A Novel Information Sharing Model using Binary Decision Diagrams for Redundancy, Shadowing, Generalisation and Correlation

May 10, 2013

Abstract

This paper outlines a novel information sharing method using Binary Decision Diagrams (BBDs). It is inspired by the work of Al-Shaer and Hamed, who applied BDDs into the modelling of network firewalls. This is applied into an information sharing policy system which optimizes the search of redundancy, shadowing, generalisation and correlation within information sharing rules.

1 Introduction

1.1 Context

INFORMATION sharing between police and community partners forms the cornerstone of effective Intelligence-Led Policing (ILP). ILP has its origins in the Kent Policing Model (KPM) [1] of the early 1990s amid an environment of economic uncertainty, constraints on policing resources and on public spending in general [2]. However, even in such an adversarial environment, the KPM acted as a catalyst in creating a culture of enhanced and integrated public services.

Today, economic constraints present an environment reminiscent to that of the early 1990s, again requiring further efficiencies in the use of limited resources. Although advancements in cloud-computing, the increased prevalence of service-oriented architectures and evolving governance standards offer possible solutions, they also present new challenges. These include concerns over the security of sensitive and confidential information; the integrity of this information; and the effective control over who has access to this information.

A core issue concerning many information sharing architectures is the management and control of information within and between organisational boundaries. Typically, the sharing of information is defined in security policies, which determine how information is managed within an organisation. However, with increasing inter-organisational information sharing and the growing need for collaborative working, the task of managing the number of possible ways that information can be shared and aggregated is becoming increasingly complex. A *Trust Framework* and *Governance Engine* are two fundamental components in resolving the issue of effective information sharing.

The Trust Framework, within the scope of this paper, refers to the contract between partners that formalises the abstracted principles of any sharing of information, for example, in line with statutory and legal directives. This contract includes the roles, services and relationships that are involved in the sharing of information. The Governance Engine forms a complementary component to the Trust Framework. It has the task of interpreting the abstracted principles defined by the Trust Framework and applying these to a specific instance of information sharing. Hence, the Trusted Framework defines the environment for any sharing of information, while the Governance Engine inspects the specifics of an instance of information sharing to verify that they comply with the principles stipulated in the Trust Framework.

2 Novel Policy Definition and Modelling

2.1 Introduction

THIS chapter introduces the concept of information sharing as a rule-based system. It proposes a framework used to formalise information sharing policies into a structure similar to that of firewall and access-control list rules. Comparisons are drawn between firewall access control lists, a *traditional* rule-based system, and the proposed framework for information sharing, highlighting the advantages and potential issues encountered in formalising information sharing policies. Major advantages of this approach are that the evaluation techniques used to model access-control lists can be applied to information sharing policies, and formal methods can be used to analyse these policies for errors and anomalies. This approach, however, presents the challenge that while a formal structure is enforced for information sharing policies, often the meaning and intent of the original policy is obscured or lost. The approach presented in this chapter preserves the linguistic intent of the rule, while allowing for a structure that is sufficiently rigid so as to be applicable for formal analysis. The concept of Binary Decision Diagrams (BDDs) is introduced and their ability to model Boolean functions is described. The goal is to illustrate the ability of BDDs to represent Boolean functions with canonicity, and the exploitation of this ability to represent packet-filtering access lists. The latter part of the chapter demonstrates how techniques used to model access-lists can effectively be extended to model information sharing policies. It should also be noted that this chapter forms part of a patent application and should be considered commercially sensitive.

2.2 Information sharing as a Rule-Based System

A fundamental difficulty with rule-based systems including governance, legal-reasoning and firewall systems, is that as the number and complexity of the policies which govern system behaviour increases, so too do the numbers of errors occurring within the rule-base, as illustrated by Avishai Wool in [3], Figure 1. Hence, with increasingly complex systems, which are gradually becoming more interconnected, the task of managing, auditing and deterministically predicting the effects of elaborate, inter-dependent policies becomes all the more demanding.

This paper focuses on inherent similarities between policy-based systems, especially concentrating on policy-based information sharing as analogous to rule-based packet-filtering network firewall configuration, is illustrated in Figure 2. In this analogy, the packet is equated to a request for information to be shared, and the firewall is equated to a *request-filtering* agent. The agent's function is to either permit or deny requests based on specified policies, just as a network firewall would permit or deny a packet based on its rules configuration.

