., $B_1(a, b), B_1(a, c), B_1(a, d)$ are all true). Then, using the method used in *ABR*, we have 5 executions (one for each of the 5 body predicates). In comparison, if we were to perform an exhaustive execution, we will have $1 + 3 + 3^2 + 3^3 + 3^4 = 364$ executions. Such an operation will be *exponential time*, in comparison to the *linear time* approach used by *ABR*. Therefore, we decide to keep to the original approach which only considers the first solution, as the exhaustive approach is clearly extremely time-consuming and is not scalable if there are many evidences given, which makes it impractical for a real-time application such as *ABR*.

7.3.2 Result.java

The Result class is the data structure that holds all the relevant information of the result of a query execution. This is also where *ABR* performs filtering for user-specified strategic rule preferences in the case when the culprits are different. We explain how the filtering is performed in this section.

First, to clarify what we mean by user-specified strategic rule, we show an example of a user-specified strategic rule preference and its corresponding strategic rules below.

```
rule(pref_rule, prefer(str_rule1(X, Att), str_rule2(Y, Att)), []) :-
 ↪  X \= Y.

rule(str_rule1(X,Att), isCulprit(X,Att),
 ↪  [existingGroupClaimedResponsibility(X,Att)]).
rule(str_rule2(C,Att), isCulprit(C,Att),
 ↪  [hasMotive(C,Att),hasCapability(C,Att)]).
```

Differing from most preferences used in ABR, this preference is a *dynamic preference* that will only be used if `X \= Y`, i.e., we are able to attribute the attack to different culprits. Due to how Gorgias was implemented (see Appendix C for details), this was not implemented in Prolog by adding preference rules as usual, but in Java by filtering the results before it is shown to the user in the GUI.

Use of Java to filter results

The results returned from the standard execution are filtered according to the strategic rule preferences. The filtering algorithm is shown in Algorithm 2, which consists of three main steps:

1. **Construct non-preferred rules set:** this set ($rulesToRemove$) contains all the non-preferred rules to be removed. For example, given the preference rule $prefer(rule1, rule2)$, $rule2$ will be in the set if one of the derivations uses $rule1$. The preference rule will only be used if both $rule1$ and $rule2$ are used in at least one of the derivations. Then, all derivations that uses $rule2$ will not be shown. Otherwise, the preference rule will not be used and all the derivations are shown as they are.

2. **Filter derivations:** the derivations are filtered by removing any derivation containing the non-preferred rules from the previous step.

3. **Re-process summary information:** the filtered derivations are re-processed to get the new list of culprits, maximum scores, and number of derivations.

Algorithm 2 Strategic rule filtering

```
\\ The variables $derivations$ and $trees$ are fields of the Result class
procedure FILTERBYSTRRULEPREFS($strRulePrefs$)
    $rulesToRemove \leftarrow \{\}$
    \\ 1. populate rulesToRemove
    for $pref \in strRulePrefs$ do
        $(preferredRule, nonpreferredRule) \leftarrow pref$
        for $derivation \in derivations$ do
            if $derivation.contains(preferredRule)$ then
                $rulesToRemove.add(nonpreferredRule)$
                $break$
            end if
        end for
    end for
    \\ 2. filter derivations
    for $i \leftarrow 0, len(derivations)$ do
        $derivation \leftarrow derivations[i]$
        $found \leftarrow false$
        for $rule \in rulesToRemove$ do
            if $derivation.contains(rule)$ then
                $found \leftarrow true$
                $break$
            end if
        end for
        if $\neg found$ then
            $filteredTrees.add(trees[i])$
            $filteredDerivations.add(derivation)$
        end if
    end for
    \\ 3. Reprocess summary of derivation
    \\ (function is pretty straightforward so its implementation is omitted here)
    $reprocessSummary(derivations)$
end procedure
```

This algorithm results in a set of filtered solutions, which, from the user's point of view in the GUI, works in the same way as the other preferences used in *ABR* that were added in Prolog. This allows users to express preferences for derivations generated using specific strategic rules over others, e.g., a user might want to prefer solutions where the culprit have capability and motive to perform the attack, over other solutions where the culprit claimed responsibility for the attack.

7.3.3 Utils.java

The Utils class contains String constants, such as file names and rule names, and utility methods that help us to parse the rules into rule name, head and body predicates. Rule parsing is mostly done with Regex and the standard Java String utility functions such as `substring` and `indexOf`.

7.3.4 DerivationNode.java

DerivationNode is the data structure used to generate graph diagrams visualisations for both the derivation and the argument tree. *ABR* creates a graph composed of DerivationNodes from both of the structures (derivation and argument tree), before using the `java-graphviz` package to visualise the generated graph. In this section we discuss the details of how the DerivationNode graph is constructed from both the derivation string and argument tree. The motivation for visualising derivation and argument tree is discussed in Section 5.3.

Derivation

We start by considering the structure of a derivation string, then describing the graph construction process in detail.

Structure of derivation First, let us look at what is a derivation. A derivation is a `List<Term>` (we call the method with the actual JPL Term instead of the String version used in Results.java). A derivation from Gorgias has the following form, as shown in Listing 4.

Listing 4 Example derivation string

```
[p4a_t(), bg1(), case_example5_f2(), case_example5_f3(),
 ↪  r_t_srcIP1(yourCountry,example5),
 ↪  r_t_attackOrigin(yourCountry,example5),
 ↪  r_str__loc(yourCountry,example5)]
```

A derivation can be represented by a root DerivationNode (conclusion) with the following key attributes:

- String result (head of rule)
- String rulename
- List<DerivationNode> childNode

The derivation returned by Gorgias is structured in a specific order. For any rule, the rules that proves its body predicates always appears on its left. Taking the example mentioned previously, if we look at the rule `r_t_attackOrigin(yourCountry, example5)`, its body predicate is `attackPossibleOrigin(yourCountry, example5)`, which is found in the rule `r_t_srcIP1(yourCountry, example5)`:

```
% attackPossibleOrigin/2 is in body
rule(r_t_attackOrigin(X,Att), attackOrigin(X,Att),
 ↪  [attackPossibleOrigin(X,Att)]).
% attackPossibleOrigin/2 is in head
rule(r_t_srcIP1(X,Att), attackPossibleOrigin(X,Att),
 ↪  [attackSourceIP(IP,Att),ipGeoloc(X,IP)]).
```

And we can confirm that the rule `r_t_srcIP1(yourCountry, example5)` appears on the left of `r_t_attackOrigin(yourCountry, example5)`.

```
[p4a_t(), bg1(), case_example5_f2(), case_example5_f3(),
↪  r_t_srcIP1(yourCountry, example5), r_t_attackOrigin(yourCountry,
↪  example5), r_str__loc(yourCountry, example5)]
```

Creation of DerivationNode from derivation Next, we explain how we create a DerivationNode from a derivation. The general idea is to use a stack of DerivationNode, and iteratively pop DerivationNode which contain the body predicates of the parent rule off from the stack. The main steps of the `createDerivationNode` method are (for each Term in the derivation):

1. if Term is a preference (starts with "p_"), continue;
2. else if Term is an abducible (starts with "ass(") or an evidence (does not start with "r_"), create DerivationNode and push onto stack;
3. else if Term is a rule, run `processRule(rulename, args, stack)`.

The main steps of `processRule` are outlined below:

1. DerivationNode currNode = new DerivationNode(...);
2. Iterate through stack;
   (a) DerivationNode n = st.pop();
   (b) List<String> bodyList = getBody(rulename) // list of body predicates of rule;
   (c) if (bodyList.contains(getHead(n), args)), currNode.add(n).

An example of illustrating how the example works can be found in Appendix D.3. The final figure generated from the derivation string in Listing 4 is shown in Figure 7.2.

Argument tree

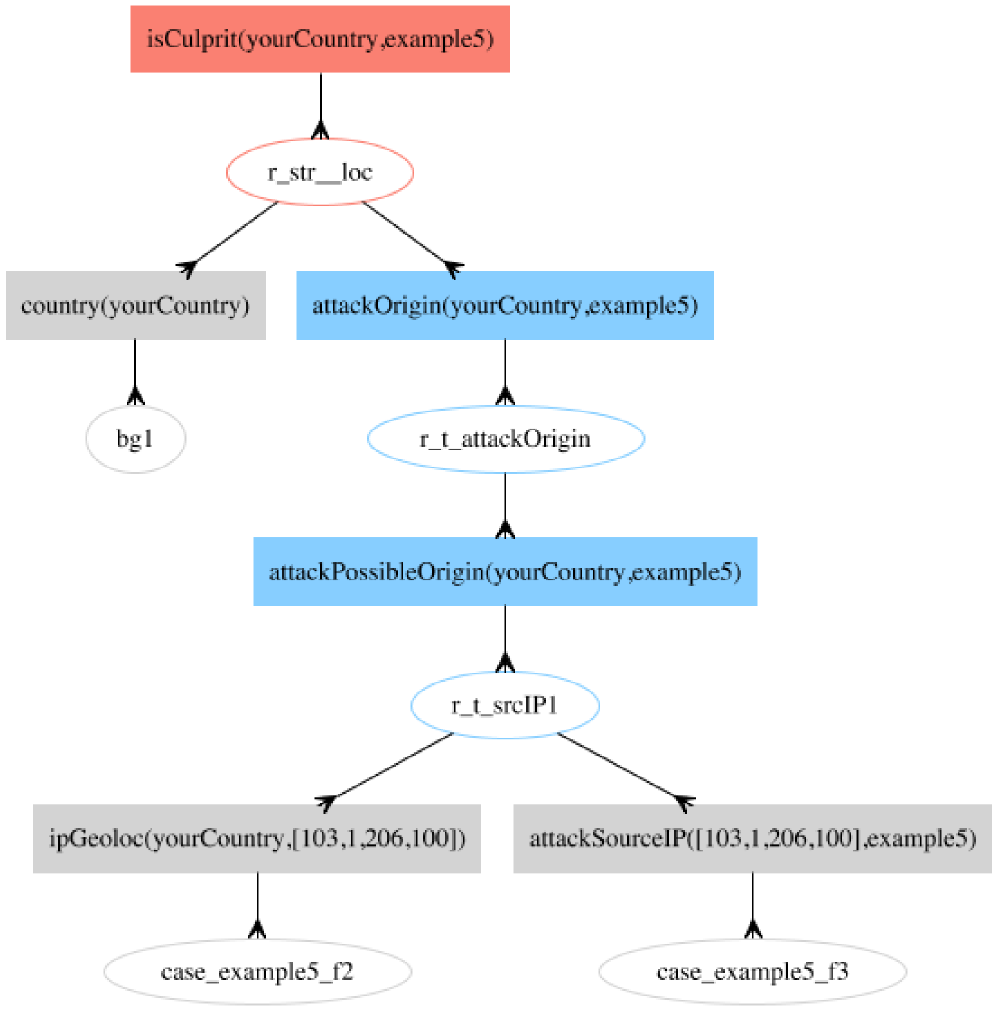
The argument tree generated by Gorgias also uses DerivationNode. Below, we briefly explain the structure of the argument tree and how the *ABR* generates a DerivationNode from an argument tree.

Structure of an argument tree An argument tree is a `String` printed to standard output by Gorgias' `visual_prove/2` predicate. An argument tree shows defense and attack nodes from the top conclusion node. Defense and attack nodes are made up of conflicting arguments. We show an argument tree in Listing 5. Argument trees can also be represented by a root node (conclusion node) with the following attributes:

- String derivation;
- int level;
- int defenseOrAttackNode;
- List<Node> childNode.

The level of a node determines which parent node and child node it is connected to.

Figure 7.2: Graph visualisation generated from example derivation string


isCulprit(yourCountry,example5)
r_str__loc
country(yourCountry)
attackOrigin(yourCountry,example5)
bg1
r_t_attackOrigin
attackPossibleOrigin(yourCountry,example5)
r_t_srcIP1
ipGeoloc(yourCountry,[103,1,206,100])
attackSourceIP([103,1,206,100],example5)
case_example5_f2
case_example5_f3

Creation of DerivationNode from argument tree We can extract the derivation by splitting the string on newlines and whether the node is defense or attack by checking if the line contains ''`{DEFENSE}`''. The level of the node can be obtained by `level=line.indexOf('[')/4`, since the level is denoted by the depth of the ''`|___[`'' branch in the String, as seen in Listing 5.

Lastly, we need to connect the nodes to their parents. To perform this operation, we need to know the level of every node. If we have a node with $level = n$, its children nodes are all the nodes below it that has $level = n + 1$. Take for example the argument tree shown in Listing 5, the children of the node on line 1 are the nodes on line 2 and line 4.

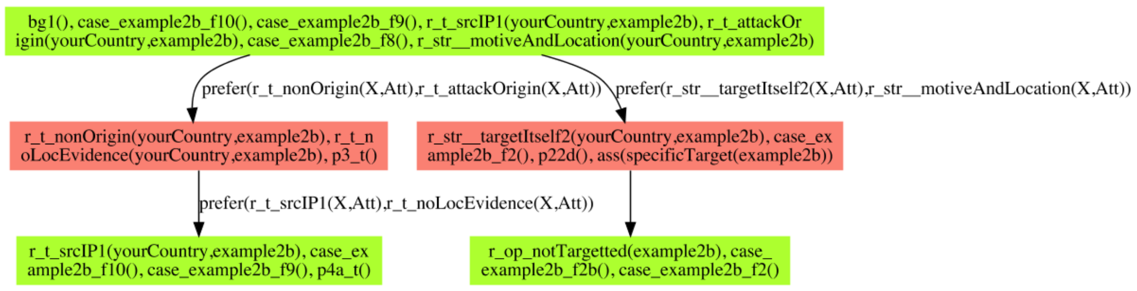
Creation of a graph from an argument tree is done using a stack, similar to the creation of a graph from a derivation mentioned previously. The graph generated from the argument tree shown above in Listing 5 is shown in Figure 7.3. An example illustrating how the node is created from an argument tree can be found in Appendix D.4.

Although the derivation diagram and argument tree are different, they share many commonalities, so to reduce code duplication, we decided to implement both of them using DerivationNode. When used for argument tree, the rulename field of a DerivationNode is an empty string.

Listing 5 Example argument tree

```
[bg1(), case_example2b_f10(), case_example2b_f9(),
 ↪  r_t_srcIP1(yourCountry,example2b), r_t_attackOrigin(yourCountry,example2b),
 ↪  r_str__loc(yourCountry,example2b)]  {DEFENSE}
|___[r_t_nonOrigin(yourCountry,example2b),
 ↪  r_t_noLocEvidence(yourCountry,example2b), p3_t()]
|   |___[r_t_srcIP1(yourCountry,example2b), case_example2b_f10(),
 ↪  case_example2b_f9(), p4a_t()]  {DEFENSE}
|___[r_str__targetItself2(yourCountry,example2b), case_example2b_f2(), p22e(),
 ↪  ass(specificTarget(example2b))]
    |___[r_op_notTargetted(example2b), case_example2b_f2b(), case_example2b_f2()]
     ↪  {DEFENSE}
```

Figure 7.3: Graph visualisation of argument tree generated from example


bg1(), case_example2b_f10(), case_example2b_f9(), r_t_srcIP1(yourCountry,example2b), r_t_attackOrigin(yourCountry,example2b), case_example2b_f8(), r_str__motiveAndLocation(yourCountry,example2b)
prefer(r_t_nonOrigin(X,Att),r_t_attackOrigin(X,Att))
prefer(r_str__targetItself2(X,Att),r_str__motiveAndLocation(X,Att))
r_t_nonOrigin(yourCountry,example2b), r_t_noLocEvidence(yourCountry,example2b), p3_t()
r_str__targetItself2(yourCountry,example2b), case_example2b_f2(), p22d(), ass(specificTarget(example2b))
prefer(r_t_srcIP1(X,Att),r_t_noLocEvidence(X,Att))
r_t_srcIP1(yourCountry,example2b), case_example2b_f10(), case_example2b_f9(), p4a_t()
r_op_notTargetted(example2b), case_example2b_f2b(), case_example2b_f2()

7.3.5 ToolIntegration.java

ToolIntegration is the helper class that contains all the methods relating to forensic tool integration in *ABR*. The implementation details of forensic tool integration are covered in Section 5.2.

Chapter 8

Evaluation

In this chapter, we give an evaluation of the usefulness of *ABR* from several perspectives. We present the core functionality of *ABR*, show how we evaluate the correctness and performance of *ABR* by testing different cyber attack scenarios, and finally, we discuss the scalability of this tool.

8.1 Functionality

ABR has several key features that can support the analyst in conducting investigation for attribution:

- Collate various different types of evidence: technical, social, historical data of other attacks.
- With sufficient evidences, attribute the attack to a specific group or country (standard execution).
- Show the reasoning behind the attribution by visualising the results of the derivation.
- Show other weaker counter arguments by visualising the argumentation tree.
- Show counter-arguments that are as strong as the winning argument, decide if one is preferred over the other.
- Show any abduced predicates used in the attribution process, giving users possible paths to investigate further in to prove the abduced predicates.
- Without sufficient evidence, show what are the other evidences that the analyst could potentially look for to make an attribution (verbose execution).
- Integration with forensic tools for the convenience of the analyst.

Iterative process These wide array of functionality are designed to make attribution an iterative process. Attribution is not a single operation where the analyst have all the required evidence and is able to immediately provide the solution. It involves many rounds of alternative between connecting the dots between evidences and collecting more evidence.

When executing in standard mode, by showing all the abduced predicates, *ABR* shows the assumptions made during reasoning process, prompting users to look for more evidence to validate the assumption. Displaying negative derivations for each positive one shows the possible counter arguments, encouraging the user to reconsider the attribution. On the other hand, executing in verbose mode gives users an overview of possible evidences to gather so that an attribution can be made.

Visualise reasoning behind attribution For each attribution result, *ABR* can give a breakdown of how the result came about, showing the breakdown of rules and evidences used. This visualisation can be understood by even non-professionals, giving the attribution result more transparency. As the attribution results can be easily comprehended, it will also be more credible and easily accepted.

Usability Since *ABR* has so many different functions, knowing how to use and interpret the results from *ABR* will be a significant obstacle. Given that *ABR* is built on argumentation, it would be good if users are familiar with logical programming languages like Prolog.

8.2 Evaluation of correctness of *ABR*

In this section we present the results of the tests we have performed and its implications on the correctness of *ABR*. To check the correctness of *ABR* and confirm that the rules and preferences works as expected, especially when dealing with conflicting information, we use synthesized scenarios, that are described in Section 8.2.1. To check that the integrations with forensic tools are correct, we test some synthesized examples that uses forensic tools as well, details of these examples are given in Section 8.2.1.

The test results and coverage are summarised in Tables 8.1 and 8.2. Many rules are used in the attribution process of each example, but we focus on just the key rules and preferences (mentioned in Section 8.2.1) that led to the expected solution. These are shown in the columns titled 'Rules tested' and 'Preferences tested'[1]. *ABR* returns the expected results for all the test scenarios; confirming the correctness of *ABR*.

8.2.1 Details of tests

We now explain the simple scenarios that were used to test the correctness and performance of *ABR* in detail, also presenting the results generated by *ABR*. We first present some stand-alone test scenarios, then some scenarios that were created to test the integration with forensic tools in *ABR*.

[1]Only the rule names are showed in the tables for simplicity, the full definition of the rules can be found in Appendix A.2

Table 8.2: Summary of forensic tool integration test cases and corresponding rules and preferences being tested

	Result	Tool tested	Rules tested	Preferences tested

autogeoloc_ex	hong_kong	ip2nation	r_t_srcIP1(X, Att)
			r_str__loc(C, Att)
				-

tor_ex	-	Bulk Tor Exit Node Exporter	r_t_spoofIPtor(IP)
			r_t_spoofIP(X, Att)
				p9a_t()

virustotal_ex	united_states, countryY	Virustotal	r_t_IPdomain1(S, M)
			r_t_similar(M1, M2)
			r_t_simCC1(M1, M2)
				-

Table 8.1: Summary of test cases and corresponding rules and preferences being tested

	Result	Rules tested	Preferences tested

Example 1	countryX, randomGroup	r_str__claimedResp(X, Att)
		r_str__linkedMalware(X, Att)
		r_str__motiveAndCapability(X, Att)
		r_t_similar1(M1, M2)
		r_op_hasCapability1(X, Att)
			p23d()

Example 1b	randomGroup	r_str__targetItself1(X, Att)	p21b(), p21g()
Example 1c	countryX, randomGroup	r_op_notTargetted(Att)	-

Example 2	fancyBear, russian_federation	r_str__aptGroupMotive(X, Att)
		r_str__motiveAndCapability(X, Att)
		r_str__claimedResp(X, Att)
			-

Example 2b	fancyBear	r_str__weakAttack(X, A)	p19(), p20()

Simple test scenarios

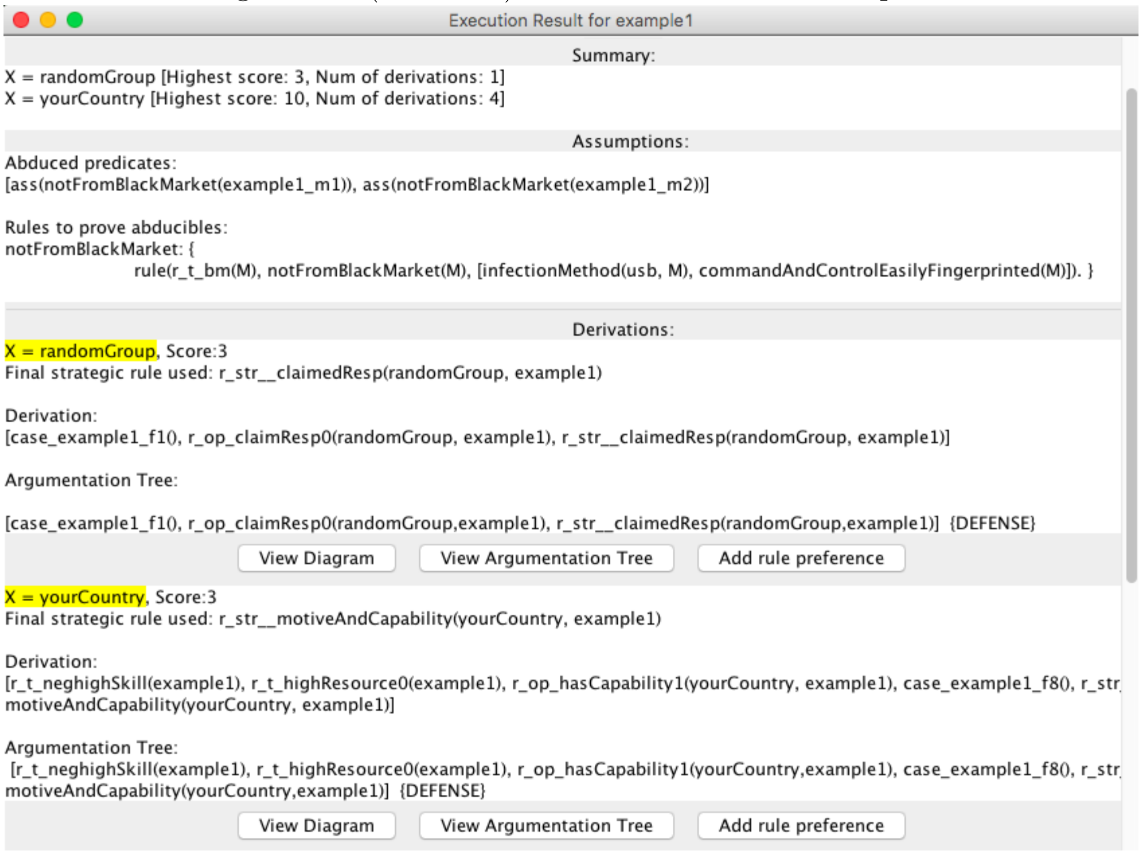
Example 1 We show the relevant evidences[2] in Listing 6, the attack name used is ‘example1’.

Listing 6 Evidence for example1

```
%% example1
%% expected:
%% randomGroup (claimResp)
%% countryX (motive and location,linkedMalware)
claimedResponsibility(randomGroup, example1).
malwareUsedInAttack(example1_m1, example1).
simlarCodeObfuscation(example1_m1, example1_m2).
malwareLinkedTo(example1_m2, countryX).
hasMotive(countryX, example1).
attackSourceIP([123,123,123,102], example1).
ipGeoloc(countryX, [123,123,123,102]).
```

The execution result for example 1 using *ABR* is shown in Figure 8.1. It corresponds to our expected result, with `randomGroup` and `countryX` as culprits.

Figure 8.1: (Standard) Execution result for example1



```
Summary:
X = randomGroup [Highest score: 3, Num of derivations: 1]
X = yourCountry [Highest score: 10, Num of derivations: 4]

Assumptions:
Abduced predicates:
[ass(notFromBlackMarket(example1_m1)), ass(notFromBlackMarket(example1_m2))]

Rules to prove abducibles:
notFromBlackMarket: {
        rule(r_t_bm(M), notFromBlackMarket(M), [infectionMethod(usb, M), commandAndControlEasilyFingerprinted(M)]). }

Derivations:
X = randomGroup, Score:3
Final strategic rule used: r_str__claimedResp(randomGroup, example1)

Derivation:
[case_example1_f1(), r_op_claimResp0(randomGroup, example1), r_str__claimedResp(randomGroup, example1)]

Argumentation Tree:

[case_example1_f1(), r_op_claimResp0(randomGroup,example1), r_str__claimedResp(randomGroup,example1)] {DEFENSE}

X = yourCountry, Score:3
Final strategic rule used: r_str__motiveAndCapability(yourCountry, example1)

Derivation:
[r_t_neghighSkill(example1), r_t_highResource0(example1), r_op_hasCapability1(yourCountry, example1), case_example1_f8(), r_str
motiveAndCapability(yourCountry, example1)]

Argumentation Tree:
 [r_t_neghighSkill(example1), r_t_highResource0(example1), r_op_hasCapability1(yourCountry,example1), case_example1_f8(), r_str
motiveAndCapability(yourCountry,example1)] {DEFENSE}
```

[2]The evidences in this section are presented in Prolog style for presentation purposes.

Example 1b Example 1b is the same as example 1, except the addition of the evidence `targetCountry(countryX, example1b)`, as shown in Listing 7, which means that countryX is the target of the attack example1b. For this example, we only expect there to be one culprit, `randomGroup`, as the following rule and preference, shown in Listing 8, will suppress the solution of `countryX` (since `specificTarget/1` is an abducible, the dynamic preference will be fired).

Listing 7 Evidence for example1b

```
%% example1b
%% expected:
%% randomGroup (claimResp)
claimedResponsibility(randomGroup, example1b).
malwareUsedInAttack(example1b_m1, example1b).
simlarCodeObfuscation(example1b_m1, example1b_m2).
malwareLinkedTo(example1b_m2, countryX).
hasMotive(countryX, example1b).
attackSourceIP([123,123,123,102], example1b).
ipGeoloc(countryX,[123,123,123,102]).
targetCountry(countryX, example1b). % added evidence
```

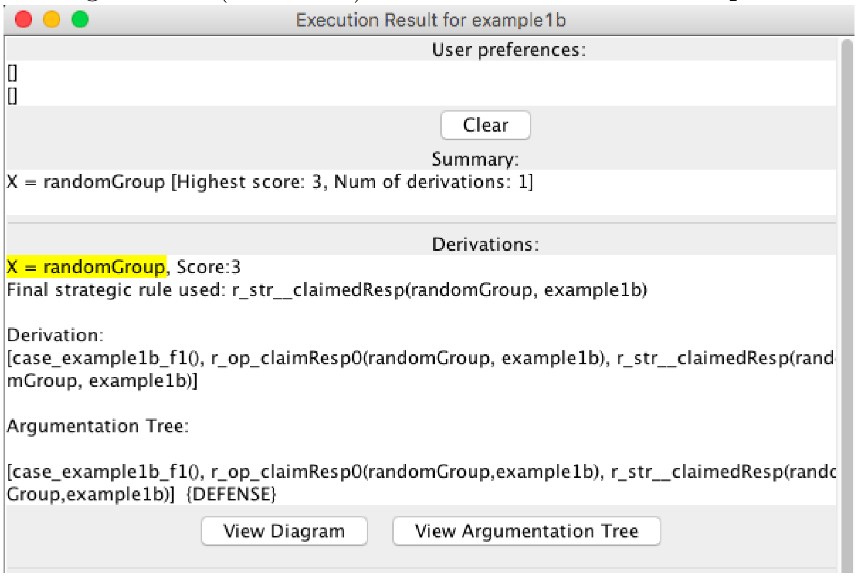
Listing 8 Relevant rules for example1b

```
rule(r_str__targetItself1(X, Att), neg(isCulprit(X, Att)),
 ↪  [target(X, Att)]).

% specificTarget(_Att) is abducible
abducible(specificTarget(_Att), []).

% dynamic preferences that will fire if specificTarget(Att) is true
rule(p21b(), prefer(r_str__targetItself1(X, Att),
 ↪  r_str__motiveAndCapability(X, Att)), [specificTarget(Att)]).
rule(p21g(), prefer(r_str__targetItself1(X, Att),
 ↪  r_str__linkedMalware(X, Att)), [specificTarget(Att)]).
```

The execution result for example 1b using *ABR* is shown in Figure 8.2. It corresponds to our expected result, with only `randomGroup` as culprit.

Example 1c Example 1c is the same as example 1b, but with one additional evidence `targetCountry(countryY, example1b)`, as shown in Listing 9, so both countryX and countryY are targets of the attack example1c. This additional evidence makes it possible to prove `neg(specificTarget(example1c))` using the rule in Listing 10.

Figure 8.2: (Standard) Execution result for example1b



```
[]
[]
X = randomGroup [Highest score: 3, Num of derivations: 1]

X = randomGroup, Score:3
Final strategic rule used: r_str__claimedResp(randomGroup, example1b)

Derivation:
[case_example1b_f1(), r_op_claimResp0(randomGroup, example1b), r_str__claimedResp(rand
mGroup, example1b)]

Argumentation Tree:

[case_example1b_f1(), r_op_claimResp0(randomGroup,example1b), r_str__claimedResp(randc
Group,example1b)]  {DEFENSE}
```

Listing 9 Evidence for example1c

```
%% example1c
%% expected:
%% randomGroup (claimResp)
%% countryX is not the only target, not specific attack, so countryX
↪  might be culprit
claimedResponsibility(randomGroup, example1c).
malwareUsedInAttack(example1c_m1, example1c).
simlarCodeObfuscation(example1c_m1, example1c_m2).
malwareLinkedTo(example1c_m2, countryX).
hasMotive(countryX, example1c).
attackSourceIP([123,123,123,102], example1c).
ipGeoloc(countryX,[123,123,123,102]).
targetCountry(countryX, example1c).
targetCountry(countryY, example1c). % added evidence
```

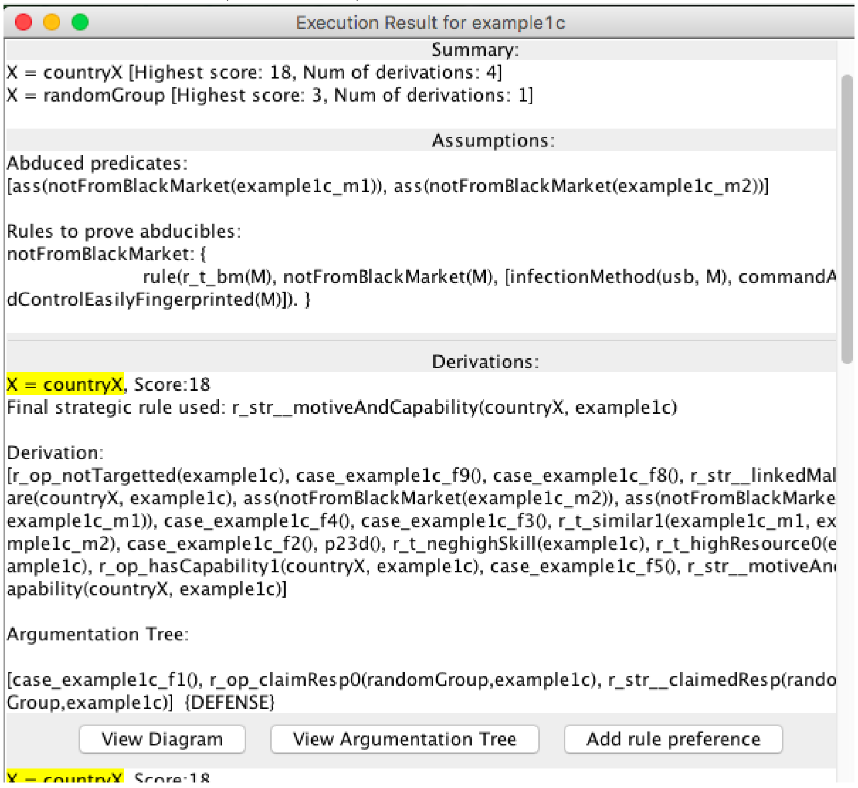
Listing 10 Relevant rules for example1c

```
% more than one country targetted
rule(r_op_notTargetted(Att), neg(specificTarget(Att)),
↪  [targetCountry(T1, Att), targetCountry(T2, Att), T1 \= T2]).
```

Since we have `neg(specificTarget(example1c))`, the dynamic preference used before will no longer fire since the condition (`specificTarget(Att)`) is false. As a result of this, we expect to be able to attribute example1c to `randomGroup` and `countryX` once again.

The execution result for example 1c using *ABR* is shown in Figure 8.3. It corresponds to our expected result, with both `randomGroup` and `countryX` as culprit (see 'Summary' section at top of Figure 8.3).

Figure 8.3: (Standard) Execution result for example1c



```
Execution Result for example1c
Summary:
X = countryX [Highest score: 18, Num of derivations: 4]
X = randomGroup [Highest score: 3, Num of derivations: 1]

Assumptions:
Abduced predicates:
[ass(notFromBlackMarket(example1c_m1)), ass(notFromBlackMarket(example1c_m2))]

Rules to prove abducibles:
notFromBlackMarket: {
                rule(r_t_bm(M), notFromBlackMarket(M), [infectionMethod(usb, M), commandA
dControlEasilyFingerprinted(M)]). }

Derivations:
X = countryX, Score:18
Final strategic rule used: r_str__motiveAndCapability(countryX, example1c)

Derivation:
[r_op_notTargetted(example1c), case_example1c_f9(), case_example1c_f8(), r_str__linkedMal
are(countryX, example1c), ass(notFromBlackMarket(example1c_m2)), ass(notFromBlackMarke
example1c_m1)), case_example1c_f4(), case_example1c_f3(), r_t_similar1(example1c_m1, ex
mple1c_m2), case_example1c_f2(), p23d(), r_t_neghighSkill(example1c), r_t_highResource0(e
ample1c), r_op_hasCapability1(countryX, example1c), case_example1c_f5(), r_str__motiveAn
apability(countryX, example1c)]

Argumentation Tree:

[case_example1c_f1(), r_op_claimResp0(randomGroup,example1c), r_str__claimedResp(rando
Group,example1c)]  {DEFENSE}
```

```
X = countryX, Score:18
```

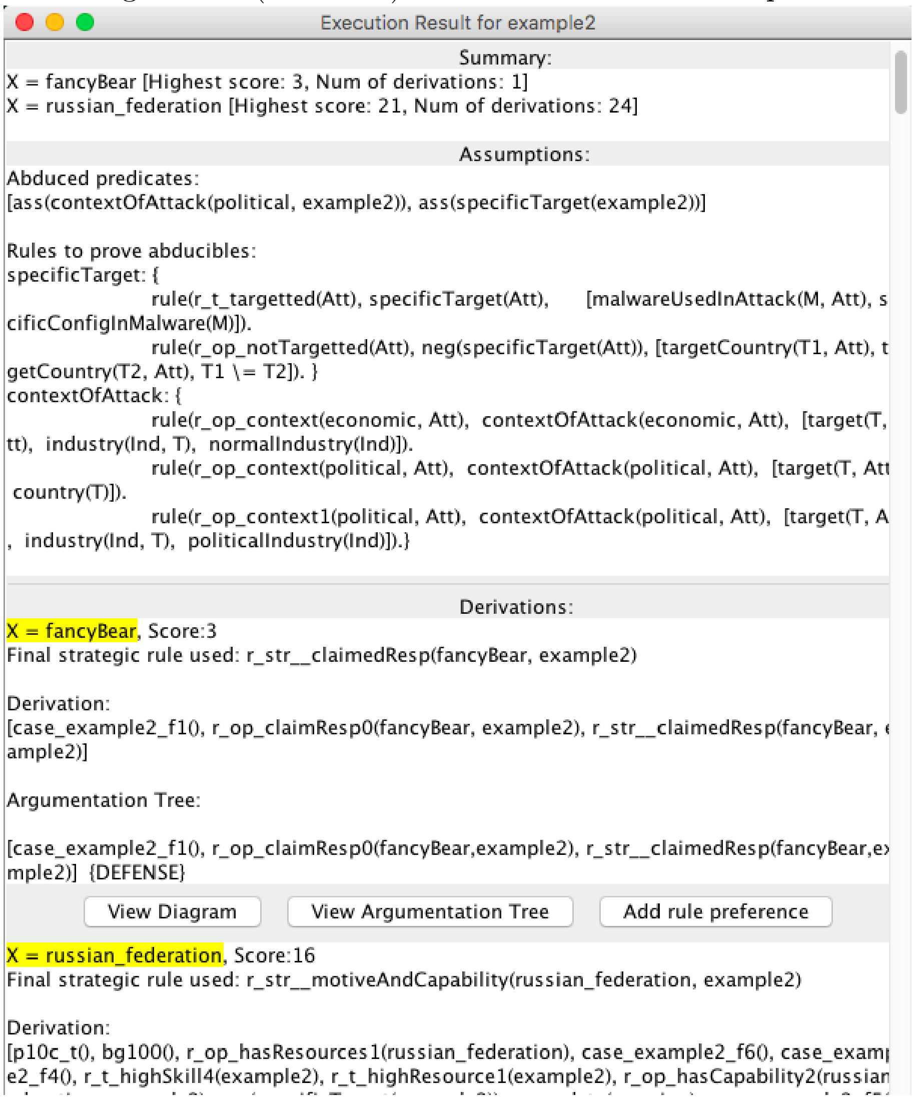
Example 2 We show below the relevant evidences for example2 in Listing 11.

Listing 11 Evidence for example2

```
%% example2
%% expected:
%% fancyBear (claimResp)
%% russian_federation (APT group link to origin country, has motive
↪  and capability)
claimedResponsibility(fancyBear,example2).
targetCountry(countryX,example2).
attackPeriod(example2, [2018,6]).
malwareUsedInAttack(example2_m1,example2).
imposedSanctions(countryX, russian_federation, ongoing).
sophisticatedMalware(example2_m1).
```

The execution result for example2 is shown in Figure 8.4. It corresponds to our expected result, with both `russian_federation` and `fancyBear` as culprit.

Figure 8.4: (Standard) Execution result for example2



```
Summary:
X = fancyBear [Highest score: 3, Num of derivations: 1]
X = russian_federation [Highest score: 21, Num of derivations: 24]

Assumptions:
Abduced predicates:
[ass(contextOfAttack(political, example2)), ass(specificTarget(example2))]

Rules to prove abducibles:
specificTarget: {
            rule(r_t_targetted(Att), specificTarget(Att),    [malwareUsedInAttack(M, Att), s
cificConfigInMalware(M)]).
            rule(r_op_notTargetted(Att), neg(specificTarget(Att)), [targetCountry(T1, Att), t
getCountry(T2, Att), T1 \= T2]). }
contextOfAttack: {
            rule(r_op_context(economic, Att),  contextOfAttack(economic, Att),  [target(T,
tt),  industry(Ind, T),  normalIndustry(Ind)]).
            rule(r_op_context(political, Att),  contextOfAttack(political, Att),  [target(T, Att
 country(T)]).
            rule(r_op_context1(political, Att),  contextOfAttack(political, Att),  [target(T, A
,  industry(Ind, T),  politicalIndustry(Ind)]).}

Derivations:
X = fancyBear, Score:3
Final strategic rule used: r_str__claimedResp(fancyBear, example2)

Derivation:
[case_example2_f1(), r_op_claimResp0(fancyBear, example2), r_str__claimedResp(fancyBear, e
ample2)]

Argumentation Tree:

[case_example2_f1(), r_op_claimResp0(fancyBear,example2), r_str__claimedResp(fancyBear,ex
mple2)]  {DEFENSE}

X = russian_federation, Score:16
Final strategic rule used: r_str__motiveAndCapability(russian_federation, example2)

Derivation:
[p10c_t(), bg100(), r_op_hasResources1(russian_federation), case_example2_f6(), case_exam
e2_f4(), r_t_highSkill4(example2), r_t_highResource1(example2), r_op_hasCapability2(russian
```

Example 2b Example 2b is modified from example 2 by removing the last evidence (`sophisticatedMalware(example2_m1)`), as shown in Listing 12. After removing this evidence, we are no longer able to attribute the attack to the Russian federation. This is due to the effects of the stronger preference on the 'weak attack' rule than the 'APT group motive' rule or the 'motive and capability' rule, shown in Listing 13. The reasoning behind the weak attack rule is that a group or country that has large amounts of resources is unlikely to preform a weak attack (that does not require high resources).