This paper views mechanisms for sharing-information as policy-driven systems, where the sharing of information is driven by well-defined policies. This allows judgements to be made about the behaviour of the system based on the analysis of its policies, as the system's behaviour will be deterministic. Further, where a system has a large corpus of policies governing its behaviour, the use of a well-defined policy structure allows

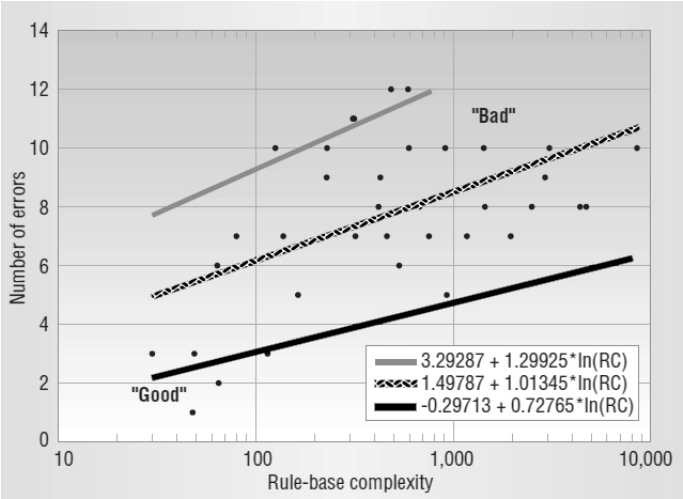

Figure 1: Number of errors as a function of rule-base complexity [3].

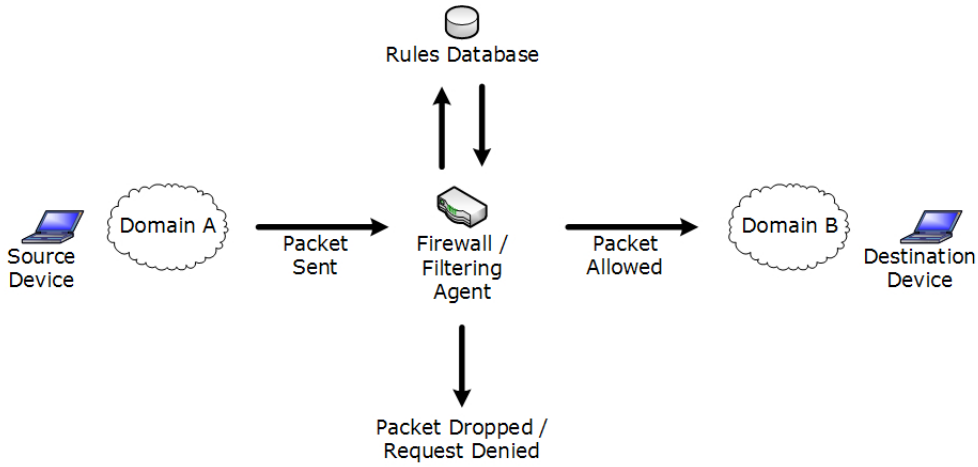

Figure 2: Representation of request filtering agent as analogous to a firewall.

such a corpus to be managed efficiently. This view also allows the use of Morris Sloman's well-known definition in [4] regarding policies as the principles governing the decisions undertaken within a system which affect the behaviour of the system. Hence, policies can be said to play a central role in governing the behaviour of a system. However, there is an essential need to separate high-level policy definitions from low-level policy implementations. This separation process presents two fundamental challenges:

1. The high-level policy should be human-readable so that it is able to effectively capture the intention of the policy originator.
2. The translation of high-level definitions to low-level implementations must be explicit, unambiguous and deterministic.

A number of approaches seeking to address these challenges have been proposed over time. However, a fundamental difficulty with these approaches is that they are not designed from the perspective of the policy originator, but that of the lower-level policy implementer. Hence, these approaches invariably require significant proficiency and expertise of the lower-level policy implementation mechanisms before they can be used effectively. This presents a further challenge from an information sharing perspective. Practitioners such as law-enforcement officers, social-workers and health professionals, usually do not readily possess this expertise. Yet their roles involve responsibilities that are subject to, and require everyday interaction with, such policies. Therefore, this paper seeks to address these challenges by approaching high-level policy syntax from a restricted natural-language perspective. By using natural-language for high-level definitions, the task of interpreting the aim of policy statements becomes intuitive and accessible to practitioners. This approach is necessary in order to effectively capture the goals of the policy, as intended by the policy originator. A natural-language approach essentially allows a high-level policy statement to be human-understandable and allows non-experts to read a policy statement and easily understand the intent of the policy originator. One considerable obstacle with the use of free-form natural-language is that a statement can have different meanings and interpretations which make it unsuitable for strict policy definition. As illustrated by Meyer [5], even carefully written natural language specifications are ambiguous and open to interpretation. Therefore, in order to address this obstacle, this paper uses a restricted natural-language syntax, developed specifically for information sharing scenarios, which allows clear, explicit and human-readable policy definitions, while avoiding any ambiguity in policy interpretation or translation.