Listing 12 Evidence for example2b

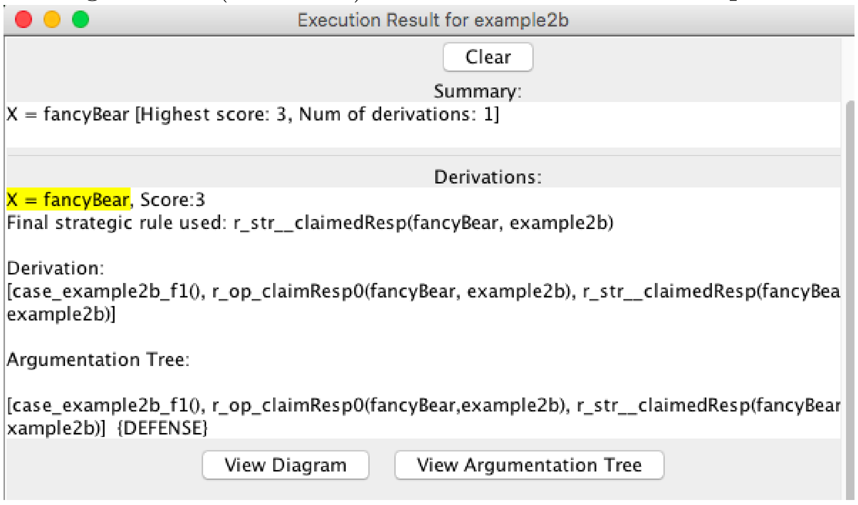
```
%% example2b
%% expected:
%% fancyBear (claimResp)
claimedResponsibility(fancyBear, example2b).
targetCountry(countryX, example2b).
attackPeriod(example2b, [2018,6]).
malwareUsedInAttack(example2b_m1, example2b).
imposedSanctions(countryX, russian_federation, ongoing).
% removed last evidence
```

Listing 13 Relevant rules for example2b

```
% weak attack rule
rule(r_str__weakAttack(X, Att), neg(isCulprit(X, Att)),
 ↪  [hasResources(X), neg(requireHighResource(Att))]).

% preferences
rule(p19(), prefer(r_str__weakAttack(X, A), r_str__aptGroupMotive(X,
 ↪  A)), []).
rule(p20(), prefer(r_str__weakAttack(X, A),
 ↪  r_str__motiveAndCapability(X, A)), []).
```

The execution result for example2b is shown in Figure 8.5. It corresponds to our expected result, with only `fancyBear` as culprit.

Figure 8.5: (Standard) Execution result for example2b



```
Summary:
X = fancyBear [Highest score: 3, Num of derivations: 1]

Derivations:
X = fancyBear, Score:3
Final strategic rule used: r_str__claimedResp(fancyBear, example2b)

Derivation:
[case_example2b_f1(), r_op_claimResp0(fancyBear, example2b), r_str__claimedResp(fancyBea
example2b)]

Argumentation Tree:

[case_example2b_f1(), r_op_claimResp0(fancyBear,example2b), r_str__claimedResp(fancyBear
xample2b)]  {DEFENSE}
```

Forensic tool test scenarios

Let us now show some of the examples we constructed to test the integration with forensic tools in *ABR*.

IP geolocation example We use the following example to test the correctness of the integration with the ip2nation database. The IP 103.234.220.195 should be geolocated to Hong Kong. Given the evidence `attackSourceIP([103,234,220,195], autogeoloc_ex)` and the auxiliary predicate `ip([103,234,220,195])`, as shown in Listing 14, we will have `hong_kong` as the culprit for the attack, according to the generated facts and rules in Listing 15.

Listing 14 Evidence for autogeoloc_ex

```
%% auto geolocation example
%% expected: hong_kong (loc)
attackSourceIP([103,234,220,195], autogeoloc_ex).
ip([103,234,220,195]).
```

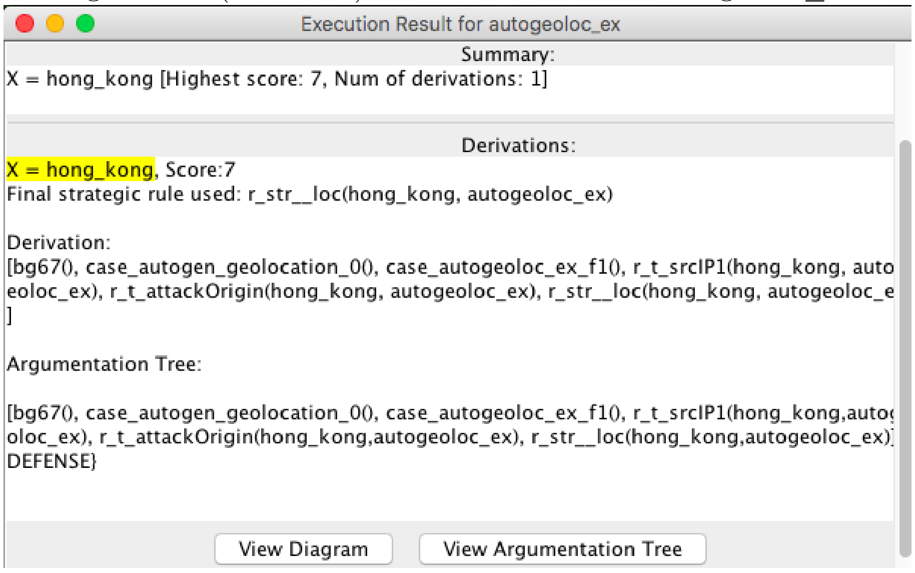
Listing 15 Relevant rules for autogeoloc_ex

```
% IP geolocation generated in automated_geolocation.pl
rule(case_autogen_geolocation_0(),
 ↪  ipGeoloc(hong_kong,[103,234,220,195]), []).

% from tech_rules.pl
rule(r_t_srcIP1(X, Att),   attackPossibleOrigin(X, Att),
 ↪  [attackSourceIP(IP, Att), ipGeoloc(X, IP)]).
rule(r_t_attackOrigin(X, Att), attackOrigin(X, Att),
 ↪  [attackPossibleOrigin(X, Att)]).

% from str_rules.pl
rule(r_str__loc(C, Att), isCulprit(C, Att), [attackOrigin(C, Att),
 ↪  country(C)]).
```

The execution result for autogeoloc_ex is shown in Figure 8.6. It corresponds to our expected result, with `hong_kong` as culprit.

Tor example We use the following example to test the correctness of the integration with the bulk Tor exit node exporter. We give the same evidence (with different attack name, now tor_ex instead of autogeoloc_ex), but we have an extra evidence, `targetServerIP([72,111,1,30], tor_ex)`, as shown in Listing 16. This will trigger the bulk Tor exit node exporter to export all Tor exit nodes that will reach the server at IP 72.111.1.30. All of those IPs will be marked as 'spoofed' since Tor

Figure 8.6: (Standard) Execution result for autogeoloc_ex



```
Execution Result for autogeoloc_ex
Summary:
X = hong_kong [Highest score: 7, Num of derivations: 1]

Derivations:
X = hong_kong, Score:7
Final strategic rule used: r_str__loc(hong_kong, autogeoloc_ex)

Derivation:
[bg67(), case_autogen_geolocation_0(), case_autogeoloc_ex_f1(), r_t_srcIP1(hong_kong, auto
eoloc_ex), r_t_attackOrigin(hong_kong, autogeoloc_ex), r_str__loc(hong_kong, autogeoloc_e
]

Argumentation Tree:

[bg67(), case_autogen_geolocation_0(), case_autogeoloc_ex_f1(), r_t_srcIP1(hong_kong,autog
oloc_ex), r_t_attackOrigin(hong_kong,autogeoloc_ex), r_str__loc(hong_kong,autogeoloc_ex)]
DEFENSE}
```

exit nodes are just the exit point of the Tor network, not where the traffic actually originated from. Since 103.234.220.195 is one of those Tor exit nodes for that can reach 72.111.1.30, it will be marked as spoofed, and so there will be no culprits for this example, according to the generated facts and rules shown in Listing 17.

Listing 16 Evidence for tor_ex

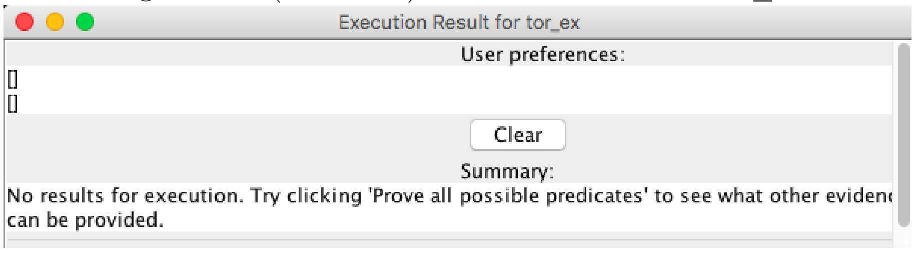
```
%% tor_ex
%% expected: no culprit (spoof)
attackSourceIP([103,234,220,195],tor_ex).
ip([103,234,220,195]).
targetServerIP([72,111,1,30], tor_ex).
```

Listing 17 Relevant rules for tor_ex

```
% torIP generated from bulk Tor exit node exporter
rule(case_torCheck1(), torIP([103,234,220,195], [72,111,1,30]), []).

% from tech_rules.pl
rule(r_t_spoofIPtor(IP), spoofedIP(IP, Att), [attackSourceIP(IP,
 ↪  Att), targetServerIP(TargetServerIP, Att), torIP(IP,
 ↪  TargetServerIP)]).
rule(r_t_spoofIP(X, Att), neg(attackPossibleOrigin(X, Att)),
 ↪  [attackSourceIP(IP, Att), spoofedIP(IP, Att), ipGeoloc(X, IP)]).
```

The execution result for tor_ex are shown in Figure 8.7. It corresponds to our expected result, with no result for the attribution.

Figure 8.7: (Standard) Execution result for tor_ex


Execution Result for tor_ex

User preferences:

[]
[]

Summary:

No results for execution. Try clicking 'Prove all possible predicates' to see what other eviden
can be provided.

Virustotal example This example is used to test the correctness of the integration with the Virustotal IP resolution. Given the evidence `ip([8,8,8,8], [2018,5]`, as shown in Listing 18, *ABR* will request for the resolution for IP 8.8.8.8 in 2018 May. This should return the domain '00027.hk'. Combined with the evidence `ccServer('00027.hk', example_past_attack_m)` and `malwareLinkedTo(example_past_attack_m, countryY)`, as shown in Listing 19, we will then be able to prove that countryY is a culprit.

Listing 18 Evidence for virustotal_ex

```
%% expected:
%% united_states (loc)
%% countryY (linkedMalware)
malwareUsedInAttack(virustotal_ex_malware, virustotal_ex).
attackSourceIP([8,8,8,8], virustotal_ex).
ip([8,8,8,8], [2018,5]). % ip([IP],[YYYY,MM]) for auto resolution
 ↪  using virustotal
ccServer('00027.hk', example_past_attack_m).
malwareLinkedTo(example_past_attack_m, countryY).
```

The execution result for virustotal_ex is shown in Figure 8.8. It corresponds to our expected result, with `united_states` and `countryY` as culprits.

8.3 Evaluation of performance of *ABR*

In this section we show the performance results obtained by running the previous test scenarios, as well as past cyber attack cases. Additionally, we performed tests to analyse the performance of *ABR* when we use an increasing number of evidences during the attribution.

8.3.1 Analysis of results

The performances of the test scenarios are summarised in Tables 8.3 and 8.4, while the performances of some real cases of cyber attacks are summarised in Table 8.5. By looking at the raw data of total runtime, we notice that there is a large disparity between different test cases, ranging from 0.4s to 20s. We now discuss what might have cause this disparity.

Listing 19 Relevant rules for virustotal_ex

```
% IP resolution generated by virustotal
rule(case_virustotal_res0(), ipResolution('00027.hk', [8,8,8,8],
↪  [2018,5]), []).

% from tech_rules.pl
rule(r_t_IPdomain1(S, M),  ccServer(S, M), [malwareUsedInAttack(M,
↪  Att), attackSourceIP(IP, Att), ipResolution(S, IP, _D)]).

rule(r_t_similar(M1, M2), similar(M1, M2), [similarCCServer(M1, M2),
↪  M1 \= M2]).
rule(r_t_simCC1(M1, M2), similarCCServer(M1, M2), [ccServer(S, M1),
↪  ccServer(S, M2)]).

% from str_rules.pl
rule(r_str__linkedMalware(X, A1), isCulprit(X, A1),
↪  [malwareUsedInAttack(M1, A1), similar(M1, M2),
↪  malwareLinkedTo(M2, X), notFromBlackMarket(M1),
↪  notFromBlackMarket(M2)]).
```

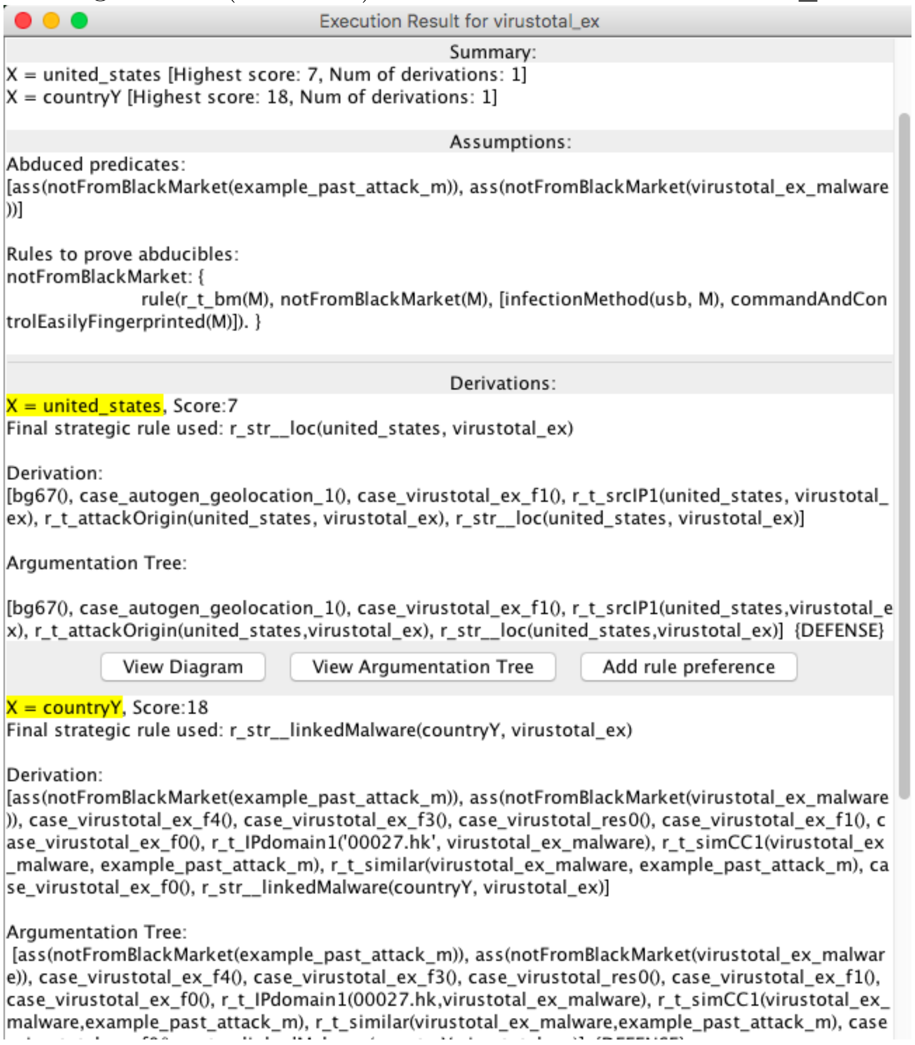
Table 8.3: Summary of performance of test cases (n = total number of derivations)

	Result	Number of unique rules fired	n	Runtime (s)	Avg runtime per derivation (s)
example1	countryX, randomGroup	8	2	0.7127	0.3563
example1b	randomGroup	2	1	0.4168	0.4168
example1c	countryX, randomGroup	8	4	2.270	0.5674
example2	fancyBear, russian_federation	12	24	20.97	0.8737
example2b	fancyBear	2	1	4.305	4.305

Number of derivations Our first hypothesis is that the time difference is due to number of derivations. A larger number of derivation will not only increase the time taken for Prolog to generate the solutions, but also more time spent on creating the derivation and argumentation diagrams. After computing the average runtime per derivation, we observe that the average time is much more consistent, only ranging from 4s to below 1s.

Number of unique rules fired There is only one outlier that runs for 4s on average per derivation (example2b). In attempt to explain this, we also computed the total number of unique rules used in all the derivations (‘Number of unique rules fired’ column). However, this justifies neither the difference in total runtime nor average runtime. For total runtime, example2b (4.3s) uses 2 unique rules but example1 (0.7s) uses 8 unique rules, and for average runtime, example2b (4.3s) only uses 2 unique rules, while example2 (0.8s) uses 12 unique rules. Comparing the results for all the other test cases, we can confirm that there is no direct correlation between number of rules fired and total and average runtime.

Figure 8.8: (Standard) Execution result for virustotal_ex



```
Summary:
X = united_states [Highest score: 7, Num of derivations: 1]
X = countryY [Highest score: 18, Num of derivations: 1]

Assumptions:
Abduced predicates:
[ass(notFromBlackMarket(example_past_attack_m)), ass(notFromBlackMarket(virustotal_ex_malware
))]

Rules to prove abducibles:
notFromBlackMarket: {
                    rule(r_t_bm(M), notFromBlackMarket(M), [infectionMethod(usb, M), commandAndCon
trolEasilyFingerprinted(M)]). }

Derivations:
X = united_states, Score:7
Final strategic rule used: r_str__loc(united_states, virustotal_ex)

Derivation:
[bg67(), case_autogen_geolocation_1(), case_virustotal_ex_f1(), r_t_srcIP1(united_states, virustotal_
ex), r_t_attackOrigin(united_states, virustotal_ex), r_str__loc(united_states, virustotal_ex)]

Argumentation Tree:

[bg67(), case_autogen_geolocation_1(), case_virustotal_ex_f1(), r_t_srcIP1(united_states,virustotal_e
x), r_t_attackOrigin(united_states,virustotal_ex), r_str__loc(united_states,virustotal_ex)]  {DEFENSE}

X = countryY, Score:18
Final strategic rule used: r_str__linkedMalware(countryY, virustotal_ex)

Derivation:
[ass(notFromBlackMarket(example_past_attack_m)), ass(notFromBlackMarket(virustotal_ex_malware
)), case_virustotal_ex_f4(), case_virustotal_ex_f3(), case_virustotal_res0(), case_virustotal_ex_f1(), c
ase_virustotal_ex_f0(), r_t_IPdomain1('00027.hk', virustotal_ex_malware), r_t_simCC1(virustotal_ex
_malware, example_past_attack_m), r_t_similar(virustotal_ex_malware, example_past_attack_m), ca
se_virustotal_ex_f0(), r_str__linkedMalware(countryY, virustotal_ex)]

Argumentation Tree:
 [ass(notFromBlackMarket(example_past_attack_m)), ass(notFromBlackMarket(virustotal_ex_malwar
e)), case_virustotal_ex_f4(), case_virustotal_ex_f3(), case_virustotal_res0(), case_virustotal_ex_f1(),
case_virustotal_ex_f0(), r_t_IPdomain1(00027.hk,virustotal_ex_malware), r_t_simCC1(virustotal_ex_
malware,example_past_attack_m), r_t_similar(virustotal_ex_malware,example_past_attack_m), case
```

8.3.2 Scaling with number of evidences

To investigate how *ABR* performs as the number of evidences increases, we carried out an experiment, using increasing number of `attackSourceIP(IP, Att)` and `ip(IP)` predicates.

Below we show an example of the list of evidences when $n = 10$:

```
rule(f0(),attackSourceIP([103,234,220,195], ex),[]).
rule(f0a(),ip([103,234,220,195], []).
rule(f1(),attackSourceIP([103,234,220,197], ex),[]).
rule(f1a(),ip([103,234,220,197], [])).
rule(f2(),attackSourceIP([103,236,201,110], ex),[]).
rule(f2a(),ip([103,236,201,110], [])).
rule(f3(),attackSourceIP([103,250,73,13], ex),[]).
rule(f3a(),ip([103,250,73,13], [])).
rule(f4(),attackSourceIP([103,27,124,82], ex),[]).
rule(f4a(),ip([103,27,124,82], [])).
rule(f5(),attackSourceIP([103,28,52,93], ex),[]).
rule(f5a(),ip([103,28,52,93], [])).
```

Table 8.4: Summary of performance of forensic tool integration test cases (n = total number of derivations)

	Result	Number of unique rules fired	n	Runtime (s)	Avg runtime per derivation (s)
autogeoloc_ex	hong_kong	3	1	0.719287352	0.719287352
tor_ex	-	-	1	0.257613654	0.257613654
virustotal_ex	united_states, countryY	7	2	1.760543103	0.8802715515

Table 8.5: Performance of *ABR* when executing case study attribution cases (n = total number of derivations)

	Result	Number of unique rules fired	n	Runtime (s)	Avg runtime per derivation (s)
apt1	china	8	7	2.826	0.4037
wannacryattack	lazarusGrp	2	1	0.1244	0.2407
gaussattack	equationGrp	6	18	2.370	0.1317
stuxnetattack	united_states, israel	13	6	4.640	0.7602
sonyhack	guardiansOfPeace, iran, north_korea	8	3	3.438	1.146
usbankhack	iran	9	8	2.867	0.3584

```
rule(f6(),attackSourceIP([103,28,53,138], ex),[]).
rule(f6a(),ip([103,28,53,138], [])).
rule(f7(),attackSourceIP([103,3,61,114], ex),[]).
rule(f7a(),ip([103,3,61,114], [])).
rule(f8(),attackSourceIP([103,8,79,229], ex),[]).
rule(f8a(),ip([103,8,79,229], [])).
rule(f9(),attackSourceIP([103,87,8,163], ex),[]).
rule(f9a(),ip([103,87,8,163], [])).
```

Each IP address specified in the `ip(IP)` predicate is automatically geolocated, and when we query `prove([isCulprit(X, ex)],D)`, each of the different countries corresponding to the IPs will show up as a culprit, using the following rule:

```
rule(r_str__loc(C, Att), isCulprit(C, Att), [attackOrigin(C, Att),
↪  country(C)]).
```

The results of the experiment are shown in Table 8.6. We observe that while both the Prolog execution time and the total runtime increases linearly up to $n = 100$ (Figure 8.9), after reaching $n = 100$, the increases in both seem to slow down significantly (Figure 8.10). This can be explained by the fact that we introduced a limit of 100 in the QueryExecutor class (see Appendix D.1). Since we only obtain the first 100 solutions, all the other IP addresses will be automatically geolocated when we preprocess the files, but *ABR* will not spend extra time proving more solutions from it.

Although this hard limit means that we might not show the full results when there are more than 100 solutions, but since *ABR* is a tool that works closely with human users, it is unlikely that the user can read and comprehend more than 100 different solutions. The user can reduce the number of solutions by adding custom preferences to disregard irrelevant solutions.

Figure 8.9: Part of execution time against number of evidences (n) graph, up to $n = 100$

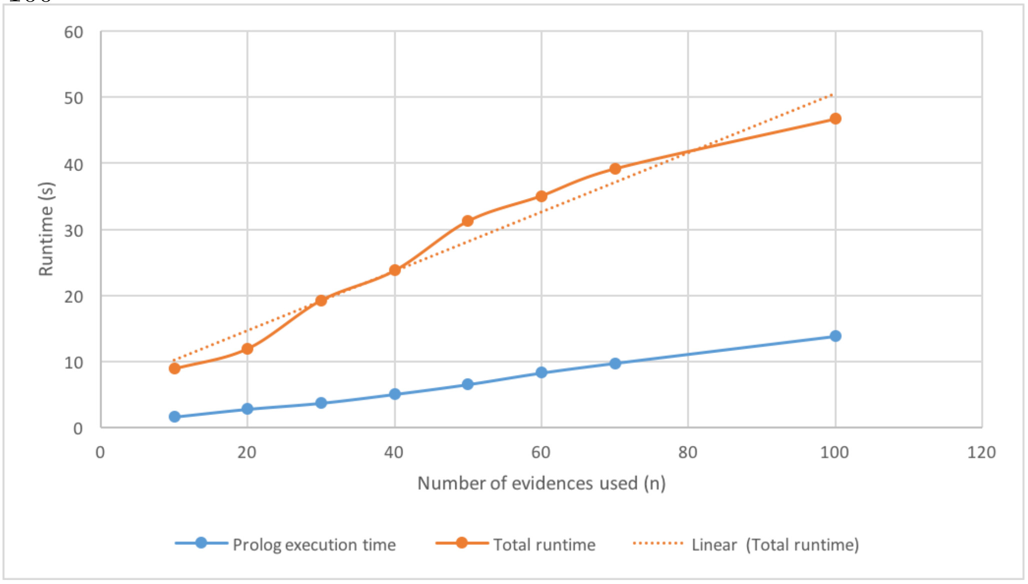


Table 8.6: Performance of *ABR* when executing example with increasing number of attack source IPs (n = number of different IPs used as evidence)

n	Prolog execution time (s)	Total runtime (s)
10	1.657	8.934
20	2.824	11.98
30	3.741	19.28
40	5.072	23.78
50	6.555	31.26
60	8.297	35.08
70	9.761	39.17
100	13.84	46.70
150	16.28	52.00
200	17.20	55.08
300	19.09	60.38

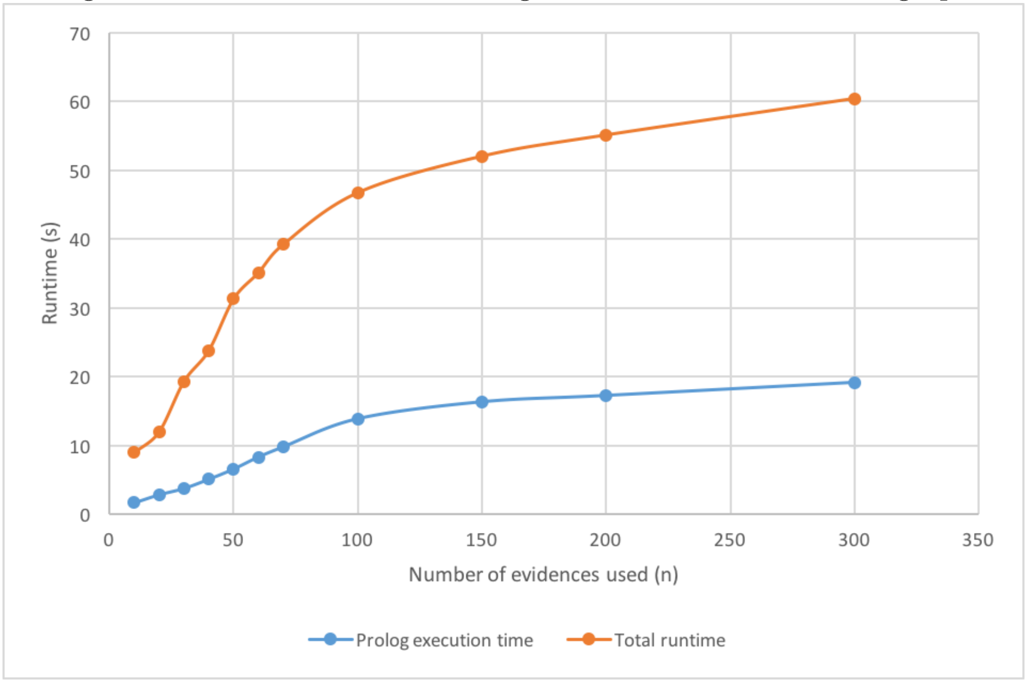
8.4 Discussion

Finally, we discuss some of the key limitations and strengths of *ABR*.

8.4.1 Limitations

Limitation of verbose execution As mentioned in Section 7.3.1, one limitation of the verbose execution of *ABR*, is that the overview generated only considers the first solution generated by the *ABR* reasoner. This means that *ABR* might underestimate the number of predicates that are derivable, giving the user a false impression. However, this might not be a big problem since the verbose execution is meant as a general overview of derivable predicates, not a detailed representation. When using *ABR*, users will likely run the verbose execution many times after adding additional evidences. Therefore, we have chosen to prioritise efficiency (which we explain later in this chapter) over producing the 'best' output every time.

Figure 8.10: Full execution time against number of evidences graph



Usability There is some room for improvement regarding the usability of the GUI. Given the large number of features that could be accessed from the main screen, a new user most likely will not be able to immediately know how to use it. Especially for users who are unfamiliar with logical programming and the notations used in *ABR*, it might take some time to understand what the predicates mean before they are able to convert the evidences they have into predicates and insert them into *ABR*. To use the tool efficiently, users will first have to overcome *ABR*'s steep learning curve, which could be accomplished with the help of user guides and tutorials.

Misinformation of past attacks Wrong attributions can be made if it is based on 'facts' derived from previously wrongly attributed attacks. For example, the Sony hack attribution was built on the (alleged) claim that North Korea was responsible for the assault on South Korean banks in 2013 [14]. The attribution of the Sony attack thus leverage on the correctness of the previous attributions. *ABR* is unable to resolve this dependency, since information derived from past attributions are used as ground truths in the background knowledge. Although this problem can be avoided by not using any information from previously attributed attacks, this will reduce the amount of evidence available. Analysts will then need to find more evidence from other sources in order to make the attribution. The attribution could sometimes even become impossible if the attackers did not leave much significant evidence behind. This problem is not specific to *ABR*, it is also faced by even the most experienced forensic analysts using existing methods.

8.4.2 Strengths

General performance As shown in Sections 8.2 and 8.3, the rules and preferences in *ABR* work as intended and the performance is acceptable for a real-time interactive application, with the longest runtime being 20s. Moreover, we have confirmed that even with extremely large number of evidences (up to 300), the total runtime is still around 60s (see Figure 8.10). According to the time scales in user experience [51], 1 minute is the time in which users should be able to "complete simple tasks". *ABR*'s performance is acceptable, given that it is a tool, not a website; its users are likely to be more patient than the website users mentioned in [51].

Performance of verbose execution Despite the drawbacks of using our current approach in verbose execution mentioned previously, we see them as a necessary trade-off for efficiency. Our approach of only considering the first solution makes the verbose execution a *linear time* operation, while attempting to consider all possible solutions is an *exponential time* operation; which is unsustainable for a real-time interactive application.

Usability Part of the problems associated with the usability of *ABR* stems from the fact that *ABR* provides the user with such numerous functions that can be performed from the GUI. Moreover, we can alleviate this problem by providing a comprehensive user guide with examples and tutorials on how to use the tool. Once users overcome the initial steepness of the learning curve, they will be able to fully utilise *ABR*.

Functionality *ABR* achieved the intended design motivations highlighted in Section 3.2, fitting into the iterative process of attribution, being transparent in its proof process and providing flexibility to users. Furthermore, *ABR* is the first tool of its kind that performs attribution using both technical and social evidences, combined with background knowledge that models the real world. *ABR* also has a suite of features built to support the core reasoner, including integration with forensic tools to automate extraction of evidences; assigning scores to each derivation using our own scoring system; visualisation of key structures relating to the execution results; and integrating any new evidences or rules into the rest of the *ABR* reasoner. These additional features help to shape the whole experience of using *ABR*, presenting necessary details to the user in a comprehensible manner.

Chapter 9

Conclusion and Future Work

We present in this chapter the conclusions with respect to the main achievements of this project, followed by some interesting ideas of how this project can be extended in future works.

9.1 Conclusion

In the 21st century, our lives are increasingly entwined with technology, which is becoming more and more interconnected. As this dependency and connectivity grows, we, as individuals and as part of nation-states, become increasingly vulnerable to cyber attacks. To better equip ourselves from these cyber attacks, accurate and timely attribution is critical in bolstering defences in a targeted way.

This project takes an initial step towards formalising the attribution process involving both technical and social evidences, using argumentation reasoning to model the deliberation of the analysts during attribution. The scope of this project is especially wide, stemming from complexity and sheer volume of information available on the topic of cyber security. Furthermore, our task was more than just comprehending existing methods of attribution in cyber security, but extracting the core reasoning used by experienced analysts into formal reasoning rules. We arranged the rules extracted from various case-studies, combining them to form one cohesive reasoner that uses an argumentation-based framework to return the attribution results.

During the course of this project, we encountered some unexpected challenges, e.g., struggles with the Gorgias framework, partly due to the limited documentation on its usage. We also changed the Prolog environment from SICStus Prolog to SWI-Prolog mid-way during implementation, consequently having to rewrite parts of the code.

Our main result for this project was the produced tool that achieves our initial design motivations of (i) modelling the iterative nature of attribution, (ii) achieving transparency by using comprehensive visualisations and (iii) having a flexible tool that allows the users to make the final decisions.

Beyond the basic setted functions, we also assembled a suite of tools/features that refines the functionality of *ABR*. Forensic tools are integrated into *ABR* to automate the extraction of some evidences, reducing the time spent on manually extracting these information, further automating the attribution process. In regard to assisting the user in adding new rules and preferences, we visualise the preference hierarchy, clearly showing how the rules relate to each other in terms of strength. This aids the user when adding their own rules and preferences by highlighting any deadlocks that were accidentally introduced. Additionally, when a new rule is inserted, *ABR* automatically scans for conflicting rules and display them to users and allows the users to add preferences between the new and existing rules. These features are key in guiding users when adding their own rules, in order to properly incorporate the new rules into the *ABR* reasoner.

Overall, *ABR* is powerful tool for automating parts of the attribution process. With further work described in the next section, *ABR* will redefine how attribution of cyber attacks is performed.

9.2 Future work

We now consider some interesting ideas of improvements that can enhance *ABR* that are left as future work.

Automated scraping for information about attacker on digital platforms An automated mechanism to obtain information that could pertain to the attacker would be beneficial, compared to manually looking through forums, blogs and other digital platforms. Since information such as the geopolitical scene might change over time, being able to automatically update the background knowledge will also be extremely helpful in keeping *ABR* updated with the times. This could be implemented by employing NLP techniques to extract relevant evidences from news articles, blogs, or technical papers, in a similar way to [52].

More sophisticated measure of cyber capability Currently, we only classify countries into three categories: (i) having large amount of resources (ii) having moderate amount of resources and (iii) having little or no resources. As future work, more sophisticated or detailed classification could be considered. In the report from Billo and Chang in [53], an in-depth study on the cyber capability of a few nation-states, as well as the motives that drive the nation states' actions is presented. Constructing a more detailed profile for the key nation-states will improve the accuracy of *ABR*, by matching the specific attributes of the attacker to distinct characteristics of nation-states.

Probabilities on results Incorporating the idea of probabilities from the InCA framework [33] could be a useful addition to *ABR*. The attribution results returned by *ABR* could have a corresponding probability, according to how definite, the attribution is. This probability could be linked to the reliability of the evidences used. For example, the IP address is a fairly weak evidence since it can be easily spoofed, but similarities in the code of malware used in the attack is a strong evidence. This extension can be used to resolve two major challenges in attribution.

Firstly, attackers might deliberately plant false evidences to mislead investigators. It is extremely difficult, even for experienced human analysts, to spot such deceptive evidences. However, though we are not likely to spot which evidences are deceptive, we can make use of the probabilities on the credibility of the evidence to reduce the effect of such deceptive evidences on the results given by *ABR*.

Another problem with attribution, according to Jeffrey Carr, president and CEO of Taia Global, is governments potentially make wrong attributions, and inaccurate information may be provided by ally nations. Take for example the Sony hack mentioned in Section 2.1.1, the FBI released a statement in less than a month that it has "concluded that the North Korean government is responsible" [54]. However, in the private sector, the view is not unanimous. Some were suspicious of how quickly the attribution was publicly confirmed after the attack, and later on, other evidence that Russians could have been involved was uncovered, that brings more doubt to the original attribution. Since evidence provided by the analyst is not cross-checked with other sources, if inaccurate evidences are given, ABR could make the wrong attribution.

In response to this problem, we could try to make use of contextual information to detect if a particular source of information, (e.g., a country or organisation), will benefit from giving inaccurate information; and if the information they gave is in their favour. Then, we could assign a credibility value to each ground truth in ABR, based on how much they benefit from giving inaccurate information and the degree to which the information is in their favour. We can then set a threshold on the ground truths to remove any ground truths with credibility value below the threshold value. However, the formula to assigning credibility will likely be extremely difficult to formulate, since both (i) the amount of benefits an entity will get from providing inaccurate information and (ii) how favourable the information is to the entity, are hard to quantify.

Bibliography

[1] S. Morgan, "Top 5 cybersecurity facts, figures and statistics for 2017." https://www.csoonline.com/article/3153707/security/top-5-cybersecurity-facts-figures-and-statistics-for-2017.html [Online. Last accessed: 2017-12-26], 2017.

[2] L. H. Newman, "The Biggest Cybersecurity Disasters of 2017 So Far." https://www.wired.com/story/2017-biggest-hacks-so-far/ [Online. Last accessed: 2017-12-26], 2017.

[3] Statista, "IoT: number of connected devices worldwide 2012-2025." https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/ [Online. Last accessed: 2017-12-26], 2018.

[4] K. J. Higgins, "How Lockheed Martin's 'Kill Chain' Stopped SecurID ...." https://www.darkreading.com/attacks-breaches/how-lockheed-martins-kill-chain-stopped-securid-attack/d/d-id/1139125? [Online. Last accessed: 2017-12-22].

[5] Z. J. Miller, "Sony Hack: Barack Obama Sanctions North Korea After Cyberattack." http://time.com/3652479/sony-hack-north-korea-the-interview-obama-sanctions/ [Online. Last accessed: 2018-01-26], 1 2015.

[6] M. J. Schwartz, "FBI's Sony Attribution: Doubts Continue." https://www.bankinfosecurity.com/fbis-sony-attribution-doubts-continue-a-7765 [Online. Last accessed: 2018-01-26], 2015.

[7] J.-a. Song, M. Murphy, and H. Kuchler, "North Korea blasts US over new sanctions amid cyber attack feud." https://www.ft.com/content/1f47b7b0-93e4-11e4-92dd-00144feabdc0 [Online. Last accessed: 2018-01-12], 2015.

[8] D. A. Wheeler and G. Larsen, "Techniques for Cyber Attack Attribution," *IDA Paper*, p. 82, 2003.

[9] R. K. Goutam, "The Problem of Attribution in Cyber Security," *International Journal of Computer Applications*, vol. 131, no. 7, pp. 975–8887, 2015.

[10] T. Rid and B. Buchanan, "Attributing Cyber Attacks," *Journal of Strategic Studies*, vol. 38, pp. 4–37, 1 2015.

[11] K. Zetter, *Countdown to Zero Day : Stuxnet and the launch of the world's first digital weapon*. New York City, USA: Crown Publishing Group, 2014.

[12] K. Zetter, "How Digital Detectives Deciphered Stuxnet, the Most Menacing Malware in History." https://www.wired.com/2011/07/how-digital-detectives-deciphered-stuxnet/ [Online. Last accessed: 2017-12-06], 2011.

[13] Mandiant, "Exposing One of China's Cyber Espionage Units," tech. rep., Mandiant, 2013.

[14] A. Altman and Z. J. Miller, "Sony Hack: FBI Accuses North Korea in Attack That Nixed The Interview." http://time.com/3642161/sony-hack-north-korea-the-interview-fbi/ [Online. Last accessed: 2018-01-10], 2014.

[15] J. Roman, "FBI Defends Sony Hack Attribution." https://www.bankinfosecurity.com/sony-a-7762 [Online. Last accessed: 2018-01-20], 2015.

[16] B. Todd and B. Brumfield, "Experts doubt North Korea was behind the big Sony hack." http://edition.cnn.com/2014/12/27/tech/north-korea-expert-doubts-about-hack/index.html [Online. Last accessed: 2018-01-18], 2014.

[17] US-CERT, "Computer Forensics." 2008.

[18] K. Kent, S. Chevalier, T. Grance, and H. Dang, "Guide to integrating forensic techniques into incident response," tech. rep., National Institute of Standards and Technology, Gaithersburg, MD, 2006.

[19] B. Carrier, "Defining Digital Forensic Examination and Analysis Tools Using Abstraction Layers," *International Journal of Digital Evidence Winter*, vol. 1, no. 4, 2003.

[20] M. H. Almeshekah and E. H. Spafford, *Cyber Deception, Building the Scientific Foundation*. Springer International Publishing, 2016.

[21] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis, "Detecting Targeted Attacks Using Shadow Honeypots," in *14th USENIX Security Symposium*, (Baltimore, MD, USA), 2005.

[22] L. F. d. C. Nassif and E. R. Hruschka, "Document Clustering for Forensic Analysis," *IEEE Transactions on Information Forensics and Security*, vol. 8, pp. 46–54, 1 2013.

[23] J. J. Yuill, *Defensive Computer-Security Deception Operations*. PhD thesis, North Carolina State University, 12 2007.

[24] P. Shakarian and J. Shakarian, "Socio-cultural modeling for cyber threat actors," in *Artificial Intelligence for Cyber Security, Papers from the 2016 AAAI Workshop*, (Phoenix, Arizona, USA), AI Access Foundation, 2016.

[25] Merriam-Webster, "Argumentation." https://www.merriam-webster.com/dictionary/argumentation [Online. Last accessed: 2018-06-17].

[26] D. Walton, "Argumentation Theory: A Very Short Introduction," in *Argumentation in Artificial Intelligence* (G. Simari and I. Rahwan, eds.), pp. 1–22, Boston, MA: Springer US, 2009.

[27] J. Mackay, P. Schulz, S. Rubinelli, and A. Pithers, "Online Patient Education and Risk Assessment," *Patient Education and Counseling*, 2007.

[28] S. Rubinelli, P. J. Schulz, and U. Hartung, "Argument-driven online genetic counselling," *Argument and Computation*, vol. 1, no. 3, pp. 199–214, 2010.

[29] S.C. Stumpf and J.T. McDonnell, "Talking about team framing," *Design Studies*, vol. 23, pp. 5–23, 1 2002.

[30] N. R. Velaga, "Development of an integrated flexible transport systems platform for rural areas using argumentation theory," *Research in Transportation Business & Management*, vol. 3, pp. 62–70, 8 2012.

[31] W. Ouerdane, N. Maudet, and A. Tsoukiàs, "Argumentation Theory and Decision Aiding," in *Trends in Multiple Criteria Decision Analysis*, pp. 177–208, Springer, Boston, MA, 2010.

[32] E. Nunes, N. Kulkarni, P. Shakarian, A. Ruef, and J. Little, "Cyber-Deception and Attribution in Capture-the-Flag Exercises," in *Cyber Deception, Building the Scientific Foundation* (S. Jajodia, V. S. Subrahmanian, V. Swarup, and C. Wang, eds.), pp. 149–165, Springer, 7 2016.

[33] P. Shakarian, G. I. Simari, G. Moores, and S. Parsons, "Cyber Attribution: An Argumentation-Based Approach," in *Cyber Warfare - Building the Scientific Foundation*, pp. 151–171, Springer, Cham, 2015.

[34] D. Gaertner and F. Toni, "CaSAPI: a system for credulous and sceptical argumentation," in *Workshop on Argumentation for Non-Monotonic Reasoning*, p. 80–95, 2007.

[35] H. Prakken and G. Sartor, "On the relation between legal language and legal argument," in *5th international conference on Artificial intelligence and law*, (New York, New York, USA), pp. 1–10, ACM Press, 1995.

[36] Wikipedia, "Abductive reasoning." https://en.wikipedia.org/wiki/Abductive_reasoning [Online. Last accessed: 2018-06-07].

[37] M. Denecker and A. Kakas, "Abduction in Logic Programming," pp. 402–436, 2002.

[38] E. Karafili, A. C. Kakas, N. I. Spanoudakis, and E. C. Lupu, "Argumentation-based Security for Social Good," in *AAAI 2017 Spring Symposium on AI for Social Good*, 5 2017.

[39] International Telecommunication Union and M. Minges, "Global Cybersecurity Index (GCI) 2017," tech. rep., 2017.

[40] K. Breene, "Who are the cyberwar superpowers?." https://www.weforum.org/agenda/2016/05/who-are-the-cyberwar-superpowers/ [Online. Last accessed: 2018-05-26], 2016.

[41] R. Pratka, "Which countries are allies and which are enemies?." https://www.msn.com/en-gb/news/photos/which-countries-are-allies-and-which-are-enemies/ss-BBBNVNJ [Online. Last accessed: 2018-04-20], 2017.

[42] YouGov, "America's Friends and Enemies." https://today.yougov.com/topics/politics/articles-reports/2017/02/02/americas-friends-and-enemies [Online. Last accessed: 2018-04-19], 2017.

[43] Brilliant Maps, "Who Americans Consider Their Allies, Friends and Enemies." https://brilliantmaps.com/us-allies-enemies/ [Online. Last accessed: 2018-04-19], 2017.

[44] FireEye, "Advanced Persistent Threat Groups." https://www.fireeye.com/current-threats/apt-groups.html [Online. Last accessed: 2018-02-21].

[45] S. Matin, "8 Active APT Groups To Watch." https://www.darkreading.com/endpoint/8-active-apt-groups-to-watch/d/d-id/1325161 [Online. Last accessed: 2018-02-21], 2016.

[46] Tor Project, "Tor Project: Overview." https://www.torproject.org/about/overview.html.en [Online. Last accessed: 2018-06-10].

[47] Wikipedia, "Intrusion detection system." https://en.wikipedia.org/wiki/Intrusion_detection_system [Online. Last accessed: 2018-05-11].

[48] "Is Signature and Rule-Based Intrusion Detection Sufficient? | CSO Online." https://www.csoonline.com/article/3181279/security/is-signature-and-rule-based-intrusion-detection-sufficient.html [Online. Last accessed: 2018-05-11], 2017.

[49] eTutorials, "Recipe 9.23 Decoding Snort Alert Messages." http://etutorials.org/Linux+systems/linux+security/Chapter+9.+Testing+and+Monitoring/Recipe+9.23+Decoding+Snort+Alert+Messages/ [Online. Last accessed: 2018-02-21].

[50] P. M. Nugues, "An Introduction to Prolog," in *Language processing with Perl and Prolog : theories, implementation, and application*, ch. Appendix A, p. 662, Berlin Heidelberg: Springer-Verlag, 2014.

[51] Nielsen, Jakob, "Powers of 10: Time Scales in User Experience." https://www.nngroup.com/articles/powers-of-10-time-scales-in-ux/ [Online. Last accessed: 2018-06-12], 2009.

[52] A. Wyner, J. Schneider, K. Atkinson, and T. Bench-Capon, "Semi-Automated Argumentative Analysis of Online Product Reviews," in *Computational Models of Argument*, (Vienna, Austria), pp. 43–50, 2012.

[53] C. Billo, W. Chang, and C. G. Billo, "Cyber Warfare: An analysis of the means and motivations of selected nation states," tech. rep., Institute for Security Technology Studies, Dartmouth College, 2004.

[54] T. Armerding, "Whodunit? In cybercrime, attribution is not easy." https://www.csoonline.com/article/2881469/malware-cybercrime/whodunit-in-cybercrime-attribution-is-not-easy.html [Online. Last accessed: 2017-12-16], 2015.

[55] B. Bartholomew and J. A. Guerrero-Saade, "Wave your false flags! Deception tactics muddying attribution in targeted attacks," in *Virus Bulletin Conference*, 2016.

[56] Nicole Perlroth, "Online Banking Attacks Were Work of Iran, U.S. Officials Say." http://www.nytimes.com/2013/01/09/technology/online-banking-attacks-were-work-of-iran-us-officials-say.html [Online. Last accessed: 2017-11-21], 2013.

[57] D. Goldman, "Major banks hit with biggest cyberattacks in history." http://money.cnn.com/2012/09/27/technology/bank-cyberattacks/index.html [Online. Last accessed: 2017-11-22], 2012.

[58] A. Fitzpatrick, "Meet the 'Gauss' Virus, Stuxnet and Flame's New Cousin." https://mashable.com/2012/08/09/gauss-virus/?europe=true#M8v5wovsp5q0 [Online. Last accessed: 2018-05-31], 2012.

[59] Kaspersky Lab Global Research and Analyst Team, "Gauss: Abnormal Distribution," tech. rep., Kaspersky Lab, 2012.

[60] L. Dignan, "Meet Gauss: The latest cyber-espionage tool." https://www.zdnet.com/article/meet-gauss-the-latest-cyber-espionage-tool/ [Online. Last accessed: 2018-05-31], 2012.

[61] S. Knapton, "Home Office blames North Korea for devastating NHS'WannaCry' cyber attack." http://www.telegraph.co.uk/science/2017/10/27/home-office-blames-north-korea-devastating-nhs-wannacry-cyber/ [Online. Last accessed: 2017-11-26], 2017.

[62] R. Browne, "North Korea hackers trying to steal bitcoin to evade sanctions." https://www.cnbc.com/2017/09/12/north-korea-hackers-trying-to-steal-bitcoin-evade-sanctions.html [Online. Last accessed: 2017-11-25], 2017.

[63] Skynews, "'Strong evidence' North Korea-linked group was behind NHS cyberattack." https://news.sky.com/story/cyberattack-tech-firms-investigate-north-korea-linked-hackers-10879388 [Online. Last accessed: 2017-11-26], 2017.

[64] BBC-News, "More evidence for WannaCry 'link' to North Korean hackers." http://www.bbc.co.uk/news/technology-40010996 [Online. Last accessed: 2017-11-30], 2017.

Appendix A

Core Gorgias Rules

In this chapter, we explain some of the key strategic rules that derives `isCulprit(X, Att)`, and list all the technical, operational and strategic rules and corresponding preferences used in *ABR* for completeness.

A.1 Explanation of Rules

The layers in *ABR* are closely linked, instead of talking about each layer separately, we go through specific rules, starting from the strategic layer, then move forward to the operation and technical layer while explaining. (Rule 2 is also explained in Section 4.3.)

A.1.1 Rule 1: Group claimed responsibility

Starting with the first rule, we show the strategic rule and the corresponding rules to prove its body predicate `existingGroupClaimedResponsibility/2`:

```
% from str_rules.pl
rule(r_str__claimedResp(X,Att), isCulprit(X,Att),
 ↪  [existingGroupClaimedResponsibility(X,Att)]).

% from op_rules.pl
rule(r_op_claimResp0(X,Att),
 ↪  existingGroupClaimedResponsibility(X,Att),
 ↪  [claimedResponsibility(X,Att)]).
rule(r_op_claimResp1(X,Att),
 ↪  neg(existingGroupClaimedResponsibility(X,Att)),
 ↪  [claimedResponsibility(X,Att), noPriorHistory(X)]).
rule(p5_op(),prefer(r_op_claimResp1(X,A),r_op_claimResp0(X,A)),[]).
```

Both `claimedResponsibility/2` and `noPriorHistory/1` are base evidences. We can prove that a country or group was behind the attack (`isCulprit/2`) if it publicly claimed responsibility for the attack. However, in many cases groups claiming responsibility might be just a façade to mislead us. A hacktivist group with absolutely no history or lineage claiming responsibility for a large-scale attack is one possible hint that the group is not the true culprit of the attack [55]. This is why we introduced the predicate `existingGroupClaimedResponsibility/2` in the `operational` layer, that denotes the fact that a group that has some prior history (we are unable to prove `noPriorHistory(X)`) has claimed responsibility for the attack. The preference rule (`p5_op()`) ensures that when both predicates are present, only `neg(existingGroupClaimedResponsibility(X,Att))` will be proven.

A.1.2 Rule 2: Has motive and capability

Rule 2 uses the predicates `hasMotive/2` and `hasCapability/2`.

```
% from str_rules.pl
rule(r_str__motiveAndCapability(C,Att), isCulprit(C,Att),
 ↪  [hasMotive(C,Att),hasCapability(C,Att)]).
```

hasMotive(X, Att) Since there are many rules that proves these two predicates (9 for `hasMotive/2` and 4 for `hasCapability/2`), we break them down into smaller parts. First, we cover the explanations for `hasMotive/2`.

```
% from op_rules.pl
rule(r_op_ecMotive(C,T), hasMotive(C,Att),
 ↪  [target(T,Att),industry(T), hasEconomicMotive(C,T),
 ↪  contextOfAttack(economic,Att), specificTarget(Att)]).

abducible(specificTarget(_Att), []).
rule(r_op_notTargetted(Att), neg(specificTarget(Att)),
 ↪  [targetCountry(T1, Att), targetCountry(T2, Att), T1 \= T2]).

abducible(contextOfAttack(economic, _Att), []).
rule(r_op_context(economic,Att), contextOfAttack(economic,Att),
 ↪  [target(T,Att), normalIndustry(Ind)]).
```

This rule can be read as: if a country/group has an economic motive to attack the industry that was targeted in the attack, and the context of the attack was economic, and the attack had a specific target, then the country/group has motive to carry out the attack.

The predicate `target(T, Att)` is a base evidence, where `T` is the target of the attack `Att`, while the predicate `industry(T)` is one of the background facts. It is true when `T` is an industry (see table:bg for the full list of background facts).

The predicate `hasEconomicMotive(C, T)` is also a base evidence. When `hasEconomicMotive(countryX, industryY)` is true, it means `countryX` will benefit economically from attacking `industryY`. For example, if countryX has identified industryY as a strategic emerging industry in official public documents such as white papers or other government reviews, we say that `hasEconomicMotive(countryX, industryY)` is true.

The predicate `contextOfAttack/2` and `specificTarget/1` are both abducibles. In Section 4.6, we cover the details of the use of abducibles in *ABR*. Briefly, an attack is assumed to be targeted unless more than one country is targeted (`targetCountry/2`).

Since `targetCountry/2` is a base evidence, it can be omitted at the user's discretion, if the user deem the number of infections in another country is too insignificant.

The predicate `contextOfAttack(economic, Att)` can be proven if the target is a 'normal' industry. A 'normal' industry, as opposed to a 'political' industry, are industries that are not closely related to a country's national interests (see Table 4.1 for more details).

```
% from op_rules.pl
rule(r_op_pMotive(C,T), hasMotive(C,Att),
 ↪  [targetCountry(T,Att), attackPeriod(Att,Date1),
 ↪  hasPoliticalMotive(C,T,Date2), dateApplicable(Date1,Date2),
 ↪  contextOfAttack(political,Att), specificTarget(Att)]).

rule(r_op_pMotive1(C,T,Date),
 ↪  hasPoliticalMotive(C,T,Date),[imposedSanctions(T,C,Date)]).

rule(r_op_context(political, Att), contextOfAttack(political, Att),
 ↪  [target(T, Att), country(T)]).
rule(r_op_context1(political, Att), contextOfAttack(political, Att),
 ↪  [target(T, Att), industry(Ind, T), politicalIndustry(Ind)]).

%% Auxilliary rule
%% dateApplicable(Y1,M1,Y2,M2) is true if Y2 M2 is before Y1 M1 but
 ↪  close enough (within 1 year)
rule(r_op_date(ongoing), dateApplicable(_, ongoing), []).
rule(r_op_date1(Y, M), dateApplicable([Y, M], [Y, M]), []).
rule(r_op_date2(Y, M1, M2), dateApplicable([Y, M1], [Y, M2]), [M2 <
 ↪  M1]).
rule(r_op_date3(Y1, Y2), dateApplicable([Y1, _], [Y2, _]), [Y2 < Y1,
 ↪  Y2 > (Y1 - 2)]).
```

This rule can be interpreted as: if a country has a political motive (we explain later what this means) to attack the target country, and the order of events fits into the general timeline (more on `dateApplicable/2` later), and the attack was a targeted attack, then the country has motive to perform the attack.

The predicate `targetCountry(T, Att)` is a base evidence, saying that country `T` is the country that was targeted by attack `Att`.

The predicate `attackPeriod(X, Date)` denotes the date of the attack. `Date` is a list, in the format `[YYYY,MM]`. We exclude the day since in many cases even though the malware is discovered on a certain day, we are unsure of when the systems were actually infiltrated.

The predicate `hasPoliticalMotive(C,T,Date2)` is a derived predicate. *ABR* has one rule that proves `hasPoliticalMotive(C,T,Date2)`. If the target country has imposed sanctions on country `C`, then as a form of retaliation, country `C` has political motive to attack the target country (`T`).

The predicate `dateApplicable/2` is an auxiliary rule used to ensure that the triggering event (in this case, the imposing of sanctions) takes place before the attack, and occurred shortly before the attack. Alternatively, if the event is long-term and ongoing, we use the constant `ongoing` in place of the date of the event

([YYYY,MM]), `dateApplicable(_, ongoing)` is always true.

Lastly, we can prove `contextOfAttack(political, Att)` if either the target is a country, or the target is a 'political' industry (industries that are closely related to a country's national interests, such as the military or energy sector).

```
% from op_rules.pl
rule(r_op_conflict1(X,T), hasMotive(X,Att), [target(T,Att),
↪  attackPeriod(Att,Date1), news(Event,T,Date2),
↪  dateApplicable(Date1,Date2), causeOfConflict(X,T,Event),
↪  specificTarget(Att)]).
```

The main idea behind this rule is that if (i) an incident occurred in the target country and was publicized, and (ii) that incident is the cause of international conflict or tension with another country shortly before the attack, then the other country has motive to attack the target country.

To better explain this rule, we use a real-world example. The Sony Pictures hack in 2014 was attributed to North Korea, that allegedly attacked Sony Pictures in retaliation for the upcoming North Korean-based comedy "The Interview". The relevant evidences are as follows:

```
% evidences are written in Prolog syntax for simplicity
target(sony, sonyhack).
attackPeriod(sonyhack,[2014,11]).
news(theInterview, sony,[2013,10]).
causeOfConflict(north_korea, sony, theInterview).
```

We know that the scandal revolving "The Interview" was publicized in October 2013 (`news(theInterview, sony,[2013,10])`), and this sparked conflict between North Korea and Sony Pictures (`causeOfConflict(north_korea, sony, theInterview)`). The target of the attack was Sony Pictures (`target(sony, sonyhack)`) and the attack occurred in November 2014 (`attackPeriod(sonyhack,[2014,11])`), which was shortly before the attack occurred. So using these evidences and the above rules, we are able to arrive at the conclusion that North Korea has motive to perform the Sony Pictures hack.

```
% from op_rules.pl
rule(r_op_nonGeopolitics1(C,T),        neg(hasMotive(C,Att)),
↪  [targetCountry(T,Att), country(T), country(C),
↪  goodRelation(C,T)]).
rule(r_op_nonGeopolitics2(C,T),        neg(hasMotive(C,Att)),
↪  [targetCountry(T,Att), country(T), country(C),
↪  goodRelation(T,C)]).
```

These rules make use of the background facts `goodRelation/2`. We clarify how these facts are used in detail in Section 4.4.1.

```
% from op_rules.pl
rule(r_op_grpPastTargets(Group, Att),
↪  hasMotive(Group,Att),[target(T,Att), prominentGroup(Group),
↪  pastTargets(Group,Ts), member(T,Ts)]).
```

The above rule makes use of background facts `prominentGroup/1` and `pastTargets/2`. In essence, this rule says that if a prominent Advanced Persistent Thread (APT) group has performed an attack on the target of this attack before, then they possibly have motive to perform this attack too. More details can be found in Section 4.4.2.

hasCapability(X, Att) Next, we cover the explanations for `hasCapability/2`, where its derivation rules are shown below.

```
% from op_rules.pl
rule(r_op_hasCapability1(X, Att), hasCapability(X, Att),
 ↪  [neg(requireHighResource(Att))]).
rule(r_op_hasCapability2(X, Att), hasCapability(X, Att),
 ↪  [requireHighResource(Att), hasResources(X)]).
rule(r_op_noCapability1(X, Att), neg(hasCapability(X, Att)),
 ↪  [requireHighResource(Att), neg(hasResources(X))]).

rule(r_op_noCapability2(X, Att), neg(hasCapability(X, Att)),
 ↪  [hasNoResources(X)]).
```

In Table A.1, we summarise the situations where `hasCapability(X,Att)` is true. Using logical notation, the first three rules can be represented by

$$\neg requireHighResource(Att) \lor hasResources(X) \rightarrow hasCapability(X, Att)$$

Gorgias does not support the use of the 'or' operator within the body, so we used several rules to implement this relationship. While `neg(hasResources(X))` is be read as 'X does not have (large amounts) of resources', `hasNoResources(X)` is read as 'X has little or no resources'. The last rule says that if X has little or no resources, then X will not be able to perform any attack, even if the attack does not require large amounts of resources.

Table A.1: Summary table of when `hasCapability(X,Att)` is true

	hasResources(X)	neg(hasResources(X))
requireHighResource(Att)	✓hasCapability(X, Att)	✗neg(hasCapability(X, Att))
neg(requireHighResource(Att))	✓hasCapability(X, Att)	✓hasCapability(X, Att)

The predicates `requireHighResource/1` and `hasResources/1` are both derived evidences. We now discuss how they are derived.

```
% from tech_rules.pl
rule(r_t_highResource0(Att), neg(requireHighResource(Att)),
 ↪          [neg(highLevelSkill(Att))]).
rule(r_t_highResource1(Att), requireHighResource(Att),
 ↪                    [highLevelSkill(Att)]).
rule(r_t_highResource2(Att), requireHighResource(Att),
 ↪                    [target(T, Att), highSecurity(T)]).
rule(r_t_highResource3(Att), requireHighResource(Att),
 ↪                    [highVolumeAttack(Att),
 ↪  longDurationAttack(Att)]).
```

```

% preferences
rule(p12a_t(), prefer(r_t_highResource1(Att),
↪  r_t_highResource0(Att)), []).
rule(p12b_t(), prefer(r_t_highResource2(Att),
↪  r_t_highResource0(Att)), []).
rule(p12c_t(), prefer(r_t_highResource3(Att),
↪  r_t_highResource0(Att)), []).
```

Taking into account the preferences, we can summarise the above rules by the following logical notion:

$$\begin{aligned} requireHighResource(Att) \leftarrow & (highLevelSkill(Att) \vee \\ & (highSecurity(T) \wedge target(T, Att)) \vee \\ & (highVolumeAttack(Att) \wedge longDurationAttack(Att)). \end{aligned}$$

The predicates `highSecurity/1, highVolumeAttack/1, longDurationAttack/1` are all base evidences, and have the following intuitive definitions:

highSecurity(T) (Organisation or company) T has high-security;

highVolumeAttack(Att) The attack had a high volume;

longDurationAttack(Att) The attack was performed over a long duration (few months or even years).

The predicate `highLevelSkill/1` is a derived evidence. We show below the rules that proves `highLevelSkill/1`.

```
rule(r_t_neghighSkill(Att), neg(highLevelSkill(Att)), []).
rule(r_t_highSkill1(Att), highLevelSkill(Att),
↪           [hijackCorporateClouds(Att)]).
rule(r_t_highSkill2(Att), highLevelSkill(Att),
↪           [malwareUsedInAttack(M, Att),
↪  usesZeroDayVulnerabilities(M)]).
rule(r_t_highSkill3(Att), neg(highLevelSkill(Att)),
↪  [malwareUsedInAttack(M, Att), fromBlackMarket(M)]).
rule(r_t_highSkill4(Att), highLevelSkill(Att),
↪  [malwareUsedInAttack(M, Att), sophisticatedMalware(M)]).

% preferences
rule(p10a_t(), prefer(r_t_highSkill1(Att), r_t_neghighSkill(Att)),
↪  []).
rule(p10b_t(), prefer(r_t_highSkill2(Att), r_t_neghighSkill(Att)),
↪  []).
rule(p10c_t(), prefer(r_t_highSkill4(Att), r_t_neghighSkill(Att)),
↪  []).
rule(p11a_t(), prefer(r_t_highSkill3(Att), r_t_highSkill2(Att)),
↪  []).
rule(p11b_t(), prefer(r_t_highSkill3(Att), r_t_highSkill4(Att)),
↪  []).
```

There are many base evidences in these rules. We summarise their definitions below:

malwareUsedInAttack(M, Att) `M` is the malware used in the attack `Att`;

hijackCorporateClouds(Att) Corporate cloud servers were hijacked as part of the attack `Att`;

usesZeroDayVulnerabilities(M) At least one zero day vulnerability was used in the malware `M`;

fromBlackMarket(M) The malware `M` was being sold on the black market;

sophisticatedMalware(M) The malware `M` was technically sophisticated.

After applying the preferences, we have the following definition for `highLevelSkill(Att)` in logical notation:

$$\begin{aligned} highLevelSkill(Att) \leftarrow\ & hijackCorporateClouds(Att) \vee \\ & (malwareUsedInAttack(M, Att) \wedge \\ & \quad \neg fromBlackMarket(M) \wedge \\ & \quad (sophisticatedMalware(M) \vee \\ & \quad\quad usesZeroDayVulnerabilities(M))). \end{aligned}$$

If the malware used in the attack was bought from the black market, meaning it was not an original malware, not that much resources are required to perform the attack, regardless of how high level the malware is.

```
% from op_rules.pl
rule(r_op_hasResources1(X), hasResources(X), [gci_tier(X,
↪  leading)]).
rule(r_op_hasResources2(X), hasResources(X), [cybersuperpower(X)]).
rule(r_op_hasNoResources(X), hasNoResources(X), [gci_tier(X,
↪  initiating)]).
```

`gci_tier/2` and `cybersuperpower/1` are background facts. `gci_tier(X, Group)` means the country `X` is in Global Cybersecurity Index[1] group (as of 2017) `Group`. `Group` can take 3 possible values: `leading`, `maturing`, `initiating` (in order of descending GCI index). The higher the GCI, the higher the cybersecurity capabilities of the country. `cybersuperpower(X)` means country `X` is identified as a cyber superpower. More details on the GCI index and cyber superpowers can be found in Section 4.4.1.

A.1.3 Rule 3: APT group to origin country

The below rule describes the situation when an attribution to an APT group can be extended to its country of origin.

[1] https://www.itu.int/dms_pub/itu-d/opb/str/D-STR-GCI.01-2017-PDF-E.pdf

```
rule(r_str__aptGroupMotive(C,Att), isCulprit(C,Att),
↪  [prominentGroup(Group), groupOrigin(Group,C), country(C),
↪  isCulprit(Group,Att), hasMotive(C,Att)]).
```

If we could make the attribution to an APT group (`prominentGroup(Group)`, `isCulprit(Group,Att)`) and its country of origin has the motive to perform the attack (`groupOrigin(Group,C)`, `hasMotive(C,Att)`), then we extend the attribution to the country.

A.1.4 Rule 4: Has location and motive

When we can prove that the attack originated from a country and that country has motive to perform the attack, then we attribute the attack to that country, as shown below.

```
rule(r_str__motiveAndLocation(C,Att), isCulprit(C,Att),
↪  [attackOrigin(C,Att), hasMotive(C,Att), country(C)]).
```

attackOrigin(X, Att) `attackOrigin(X, Att)` means the attack `Att` was found to have come from country `X`. These are the rules pertaining to `attackOrigin/2` (or its negation):

```
% from tech_rules.pl
rule(r_t_attackOriginDefault(X, Att), neg(attackOrigin(X, Att)),
↪  []).
rule(r_t_attackOrigin(X, Att), attackOrigin(X, Att),
↪  [attackPossibleOrigin(X, Att)]).
rule(r_t_conflictingOrigin(X, Y, Att), neg(attackOrigin(X, Att)),
↪  [attackPossibleOrigin(X, Att), attackPossibleOrigin(Y, Att),
↪  country(X), country(Y), X \= Y]).
```

`attackPossibleOrigin(X, Att)` is an auxiliary predicate. These rules denote that if we can prove one country is the possible attack origin, then that is the attack origin. However, if we can prove two distinct possible attack origins, then neither is the attack origin.

```
rule(r_t_noLocEvidence(X, Att), neg(attackPossibleOrigin(X, Att)),
↪  []).
rule(r_t_srcIP1(X, Att),   attackPossibleOrigin(X, Att),
↪  [attackSourceIP(IP, Att), ipGeoloc(X, IP)]).
rule(r_t_srcIP2(X, Att),   attackPossibleOrigin(X, Att),
↪  [majorityIpOrigin(X, Att)]).
rule(r_t_spoofIP(X, Att),  neg(attackPossibleOrigin(X, Att)),
↪  [attackSourceIP(IP, Att), spoofedIP(IP, Att), ipGeoloc(X, IP)]).
rule(r_t_spoofIPtor(IP),  spoofedIP(IP, Att),  [attackSourceIP(IP,
↪  Att),  targetServerIP(TargetServerIP, Att),  torIP(IP,
↪  TargetServerIP)]).
rule(r_t_lang1(X, Att),   attackPossibleOrigin(X, Att),
↪  [sysLanguage(L, Att), firstLanguage(L, X)]).
```

```
rule(r_t_lang2(X, Att),   attackPossibleOrigin(X, Att),
 ↪  [languageInCode(L, Att), firstLanguage(L, X)]).
rule(r_t_infra(X, Att),   attackPossibleOrigin(X, Att),
 ↪  [infraUsed(Infra, Att), infraRegisteredIn(X, Infra)]).
rule(r_t_domain(X, Att),  attackPossibleOrigin(X, Att),
 ↪  [malwareUsedInAttack(M, Att), ccServer(S, M),
 ↪  domainRegisteredDetails(S, _, Addr), addrInCountry(Addr, X)]).
```

A.2 All rules used in *ABR*

In this section we show all the rules used in the technical, operational and strategic layers in *ABR*.

A.2.1 All technical rules

Below we show all the technical rules and their preferences that can be found in `tech_rules.pl`.

```
rule(r_t_neghighSkill(Att), neg(highLevelSkill(Att)), []).
rule(r_t_highSkill1(Att), highLevelSkill(Att), [hijackCorporateClouds(Att)]).
rule(r_t_highSkill2(Att), highLevelSkill(Att), [malwareUsedInAttack(M, Att),
 ↪  usesZeroDayVulnerabilities(M)]).
rule(r_t_highSkill3(Att), neg(highLevelSkill(Att)), [malwareUsedInAttack(M,
 ↪  Att), neg(notFromBlackMarket(M))]).
rule(r_t_highSkill4(Att), highLevelSkill(Att), [malwareUsedInAttack(M, Att),
 ↪  sophisticatedMalware(M)]).

rule(r_t_highResource0(Att), neg(requireHighResource(Att)),
 ↪  [neg(highLevelSkill(Att))]).
rule(r_t_highResource1(Att), requireHighResource(Att), [highLevelSkill(Att)]).
rule(r_t_highResource2(Att), requireHighResource(Att), [target(T, Att),
 ↪  highSecurity(T)]).
rule(r_t_highResource3(Att), requireHighResource(Att), [highVolumeAttack(Att),
 ↪  longDurationAttack(Att)]).

rule(r_t_IPdomain1(S, M),  ccServer(S, M), [malwareUsedInAttack(M, Att),
 ↪  attackSourceIP(IP, Att),  ipResolution(S, IP, _D)]).
rule(r_t_IPdomain2(S, M),  neg(ccServer(S, M)), [malwareUsedInAttack(M, Att),
 ↪  attackSourceIP(IP, Att), spoofedIP(IP, Att), ipResolution(S, IP, _D)]).
rule(r_t_IPdomain3(S, M),  neg(ccServer(S, M)), [malwareUsedInAttack(M, Att),
 ↪  attackSourceIP(IP, Att), attackPeriod(Att, D1), ipResolution(S, IP, D),
 ↪  neg(recent(D, D1))]).

% to get auto ip resolution via virustotal
rule(r_t_IP(IP,  Date),  ip(IP, Date), [ip(IP),  attackSourceIP(IP, Att),
 ↪  attackPeriod(Att, Date)]).

rule(r_t_noLocEvidence(X, Att), neg(attackPossibleOrigin(X, Att)), []).
rule(r_t_srcIP1(X, Att),   attackPossibleOrigin(X, Att), [attackSourceIP(IP,
 ↪  Att), ipGeoloc(X, IP)]).
rule(r_t_srcIP2(X, Att),   attackPossibleOrigin(X, Att), [majorityIpOrigin(X,
 ↪  Att)]).
```

```
rule(r_t_spoofIP(X, Att),  neg(attackPossibleOrigin(X, Att)),
 ↪  [attackSourceIP(IP, Att), spoofedIP(IP, Att), ipGeoloc(X, IP)]).
rule(r_t_spoofIPtor(IP),  spoofedIP(IP, Att), [attackSourceIP(IP, Att),
 ↪  targetServerIP(TargetServerIP, Att),  torIP(IP,  TargetServerIP)]).
rule(r_t_lang1(X, Att),   attackPossibleOrigin(X, Att), [sysLanguage(L, Att),
 ↪  firstLanguage(L, X)]).
rule(r_t_lang2(X, Att),   attackPossibleOrigin(X, Att), [languageInCode(L, Att),
 ↪  firstLanguage(L, X)]).
rule(r_t_infra(X, Att),   attackPossibleOrigin(X, Att), [infraUsed(Infra, Att),
 ↪  infraRegisteredIn(X, Infra)]).
rule(r_t_domain(X, Att),  attackPossibleOrigin(X, Att), [malwareUsedInAttack(M,
 ↪  Att), ccServer(S, M), domainRegisteredDetails(S, _, Addr),
 ↪  addrInCountry(Addr, X)]).

rule(r_t_recent1(Y),  recent([Y, _], [Y, _]),  []).
rule(r_t_recent2(Y1, Y2, M1, M2),  recent([Y1, M1], [Y2, M2]),  [Y1 is Y2 - 1,
 ↪  M1 > M2]).
rule(r_t_recent3(Y1, Y2, M1, M2),  recent([Y1, M1], [Y2, M2]),  [Y2 is Y1 - 1,
 ↪  M2 > M1]).

rule(r_t_attackOriginDefault(X, Att), neg(attackOrigin(X, Att)), []).
rule(r_t_attackOrigin(X, Att), attackOrigin(X, Att), [attackPossibleOrigin(X,
 ↪  Att)]).
rule(r_t_conflictingOrigin(X, Y, Att), neg(attackOrigin(X, Att)),
 ↪  [attackPossibleOrigin(X, Att), attackPossibleOrigin(Y, Att), country(X),
 ↪  country(Y), X \= Y]).

rule(r_t_bm(M), notFromBlackMarket(M), [infectionMethod(usb, M),
 ↪  commandAndControlEasilyFingerprinted(M)]).

rule(r_t_similarDefault(M1, M2), neg(similar(M1, M2)),  []).
rule(r_t_similar(M1, M2), similar(M1, M2), [similarCCServer(M1, M2), M1 \= M2]).
rule(r_t_simCC1(M1, M2), similarCCServer(M1, M2), [ccServer(S, M1), ccServer(S,
 ↪  M2)]).
rule(r_t_simCC2(M1, M2), similarCCServer(M1, M2), [ccServer(S1, M1),
 ↪  ccServer(S2, M2), S1 \= S2, domainRegisteredDetails(S1, _, A),
 ↪  domainRegisteredDetails(S2, _, A)]).
rule(r_t_simCC3(M1, M2), similarCCServer(M1, M2), [ccServer(S1, M1),
 ↪  ccServer(S2, M2), S1 \= S2, domainRegisteredDetails(S1, Name, _),
 ↪  domainRegisteredDetails(S2, Name, _)]).

rule(r_t_similar1(M1, M2), similar(M1, M2), [similarCodeObfuscation(M1, M2)]).
rule(r_t_similar2(M1, M2), similar(M1, M2), [sharedCode(M1, M2)]).
rule(r_t_similar3(M1, M2), similar(M1, M2), [malwareModifiedFrom(M1, M2)]).
rule(r_t_similar4(M1, M2), similar(M1, M2), [M1 \= M2, fileCharaMalware(C1, M1),
 ↪  fileCharaMalware(C2, M2), similarFileChara(C1, C2)]).

rule(r_t_targetted(Att), specificTarget(Att), [malwareUsedInAttack(M, Att),
 ↪  specificConfigInMalware(M)]).

rule(r_t_similarFileChara1(C1, C2), similarFileChara(C1, C2),
 ↪  [fileChara(Filename, _, _, _, _, _, C1), fileChara(Filename, _, _, _, _, _,
 ↪  C2)]).
rule(r_t_similarFileChara2(C1, C2), similarFileChara(C1, C2), [fileChara(_, MD5,
 ↪  _, _, _, _, C1), fileChara(_, MD5, _, _, _, _, C2)]).
rule(r_t_similarFileChara3(C1, C2), similarFileChara(C1, C2), [fileChara(_, _,
 ↪  _, _, Desc, _, C1), fileChara(_, _, _, _, Desc, _, C2)]).
```

```
rule(r_t_similarFileChara4(C1, C2), similarFileChara(C1, C2), [fileChara(_, _,
↪  Size, CompileTime, _, Filetype, C1), fileChara(_, _, Size, CompileTime, _,
↪  Filetype, C2)]).


% preferences
rule(p1_t(), prefer(r_t_attackOrigin(X, Att), r_t_attackOriginDefault(X, Att)),
↪  []).
rule(p4a_t(), prefer(r_t_srcIP1(X, Att), r_t_noLocEvidence(X, Att)), []).
rule(p4b_t(), prefer(r_t_srcIP2(X, Att), r_t_noLocEvidence(X, Att)), []).
rule(p5_t(), prefer(r_t_lang1(X, Att), r_t_noLocEvidence(X, Att)), []).
rule(p6_t(), prefer(r_t_lang2(X, Att), r_t_noLocEvidence(X, Att)), []).
rule(p7_t(), prefer(r_t_infra(X, Att), r_t_noLocEvidence(X, Att)), []).
rule(p8_t(), prefer(r_t_domain(X, Att), r_t_noLocEvidence(X, Att)), []).
rule(p9a_t(), prefer(r_t_spoofIP(X, Att), r_t_srcIP1(X, Att)), []).
rule(p9b_t(), prefer(r_t_spoofIP(X, Att), r_t_srcIP2(X, Att)), []).
rule(p10a_t(), prefer(r_t_highSkill1(Att), r_t_neghighSkill(Att)), []).
rule(p10b_t(), prefer(r_t_highSkill2(Att), r_t_neghighSkill(Att)), []).
rule(p10c_t(), prefer(r_t_highSkill4(Att), r_t_neghighSkill(Att)), []).
rule(p11b_t(), prefer(r_t_highSkill3(Att), r_t_highSkill2(Att)), []).
rule(p11c_t(), prefer(r_t_highSkill3(Att), r_t_highSkill4(Att)), []).
rule(p12a_t(), prefer(r_t_highResource1(Att), r_t_highResource0(Att)), []).
rule(p12b_t(), prefer(r_t_highResource2(Att), r_t_highResource0(Att)), []).
rule(p12c_t(), prefer(r_t_highResource3(Att), r_t_highResource0(Att)), []).
rule(p13a_t(), prefer(r_t_IPdomain2(S, M), r_t_IPdomain1(S, M)), []).
rule(p13b_t(), prefer(r_t_IPdomain3(S, M), r_t_IPdomain1(S, M)), []).
rule(p14a_t(), prefer(r_t_similar(M1, M2), r_t_similarDefault(M1, M2)), []).
rule(p14b_t(), prefer(r_t_simCC1(M1, M2), r_t_similarDefault(M1, M2)), []).
rule(p14c_t(), prefer(r_t_simCC2(M1, M2), r_t_similarDefault(M1, M2)), []).
rule(p14d_t(), prefer(r_t_simCC3(M1, M2), r_t_similarDefault(M1, M2)), []).
```

A.2.2 All operational rules

Below we show all the operational rules and their preferences that can be found in `op_rules.pl`.

```
%% Main rules:
abducible(specificTarget(_Att), []).
abducible(contextOfAttack(political, _Att), []).
abducible(contextOfAttack(economic, _Att), []).

rule(r_op_hasResources1(X), hasResources(X), [gci_tier(X, leading)]).
rule(r_op_hasResources2(X), hasResources(X), [cybersuperpower(X)]).
rule(r_op_hasNoResources(X),  hasNoResources(X), [gci_tier(X, initiating)]).

% more than one country targetted
rule(r_op_notTargetted(Att), neg(specificTarget(Att)), [targetCountry(T1, Att),
↪  targetCountry(T2, Att), T1 \= T2]).

rule(r_op_hasCapability1(X, Att), hasCapability(X, Att),
↪  [neg(requireHighResource(Att))]).
rule(r_op_hasCapability2(X, Att), hasCapability(X, Att),
↪  [requireHighResource(Att), hasResources(X)]).
rule(r_op_noCapability1(X, Att), neg(hasCapability(X, Att)),
↪  [requireHighResource(Att), neg(hasResources(X))]).
rule(r_op_noCapability2(X, Att), neg(hasCapability(X, Att)),
↪  [hasNoResources(X)]).
```

```


rule(r_op_ecMotive(C, T),         hasMotive(C, Att), [target(T, Att),
 ↪  industry(T), contextOfAttack(economic, Att), hasEconomicMotive(C, T),
 ↪  specificTarget(Att)]).
rule(r_op_pMotive(C, T), hasMotive(C, Att), [targetCountry(T, Att),
 ↪  attackPeriod(Att, Date1), contextOfAttack(political, Att),
 ↪  hasPoliticalMotive(C, T, Date2), dateApplicable(Date1, Date2),
 ↪  specificTarget(Att)]).
rule(r_op_pMotive1(C, T, Date), hasPoliticalMotive(C, T, Date),
 ↪  [imposedSanctions(T, C, Date)]).
rule(r_op_conflict(X, T), hasMotive(X, Att), [targetCountry(T, Att),
 ↪  attackPeriod(Att, Date1), news(Event, T, Date2), dateApplicable(Date1,
 ↪  Date2), causeOfConflict(X, T, Event), specificTarget(Att)]).
rule(r_op_conflict1(X, T), hasMotive(X, Att), [target(T, Att), attackPeriod(Att,
 ↪  Date1), news(Event, T, Date2), dateApplicable(Date1, Date2),
 ↪  causeOfConflict(X, T, Event), specificTarget(Att)]).
rule(r_op_nonGeopolitics1(C, T), neg(hasMotive(C, Att)), [targetCountry(T, Att),
 ↪  country(T), country(C), goodRelation(C, T)]).
rule(r_op_nonGeopolitics2(C, T), neg(hasMotive(C, Att)), [targetCountry(T, Att),
 ↪  country(T), country(C), goodRelation(T, C)]).

rule(r_op_grpPastTargets(Group,  Att), hasMotive(Group, Att), [target(T, Att),
 ↪  prominentGroup(Group), pastTargets(Group, Ts), member(T, Ts)]). %WEAK RULE

rule(r_op_claimResp0(X, Att), existingGroupClaimedResponsibility(X, Att),
 ↪  [claimedResponsibility(X, Att)]).
rule(r_op_claimResp1(X, Att),  neg(existingGroupClaimedResponsibility(X, Att)),
 ↪  [claimedResponsibility(X, Att),  noPriorHistory(X)]).

rule(r_op_social1(P, C), governmentLinked(P, C), [geolocatedInGovFacility(P,
 ↪  C)]).
rule(r_op_social2(P, C), governmentLinked(P, C), [publicCommentsRelatedToGov(P,
 ↪  C)]).

%% politicalIndustries are industries that are closely related to well-being of
 ↪  country/sensitive to national interests
rule(r_op_context(economic, Att),  contextOfAttack(economic, Att), [target(T,
 ↪  Att), industry(Ind, T), normalIndustry(Ind)]).
rule(r_op_context(political, Att), contextOfAttack(political, Att), [target(T,
 ↪  Att),  country(T)]).
rule(r_op_context1(political, Att), contextOfAttack(political, Att), [target(T,
 ↪  Att),  industry(Ind, T), politicalIndustry(Ind)]).

%% Auxiliary rules
%% Y2 M2 is before Y1 M1 but recent enough (within 1 year)
rule(r_op_date(ongoing), dateApplicable(_, ongoing), []).
rule(r_op_date1(Y, M), dateApplicable([Y, M], [Y, M]), []).
rule(r_op_date2(Y, M1, M2), dateApplicable([Y, M1], [Y, M2]), [M2 < M1]).
rule(r_op_date3(Y1, Y2), dateApplicable([Y1, _], [Y2, _]), [Y2 < Y1, Y2 > (Y1 -
 ↪  2)]).

% preferences
rule(p1a_op(), prefer(r_op_ecMotive(C, T), r_op_nonGeopolitics1(C, T)), []).
rule(p1b_op(), prefer(r_op_ecMotive(C, T), r_op_nonGeopolitics2(C, T)), []).
rule(p2a_op(), prefer(r_op_conflict(C, T), r_op_nonGeopolitics1(C, T)), []).
rule(p2b_op(), prefer(r_op_conflict(C, T), r_op_nonGeopolitics2(C, T)), []).
rule(p3a_op(), prefer(r_op_conflict1(C, T), r_op_nonGeopolitics1(C, T)), []).
rule(p3b_op(), prefer(r_op_conflict1(C, T), r_op_nonGeopolitics2(C, T)), []).
rule(p4a_op(), prefer(r_op_pMotive(C, T), r_op_nonGeopolitics1(C, T)), []).
rule(p4b_op(), prefer(r_op_pMotive(C, T), r_op_nonGeopolitics2(C, T)), []).
rule(p5_op(), prefer(r_op_claimResp1(X, A), r_op_claimResp0(X, A)), []).
rule(p6_op(), prefer(r_op_noCapability2(X, Att),  r_op_hasCapability1(X,  Att)),
 ↪  []).
```

A.2.3 All strategic rules

Below we show all the strategic rules and their preferences that can be found in `str_rules.pl`.

```
abducible(notFromBlackMarket(_), []).

% helper rules
rule(r_str_emptyHasCap(Att), hasCapability([], Att),   []).
rule(r_str_allHaveCap([X|L], Att), hasCapability([X|L], Att), [\+ is_list(X),
 ↪  is_list(L), hasCapability(X, Att), hasCapability(L, Att)]).
rule(r_str_prominentGrpHasCap(X, Att), hasCapability(X, Att),
 ↪  [prominentGroup(X)]).

rule(r_str__claimedResp(X, Att), isCulprit(X, Att),
 ↪  [existingGroupClaimedResponsibility(X, Att)]).

% rules proving (neg)isCulprit
rule(r_str__motiveAndCapability(C, Att), isCulprit(C, Att), [hasMotive(C, Att),
 ↪  hasCapability(C, Att)]).
rule(r_str__aptGroupMotive(C, Att), isCulprit(C, Att), [prominentGroup(Group),
 ↪  groupOrigin(Group, C), country(C), isCulprit(Group, Att), hasMotive(C,
 ↪  Att)]).
rule(r_str__motiveAndLocation(C, Att), isCulprit(C, Att), [attackOrigin(C, Att),
 ↪  hasMotive(C, Att), country(C)]).
rule(r_str__loc(C, Att), isCulprit(C, Att), [attackOrigin(C, Att), country(C)]).
rule(r_str__social(C, Att), isCulprit(C, Att), [governmentLinked(P, C),
 ↪  country(C), identifiedIndividualInAttack(P, Att)]).
rule(r_str__linkedMalware(X, A1), isCulprit(X, A1), [malwareUsedInAttack(M1,
 ↪  A1), similar(M1, M2), malwareLinkedTo(M2, X), notFromBlackMarket(M1),
 ↪  notFromBlackMarket(M2)]).

rule(r_str__noEvidence(X, Att), neg(isCulprit(X, Att)), []).
rule(r_str__noHistory(X, Att), neg(isCulprit(X, Att)),
 ↪  [neg(existingGroupClaimedResponsibility(X, Att))]).
rule(r_str__negAttackOrigin(X, Att), neg(isCulprit(X, Att)),
 ↪  [neg(attackOrigin(X, Att))]).
rule(r_str__noCapability(X, Att), neg(isCulprit(X, Att)), [neg(hasCapability(X,
 ↪  Att))]).
rule(r_str__noMotive(X, Att), neg(isCulprit(X, Att)), [neg(hasMotive(X, Att))]).
rule(r_str__weakAttack(X, Att), neg(isCulprit(X, Att)), [hasResources(X),
 ↪  neg(requireHighResource(Att))]).
rule(r_str__targetItself1(X, Att), neg(isCulprit(X, Att)), [target(X, Att)]).
rule(r_str__targetItself2(X, Att), neg(isCulprit(X, Att)), [targetCountry(X,
 ↪  Att)]).

% preferences
```

```
rule(p0a(), prefer(r_str__claimedResp(X, A), r_str__noEvidence(X, A)), []).
 ↪  %With any evidence, we prefer to attribute the culprit accordingly
rule(p0b(), prefer(r_str__motiveAndCapability(X, A), r_str__noEvidence(X, A)),
 ↪  []).
rule(p0c(), prefer(r_str__aptGroupMotive(X, A), r_str__noEvidence(X, A)), []).
rule(p0d(), prefer(r_str__motiveAndLocation(X, A), r_str__noEvidence(X, A)),
 ↪  []).
rule(p0e(), prefer(r_str__loc(X, A), r_str__noEvidence(X, A)), []).
rule(p0f(), prefer(r_str__social(X, A), r_str__noEvidence(X, A)), []).
rule(p0g(), prefer(r_str__linkedMalware(X, A), r_str__noEvidence(X, A)), []).

rule(p6(), prefer(r_str__noCapability(X, A),  r_str__claimedResp(X, A)), []). %
 ↪  hacker group might claim responsibility for attack backed by nation state
rule(p8(), prefer(r_str__noCapability(X, A),  r_str__aptGroupMotive(X, A)), []).
rule(p9(), prefer(r_str__noCapability(X, A),  r_str__motiveAndLocation(X, A)),
 ↪  []).
rule(p10(), prefer(r_str__noCapability(X, A),  r_str__loc(X, A)), []).
rule(p11(), prefer(r_str__noCapability(X, A), r_str__social(X, A)), []). %
 ↪  social evidences e.g. twitter posts/ emails can be easily forged
rule(p12(), prefer(r_str__noCapability(X, A), r_str__linkedMalware(X, A)), []).
rule(p18(), prefer(r_str__linkedMalware(X, A), r_str__negAttackOrigin(X, A)),
 ↪  []).
rule(p19(), prefer(r_str__weakAttack(X, A),       r_str__aptGroupMotive(X, A)),
 ↪  []).
rule(p20(), prefer(r_str__weakAttack(X, A),       r_str__motiveAndCapability(X,
 ↪  A)), []).

rule(p21a(), prefer(r_str__targetItself1(X, Att), r_str__claimedResp(X, Att)),
 ↪  [specificTarget(Att)]).
rule(p21b(), prefer(r_str__targetItself1(X, Att), r_str__motiveAndCapability(X,
 ↪  Att)), [specificTarget(Att)]).
rule(p21c(), prefer(r_str__targetItself1(X, Att), r_str__aptGroupMotive(X,
 ↪  Att)),      [specificTarget(Att)]).
rule(p21d(), prefer(r_str__targetItself1(X, Att), r_str__motiveAndLocation(X,
 ↪  Att)),   [specificTarget(Att)]).
rule(p21e(), prefer(r_str__targetItself1(X, Att), r_str__loc(X, Att)),
 ↪  [specificTarget(Att)]).
rule(p21f(), prefer(r_str__targetItself1(X, Att), r_str__social(X, Att)),
 ↪  [specificTarget(Att)]).
rule(p21g(), prefer(r_str__targetItself1(X, Att), r_str__linkedMalware(X, Att)),
 ↪  [specificTarget(Att)]).

rule(p22a(), prefer(r_str__targetItself2(X, Att), r_str__claimedResp(X, Att)),
 ↪  [specificTarget(Att)]).
rule(p22b(), prefer(r_str__targetItself2(X, Att), r_str__motiveAndCapability(X,
 ↪  Att)), [specificTarget(Att)]).
rule(p22c(), prefer(r_str__targetItself2(X, Att), r_str__aptGroupMotive(X,
 ↪  Att)),      [specificTarget(Att)]).
rule(p22d(), prefer(r_str__targetItself2(X, Att), r_str__motiveAndLocation(X,
 ↪  Att)),   [specificTarget(Att)]).
rule(p22e(), prefer(r_str__targetItself2(X, Att), r_str__loc(X, Att)),
 ↪  [specificTarget(Att)]).
rule(p22f(), prefer(r_str__targetItself2(X, Att), r_str__social(X, Att)),
 ↪  [specificTarget(Att)]).
rule(p22g(), prefer(r_str__targetItself2(X, Att), r_str__linkedMalware(X, Att)),
 ↪  [specificTarget(Att)]).

```

```
rule(p23a(), prefer(r_str__linkedMalware(X, A), r_str__noHistory(X, A)), []).
rule(p23c(), prefer(r_str__linkedMalware(X, A), r_str__noMotive(X, A)), []).
rule(p23d(), prefer(r_str__linkedMalware(X, A), r_str__weakAttack(X, A)), []).
```

Appendix B

Attribution Cases Used to Extract Rules

In this section we list the real-world cyber attacks, together with their evidence[1] and reasoning rules that were extracted from each case. See Section 4.3 for the description of the rules and how they are combined together to derive the conclusion.

B.1 Stuxnet attack

Below we show the rules and evidences extracted from [12, 11]. See Section 2.1.1 for a summary of the cyber attack case.

Evidences

```
target(iranian_org,stuxnetattack).
industry(nuclear,iranian_org).
targetCountry(iran,stuxnetattack).
usesZeroDayVulnerabilities(stuxnet).
news(nuclear,iran,ongoing).
causeOfConflict(united_states, iran, nuclear).
causeOfConflict(israel, iran, nuclear).
attackPeriod(stuxnetattack,[2010,7]).
malwareUsedInAttack(stuxnet, stuxnetattack).
specificConfigInMalware(stuxnet).
infectionMethod(usb,stuxnet).
target(iran_nuclear_facilities, stuxnetattack).
industry(nuclear, iran_nuclear_facilities).
```

Rules used

```
% tech_rules.pl
rule(r_t_targetted(Att), specificTarget(Att),
 ↪  [malwareUsedInAttack(M, Att), specificConfigInMalware(M)]).
```

[1]Evidences are listed in Prolog style for presentation purposes.

```

rule(r_t_highSkill2(Att), highLevelSkill(Att),
 ↪          [malwareUsedInAttack(M, Att),
 ↪  usesZeroDayVulnerabilities(M)]).
rule(r_t_highResource1(Att), requireHighResource(Att),
 ↪                   [highLevelSkill(Att)]).

rule(r_op_context1(political, Att),  contextOfAttack(political,
 ↪  Att),  [target(T, Att),  industry(Ind, T),
 ↪  politicalIndustry(Ind)]).

% op_rules.pl
rule(r_op_conflict(X, T), hasMotive(X, Att), [targetCountry(T, Att),
 ↪  attackPeriod(Att, Date1), news(Event, T, Date2),
 ↪  dateApplicable(Date1, Date2), causeOfConflict(X, T, Event),
 ↪  specificTarget(Att)]).
rule(r_op_hasCapability2(X, Att), hasCapability(X, Att),
 ↪  [requireHighResource(Att), hasResources(X)]).
rule(r_op_hasResources2(X), hasResources(X), [cybersuperpower(X)]).

% str_rules.pl
rule(r_str__motiveAndCapability(C, Att), isCulprit(C, Att),
 ↪  [hasMotive(C, Att), hasCapability(C, Att)]).
```

B.2 APT1

Below we show the rules and evidences extracted from [56, 57]. See Section 2.1.1 for a summary of the cyber attack case.

Evidences

```
majorityIpOrigin(china, apt1). % many IPs detected, but the majority
 ↪  of them originated from china
sysLanguage(chinese, apt1). % attacker's system default language
 ↪  configuration detected from malware is chinese
firstLanguage(chinese, china).
infraUsed(apt1_infra, apt1).
infraRegisteredIn(china, apt1_infra). % infrastructure registered in
 ↪  china
hasEconomicMotive(china, infocomm). % china has economic motive to
 ↪  attack organizations in the infocomm industry
target(v, apt1). % the target of the apt1 attack are a group of
 ↪  victims 'v'
industry(infocomm, v). % this group of victims are part of the
 ↪  infocomm industry
highVolumeAttack(apt1).
```

```
longDurationAttack(apt1).
%'superhard' and 'dota' are handle names of individuals identified
 ↪  as part of the attackers
identifiedIndividualInAttack(superhard, apt1).
identifiedIndividualInAttack(dota , apt1).
% 'superhard' was geolocated to frequent one of the government
 ↪  facilities in china
geolocatedInGovFacility(superhard, china). %
% 'dota' released some comments on social media hinting that he was
 ↪  related to the chinese government
publicCommentsRelatedToGov(dota , china).
```

Rules used

```
% tech_rules.pl
rule(srcIP(X,Att), attackPossibleOrigin(X,Att),
 ↪  [majorityIpOrigin(X,Att)]).
rule(lang1(X,Att), attackPossibleOrigin(X,Att), [sysLanguage(L,
 ↪  Att), firstLanguage(L, X)]).
rule(infra(X,Att), attackPossibleOrigin(X,Att), [infraUsed(Infra,
 ↪  Att), infraRegisteredIn(X, Infra)]).

rule(highResource3(Att), requireHighResource(Att),
 ↪  [highVolumeAttack(Att),longDurationAttack(Att)]).

% op_rules.pl
rule(ecMotive(C,T), hasMotive(C, Att), [industry(T), target(T, Att),
 ↪  hasEconomicMotive(C, T), specificTarget(Att)]).
rule(r_op_hasResources2(X), hasResources(X), [cybersuperpower(X)]).

% str_rules.pl
rule(r_str__motiveAndLocation(C, Att), isCulprit(C, Att),
 ↪  [attackOrigin(C, Att), hasMotive(C, Att), country(C)]).
rule(r_str__motiveAndCapability(C, Att), isCulprit(C, Att),
 ↪  [hasMotive(C, Att), hasCapability(C, Att)]).
```

B.3 Sony hack

Below we show the rules and evidences extracted from [14, 15, 16]. See Section 2.1.1 for a summary of the cyber attack case.

Evidences

```
claimedResponsibility(guardiansOfPeace, sonyhack).
target(sony, sonyhack).
targetCountry(united_states, sonyhack).
news(theInterview, sony,[2013,10]).
attackPeriod(sonyhack,[2014,11]).
causeOfConflict(north_korea, sony, theInterview).
majorityIpOrigin(north_korea, sonyhack).
malwareUsedInAttack(trojanVolgmer, sonyhack).
malwareUsedInAttack(backdoorDestover, sonyhack).
```

Rules used

```
% tech_rules.pl
rule(r_t_srcIP2(X, Att), attackPossibleOrigin(X, Att),
 ↪  [majorityIpOrigin(X, Att)]).

% op_rules.pl
rule(r_op_claimResp0(X, Att), existingGroupClaimedResponsibility(X,
 ↪  Att), [claimedResponsibility(X, Att)]).
rule(r_op_conflict1(X, T), hasMotive(X, Att), [target(T, Att),
 ↪  attackPeriod(Att, Date1), news(Event, T, Date2),
 ↪  dateApplicable(Date1, Date2), causeOfConflict(X, T, Event),
 ↪  specificTarget(Att)]).

% str_rules.pl
rule(r_str__claimedResp(X, Att), isCulprit(X, Att),
 ↪  [existingGroupClaimedResponsibility(X, Att)]).
rule(r_str__linkedMalware(X, A1), isCulprit(X, A1),
 ↪  [malwareUsedInAttack(M1, A1), similar(M1, M2),
 ↪  malwareLinkedTo(M2, X), notFromBlackMarket(M1),
 ↪  notFromBlackMarket(M2)]).
```

B.4 US bank hack

Iran was blamed for the 2012 denial of service (Dos) attacks on banks in United States, causing websites of many banks to suffer slowdowns and even unreachable for many customers [57]. Experts said that attackers were "crafting their own private clouds, by creating networks of individual machines or by stealing resources wholesale from poorly maintained corporate clouds" [56]. These web hosting services were infected by a malware called *Itsoknoproblembro*. Botnets can usually be traced back to a specific control centre, but *Itsoknoproblembro* was crafted in a way that made it untraceable. The skills required to carry out such a large-scale sophisticated attack indicated that the attack being backed up by a large organization with sufficiently large capability and resources. Furthermore, economic sanctions and online attacks (Flame, Duqu and Stuxnet) by the United States against Iran constitutes a strong motive for Iran to carry out the attack. Consequently, despite independent hacktivist group, *Izz ad-Din al-Qassam Cyber Fighters*, claiming responsibility for the attacks, investigators believed that the claim was in fact a disguise for the Iranian government [56], who was the true source of the attack.

Below we show the rules and evidences extracted from [56, 57].

Evidences

```prolog
targetCountry(usa ,  usbankhack).
imposedSanctions(usa, iran, [2012, 2]). % usa imposed a new wave of sanctions on iran on 2012 Feb
hijackCorporateClouds(usbankhack). % in the us bank hack, corporate cloud servers were hijacked
sophisticatedMalware(itsoknoproblembro).
malwareUsedInAttack(itsoknoproblembro , usbankhack).
attackPeriod(usbankhack , [2012, 9]). % the attack happened in 2012 Sept
target(us_banks, usbankhack). % target of the attack are US banks
industry(banking, us_banks). % the US banks belong to the banking industry
```

Rules used

```prolog
% tech_rules.pl
rule(highSkill1, highLevelSkill(Att), [hijackCorporateClouds(Att)]).
rule(highSkill2, highLevelSkill(Att), [malwareUsedInAttack(M, Att), sophisticatedMalware(M)]).

rule(highResource1, requireHighResource(Att), [highLevelSkill(Att)]).

% op_rules.pl
rule(hasCapability1, hasCapability(_X, Att), [neg(requireHighResource(Att))]).
rule(hasCapability2, hasCapability(X, Att), [requireHighResource(Att), hasResources(X)]).
rule(noCapability, neg(hasCapability(X, Att)), [requireHighResource(Att), neg(hasResources(X))]).

rule(pMotive(C,T), hasMotive(C, Att), [targetCountry(T, Att), attackPeriod(Att, Date1), hasPoliticalMotive(C, T, Date2), dateApplicable(Date1, Date2), specificTarget(Att)]).
rule(pMotive(C,T,Date), hasPoliticalMotive(C, T, Date), [imposedSanctions(T, C, Date)]).

```

```
% str_rules.pl
rule(motiveAndCapability(C,Att),isCulprit(C,Att),
↪  [hasMotive(C,Att),hasCapability(C,Att)]).
```

B.5 Gauss attack

"Gauss" is a virus discovered in 2011. It targeted the middle east region, mostly concentrated on attacking Lebanese banks, stealing data and spying on bank transactions [58]. The Gauss attack is widely believed to be a state-sponsored attack backed up by the United States. Gauss is found to be very similar to Flame, Duqu and Stuxnet. It shares several similarities with its closest relative, Flame. These similarities includes using a similar architecture, same encoded names, similar command and control servers [59]. This similarity has led to the conclusion that Gauss "comes from the same factory or factories" as Flame, Duqu and Stuxnet [60].

Below we show the rules and evidences extracted from [59, 60, 58].

Evidences

```
% expected: equationGroup (linkedMalware)
malwareUsedInAttack(gauss, gaussattack).
sophisticatedMalware(gauss).
targetCountry(lebanon, gaussattack). % Note: other countries were
↪  attacked too, but focus is on lebanon
infectionMethod(usb, gauss). % gauss malware infects machines by
↪  USB
% the control and command used by gauss was easily fingerprinted.
↪  Unique fingerprint detection of C&C traffic can be used by
↪  anti-virus software to flag the malware
commandAndControlEasilyFingerprinted(gauss).
% 'gowin7' and 'secuurity' are command and control servers used by
↪  gauss
ccServer(gowin7, gauss).
ccServer(secuurity, gauss).
% the domains are registered under the name 'Adolph Dybevek' and
↪  under the address 'Prinsen Gate 6'
domainRegisteredDetails(gowin7, adolph_dybevek,  prinsen_gate_6).
domainRegisteredDetails(secuurity ,  adolph_dybevek,
↪  prinsen_gate_6).
attackPeriod(gaussattack, [2011, 9]). % the attack occured in 2011
↪  Sept

% background evidence (flame malware)
malwareLinkedTo(flame, equationGrp). % 'flame' malware said to by
↪  made by 'equation group'
target(middleeast , flameattack).
malwareUsedInAttack(flame, flameattack).
```

```
ccServer(gowin7, flame).
ccServer(secuurity, flame).
domainRegisteredDetails(gowin7, adolph_dybevek, prinsen_gate_6).
domainRegisteredDetails(secuurity, adolph_dybevek, prinsen_gate_6)
```

Rules used

```
% tech_rules.pl
rule(bm, notFromBlackMarket(M),
 ↪  [infectionMethod(usb,M),commandAndControlEasilyFingerprinted(M)]).

% 2 malwares are similar to each other if they have similar C&C
 ↪  servers
rule(similar,similar(M1, M2), [similarCCServer(M1, M2), M1 \= M2]).
rule(simCC1, similarCCServer(M1, M2), [ccServer(S, M1), ccServer(S,
 ↪  M2)]).
rule(simCC2, similarCCServer(M1, M2), [ccServer(S1, M1),
 ↪  ccServer(S2, M2), S1 \= S2, domainRegisteredDetails(S1,_,A),
 ↪  domainRegisteredDetails(S2,_,A)]).
rule(simCC3, similarCCServer(M1, M2), [ccServer(S1, M1),
 ↪  ccServer(S2, M2), S1 \= S2, domainRegisteredDetails(S1,Name,_),
 ↪  domainRegisteredDetails(S2,Name,_)]).

% str_rules.pl
rule(linkedMalware(X,A1), isCulprit(X,A1),
 ↪  [malwareUsedInAttack(M1,A1), similar(M1,M2),
 ↪  malwareLinkedTo(M2,X), notFromBlackMarket(M1),
 ↪  notFromBlackMarket(M2)]).
```

B.6 Wannacry attack

WannaCry

The recent NHS ransomware attack has also been attributed to a nation state. The UK home office "can be as sure as possible" and "it is widely believed in the community and across a number of countries that North Korea had taken this role" [61]. Prior to the NHS incident, there have been reports of North Korean hackers using malicious software to extort Bitcoin [62]. Investigators have also discovered that the WannaCry code shares connections to previous attacks attributed to Lazarus Group, a North-Korean-linked group [63]. Despite this, some experts say that the lack of sophistication of WannaCry indicates a lack of structure thus "rather than being a nation-state campaign, it looked more like a 'typical' cyber-crime campaign that sought to enrich its operators" [64].

Below we show the rules and evidences extracted from [62, 63, 64].

Evidences

```
malwareUsedInAttack(wannacry, wannacryattack).
malwareUsedInAttack(trojanAlphanc, wannacryattack).
malwareModifiedFrom(trojanAlphanc, backdoorDuuzer).
malwareUsedInAttack(trojanBravonc, wannacryattack).
% 'backdoorBravonc' and 'infostealerFakepude' use similar code
↪  obfuscation techniques
similarCodeObfuscation(backdoorBravonc, infostealerFakepude).
%  'wannacry' and 'backdoorCantopee' have some shared code
sharedCode(wannacry, backdoorCantopee).
% wannacry attack occured on 2017 May
attackPeriod(wannacryattack, [2017, 5]).
% wannacry attack did not have a specific target, multiple countries
↪  and industries were attacked
neg(specificTarget(wannacryattack)).
% NHS was one of the targets
target(nhs, wannacryattack).
targetCountry(uk, wannacryattack).
```

Rules used

```
% tech_rules.pl
rule(r_t_similar1(M1, M2), similar(M1, M2),
↪  [similarCodeObfuscation(M1, M2)]).
rule(r_t_similar2(M1, M2), similar(M1, M2), [sharedCode(M1, M2)]).

% str_rules.pl
rule(r_str__linkedMalware(X, A1), isCulprit(X, A1),
↪  [malwareUsedInAttack(M1, A1), similar(M1, M2),
↪  malwareLinkedTo(M2, X), notFromBlackMarket(M1),
↪  notFromBlackMarket(M2)]).
```

Appendix C

Adding Strategic Rule Preferences in Gorgias

In this chapter, we explain why it is difficult to add strategic rule preferences for different culprits in Gorgias.

We first give an example of what we would like to achieve. Given two strategic rules that derives `isCulprit(X, Att)` (Listing 20), we will like to be able to add a preference as shown in Listing 21, to prefer the first rule (over the second) if we are able to derive different culprits using the two rules (if `X \= Y`).

Listing 20 Two strategic rules both deriving `isCulprit(X, Att)`

```
rule(r_str__motiveAndCapability(X, Att), isCulprit(X, Att),
 ↪  [hasMotive(X, Att), hasCapability(X, Att)]).
rule(r_str__loc(X, Att), isCulprit(X, Att), [attackOrigin(X, Att),
 ↪  country(X)]).
```

Listing 21 Prefer the rule `r_str__motiveAndCapability(X, Att)` over `r_str__loc(Y, Att)` if the rules derive different culprits

```
rule(p, prefer(r_str__motiveAndCapability(X, Att), r_str__loc(Y,
 ↪  Att)), []) :- X \= Y.
```

Next, we explain the difficulty involved in doing this in Gorgias. Normally, preferences can be written directly into the Prolog file and be processed by the Gorgias framework. By writing rules as the ones described below,

```
rule(p, prefer(rule1, rule2), []).

% L is any literal
rule(rule1, L, [body1, ...]).
rule(rule2, neg(L), [body2, ...]).
```

we are able to make a preference of `rule1` over `rule2` when both can be proven. In order for the preference rule to fire, the two rules in the preference must be in *conflict*. In the Gorgias framework, conflict is expressed by the following Prolog code:

```
 1  % from gorgias-src-0.6d-Visual-20Feb2018/lib/gorgias.pl
 2  conflict(ass(L), ass(NL)) :-
 3          complement(L,NL).
 4  conflict(Sig1, Sig2) :-
 5          rule(Sig1, Head1, _),
 6          rule(Sig2, Head2, _),
 7          complement(Head1, Head2).
 8
 9  complement(prefer(Sig1, Sig2), prefer(Sig2, Sig1)):- !.
10  complement(neg(L), L):- !.
11  complement(L, neg(L)).
```

While `neg(L)` and `L` are predefined to be in conflict (line 10), the predicates `isCulprit(X, Att)` and `isCulprit(Y, Att)` when $X \neq Y$ are not in conflict by default. We can include the predicate `complement(isCulprit(X, Att), isCulprit(Y, Att)) :- X \= Y` to denote the predicates `isCulprit(X, Att)` and `isCulprit(Y, Att)` as complementary. However, because of the way `conflict(Sig1, Sig2)` is defined, the body of the rule does not get called, and the arguments in the head are not ground when proving `conflict(Sig1, Sig2)`.

To clarify what we mean by this, let us look at the following simple example.

```
:- compile('lib/gorgias2').
:- compile('ext/lpwnf2').

complement(p(X), p(Y)) :- X \= Y.
rule(r1(X), p(X),[a(X)]).
rule(r2(X), neg(p(X)),[]).

rule(f1, a(1),[]).
rule(f2, a(2),[]).
rule(p, prefer(r1(1), r1(2))).
```

This example has two rules, two facts and one preference. Without the preference, both `p(1)` and `p(2)` can be proven by `r1(1)` and `r1(2)`. To add a preference for `r1(1)` over `r1(2)`, we have added the extra `complement/2` predicate to make `p(1)` and `p(2)` complementary. If we execute `prove([p(X)],D)` now, we would expect to only see the result `X = 1`. However, what we actually see is that we can still prove both `p(1)` and `p(2)`.

```
| ?- prove([p(X)],D).
X = 1,
D = [f1, r1(1)] ;
X = 2,
D = [f2, r1(2)] ;
false.
```

To reveal the cause of this problem, we execute `conflict(Sig1,Sig2)` as shown below:

```
| ?- conflict(Sig1,Sig2).
Sig1 = ass(prefer(_4056, _4058)),
Sig2 = ass(prefer(_4058, _4056)) ;
Sig1 = r1(_4048),
Sig2 = r2(_4048) ;
Sig1 = r2(_4048),
Sig2 = r1(_4048) ;
false.
```

Breaking down the result, the first pair of `Sig1` and `Sig2` are just any two preferences `prefer(L1,L2)` and `prefer(L2,L1)` for any literals `L1,L2` are in conflict. The second and third pair shows that `r1(X)` and `r2(X)` are in conflict (since their heads are the negation of each other). However, we do not see `Sig1 = r1(1), Sig2 = r1(2)`, which is required for the preference relation `prefer(r1(1), r1(2))` to fire. As mentioned before, this is the problem that arises due to the fact that the arguments in the rule names are not ground when Gorgias computes `conflict(Sig1, Sig2)`.

Due to this problem, we have implemented preferences between strategic rules when different culprits are derived in Java, instead of in Gorgias.

Appendix D

Other Implementation Details

In this chapter, we include the explanations of some implementation details of *ABR* that are not essential for understanding how *ABR* works.

D.1 Query limit

In *ABR*, we set the limit to 100 for normal queries. This means that we will only show the first 100 results obtained from Prolog. During execution, we rarely hit the limit of 100, it is just a precautionary limit, for example if the user accidentally adds recursive rules. For the negative derivations, we limit the number to 10. The purpose of the negative derivations is to help analyst to spot conflicting arguments and add preferences to resolve them. Displaying up to 10 negative derivations for each unique culprit `X` is sufficient to achieve that purpose, and can reduce the processing time for each execution.

D.2 Extraction of argument tree

By default, `visual_prove/2` prints out the argument tree in standard output. To extract that into *ABR*, we can either change the default implementation of Gorgias or save the output to a file and parse the file. We chose the latter due to the complexity of the Gorgias framework implementation. First, we have to pipe the Prolog output to another file. To achieve this, we used the built-in SWI-Prolog predicates `tell(+SrcDest)` and `told/0`[1]. These queries opens `SrcDest` to use it as the current output stream and closes the current output stream respectively. After execution, the file `visual.log` contains n argument trees, one for each of the n solutions. Since within each argument tree there are no new lines, we are able to split the n argument trees by splitting on "`\n\n`".

D.3 Construction of DerivationNode from derivation example

To illustrate how the DerivationNode is constructed from the derivation string, we show an example of how a DerivationNode is constructed from the following deriva-

[1]http://www.swi-prolog.org/pldoc/man?predicate=tell/1

tion:

```
[p4a_t(), bg1(), case_example5_f2(), case_example5_f3(),
↪  r_t_srcIP1(yourCountry,example5),
↪  r_t_attackOrigin(yourCountry,example5),
↪  r_str__loc(yourCountry,example5)]
```

The 1st Term `p4a_t()` is a preference, so we continue onto the next term. `bg1()` is an evidence, we create a new DerivationNode and push it onto the stack (Table D.1).

Table D.1: Stack after pushing 2nd Term

DerivationNode d1 rulename: bg1() result: country(myCountry) children: []

`case_example5_f2()` and `case_example5_f3()` are, again, evidences, so we create new DerivationNodes and add them onto the stack (Table D.2).

Table D.2: Stack after pushing 4th Term

DerivationNode d3 rulename: case_example5_f2() result: ipGeoloc(yourCountry,[103,1,206,100]) children: []
DerivationNode d2 rulename: case_example5_f3() result: attackSourceIP([103,1,206,100],example5) children: []
DerivationNode d1 rulename: bg1() result: country(yourCountry) children: []

The 5th Term (`r_t_srcIP1(yourCountry, example5)`) is a technical rule. The rule is:

```
rule(r_t_srcIP1(X,Att), attackPossibleOrigin(X,Att),
↪  [attackSourceIP(IP,Att),ipGeoloc(X,IP)]).
```

Since the body predicates correspond to the top 2 DerivationNodes on the stack, we pop them off the stack and add them as children to the new DerivationNode created from the 5th Term (Table D.3).

Table D.3: Stack after pushing 5th Term

DerivationNode d4 rulename: r_t_srcIP1(yourCountry, example5) result: attackPossibleOrigin(yourCountry, example5) children: [d2, d3]
DerivationNode d1 rulename: bg1() result: country(yourCountry) children: []

The 6th and 7th terms are also rules, technical and strategic rules respectively. They are processed similarly to give the final stack as follows in Table D.4.

Table D.4: Final state of the stack

DerivationNode d6 rulename: r_str__loc(yourCountry, example5) result: isCulprit(yourCountry, example5) children: [d5, d1]

At the end, the stack only contains one DerivationNode, which is the root node with result `isCulprit(X, Att)`. We can then use the `java-graphviz` package to visualise the DerivationNode. The figure generated from DerivationNode `d6` is shown in Figure D.1.

D.4 Construction of DerivationNode from argument tree example

We illustrate how the DerivationNode is created from an argument tree, with a walk-through example. We show below the argument tree that will be used in this example:

```
[bg1(), case_example2b_f10(), case_example2b_f9(),
 ↪  r_t_srcIP1(yourCountry,example2b), r_t_attackOrigin(yourCountry,example2b),
 ↪  r_str__loc(yourCountry,example2b)]  {DEFENSE}
|___[r_t_nonOrigin(yourCountry,example2b),
 ↪  r_t_noLocEvidence(yourCountry,example2b), p3_t()]
|   |___[r_t_srcIP1(yourCountry,example2b), case_example2b_f10(),
 ↪  case_example2b_f9(), p4a_t()]  {DEFENSE}
|___[r_str__targetItself2(yourCountry,example2b), case_example2b_f2(), p22e(),
 ↪  ass(specificTarget(example2b))]
    |___[r_op_notTargetted(example2b), case_example2b_f2b(), case_example2b_f2()]
     ↪  {DEFENSE}
```

Starting from the last line of the string, the first node is pushed into the stack (Table D.5).

Figure D.1: Graph visualisation generated from DerivationNode d6

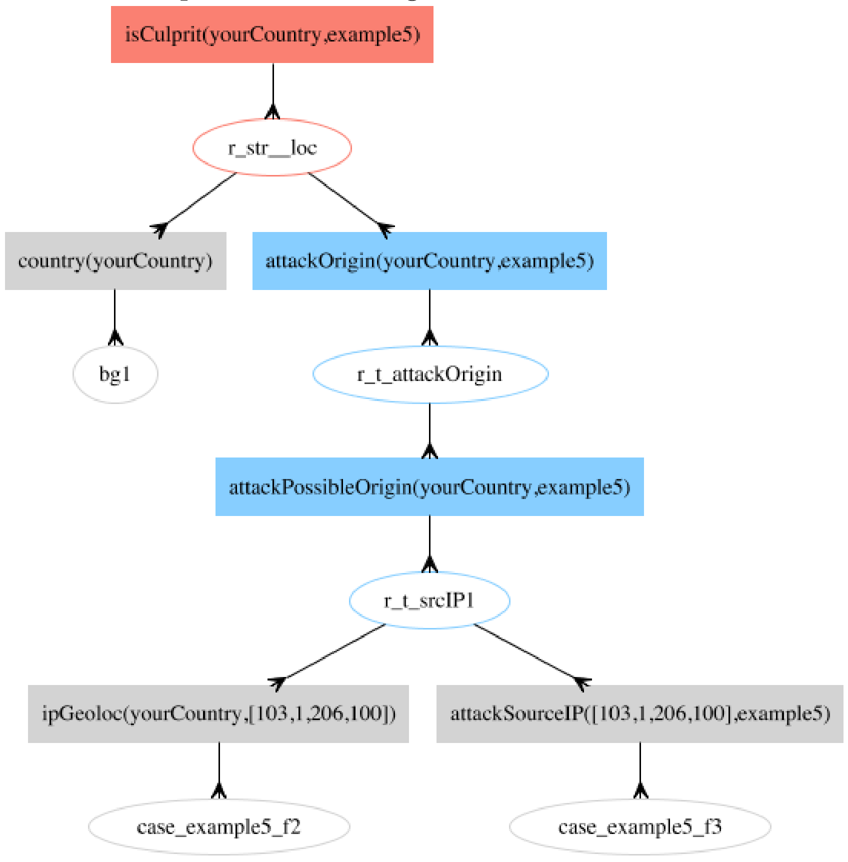

isCulprit(yourCountry,example5)
r_str__loc
country(yourCountry)
attackOrigin(yourCountry,example5)
bg1
r_t_attackOrigin
attackPossibleOrigin(yourCountry,example5)
r_t_srcIP1
ipGeoloc(yourCountry,[103,1,206,100])
attackSourceIP([103,1,206,100],example5)
case_example5_f2
case_example5_f3

Table D.5: Stack at node 1

Node n1
Derivation: [r_op_notTargetted(example2b), case_example2b_f2b(),...
level: 2
type: DEFENSE
child: []

The level of the node on top of the stack is 2 and the level of the current node is 1, so the top node is popped off the stack and added the the new node, which is pushed onto the stack (Table D.6).

Table D.6: Stack at node 2

Node n2
Derivation: [r_str__targetItself2(yourCountry,example2b),...
level: 1
type: ATTACK
child: [n1]

The new node has level 2, the top node is not a child of the current node, so the new node is simply pushed onto the stack (Table D.7).

Table D.7: Stack at node 4

Node n3 Derivation: [r_t_srcIP1(yourCountry,example2b),... level: 2 type: DEFENSE child: []
Node n2 Derivation: [r_str__targetItself2(yourCountry,example2b),... level: 1 type: ATTACK child: [n1]

The new node has level 1, the top node is a child node, so it is popped off again and added as a child node (Table D.8).

Table D.8: Stack at node 5

Node n4 Derivation: [r_t_nonOrigin(yourCountry,example2b),... level: 1 type: ATTACK child: [n3]
Node n2 Derivation: [r_str__targetItself2(yourCountry,example2b),... level: 1 type: ATTACK child: [n1]

The last node has level 0. Both nodes on the stack are its children so they are both popped off and added as children nodes. The final node left is the conclusion node, also the root node of the argument tree (Table D.9).

Table D.9: Final state of stack

Node n5 Derivation: [bg1(), case_example2b_f10(),... level: 1 type: DEFENSE child: [n2, n4]