2.2.1 Firewall Rules Syntax

Firewall rules can be thought of as the lower-level implementation of higher-level network security policies which define which packets should be permitted or denied by the network. Firewall rules are used to implement these policies and usually include a number of essential components based on packet header fields, such as protocol-type, source IP address, source port, destination IP address, and destination port, in order to classify packets. When a packet matches a rule, the decision to permit or deny it is defined by the action component of the rule. Usually, a number of firewall rules, structured in sets called access control lists (ACLs) [6], are needed to effectively implement security policies. If no matching rules are found, the firewall will use a default, usually deny, action. For simplicity, this paper only considers the following fields:

- Protocol: This identifies the protocol of the packet. For simplicity, this paper only considers the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).
- Source IP address: This is made up of the standard IP address segments, each segment consisting of a binary octet represented in decimal by a number in the range 0 to 255.
- Source Port: This defines a port number in Internet Protocol (IP) suite, ranging from 0 to 65,535.
- Destination IP address: This follows the same format as the Source IP address.
- Destination Port: This follows the same format as the Source Port number.
- Action: This defines the action that the firewall must take, accept or reject, when a packet matches the defined rule.

Hazelhurst et al. demonstrate in [7] how integers from these firewall rule fields can be represented as bit vectors and, hence, as variables of Binary Decision Diagrams (BDDs). For example, a firewall rule which denies TCP packets from source port 80 of the IP address 140.192.37.20, destined for port 21 of IP address 192.168.21.20, could be of the form illustrated in Listing 1:

Listing 1: Example of a firewall rule.

```
TCP 140.192.37.20 80 192.168.21.20 21 Deny
```

2.2.2 Information Sharing Policy Syntax

A restricted natural-language syntax has been developed as part of this paper specifically for police and community partner information sharing scenarios, as outlined by Uthmani et al. in [8]. This syntax is loosely inspired from firewall policy rules, as detailed by Al-Shaer and Hamed in [9] and outlined in Listing 2.

Listing 2: Firewall rule syntax as defined in [9].

```
<rule order> <protocol> <source ip> <source port> <
   destination ip> <destination port> <permission>
```

Al-Shaer and Hamed describe this syntax as consisting of five tuples, where each tuple is a required field in the header of a packet and is used by a packet-filtering firewall to determine how the packet should be handled. Similarly, using the analogy illustrated in Figure 2 where a packet is compared to a request for information and a firewall is compared to a request filtering agent, a generic information sharing rule syntax is developed based on fields outlined in Listing 3.

Listing 3: information sharing rule syntax as defined in [8].

```
<permission> <requester> <relationship> <action> <
   attribute>  <object> <context> <owner> <
   compliance> <risk-level>
```

A simple information sharing rule can, therefore, be reduced to a ten field syntax where the contents of these fields are defined as:

- <permission> is part of the rule syntax which indicates the action of the rule. This defines whether a request matching the rule criteria will be permitted or denied access.
- <requester> identifies the source of a request as a specific individual or the membership of a certain role.
- <relationship> defines the relationship which exists between a requester and an owner with respect to an object.
- <action> defines the action a requester is permitted to perform on an object attribute, such as create, read, update or delete (CRUD).
- <attribute> is a unit of information describing an object.
- <object> refers to any entity about which information is held.
- <context> identifies the reason why the information is being shared. The context also governs the level of access and permissions associated with information exchange and, hence, impacts the priority accorded to information requests.
- <owner> species a role with sufficient privileges to manage all aspects of an information element. The owner has the authority to allow or deny access to an information element, as required by legislation and defined responsibilities.
- <compliance> refers to legislative requirements that affect the exchange of information, as well as data anonymisation and sanitisation requirements.
- <risk-level>, within a crime context, refers to the crime risk-level associated with the information sharing rule. The crime risk-level is calculated using the methodology described in Chapter **??**.

information sharing policies can, therefore, be generalised and reduced to be defined using a ten-field syntax, as outlined in Listing 3. Each rule represented using this syntax may be thought of as a higher-level information sharing policy. Hazelhurst et al. demonstrate in [7] how integers from a specific field can be represented as bit vectors and, hence, as variables in Binary Decision Diagrams (BDDs). As outlined earlier, a policy in a rule-based system such as a firewall which denies TCP packets from source port 80 of the IP address 140.192.37.20, destined for port 21 of IP address 192.168.21.20, could be of the form illustrated in Listing 4:

Listing 4: Example of a firewall rule.

```
TCP 140.192.37.20 80 192.168.21.20 21 Deny
```

Similarly, in an information sharing scenario, a policy could exist between the *Domestic Violence Unit* of *Police Force A* and the *Records Unit* of *Child Protection Agency B*. In this example, the policy permits the chief-investigator of a child-protection investigation to read the health history record of a child. The health history record is maintained by the *Records Unit* of *Child Protection Agency B* and needs the approval of that unit's *Records Admin* before it can be shared. Further, the policy additionally stipulates that the investigator must be of the rank of *Sergeant* within the *Domestic Violence Unit* of *Police Force A*, that the request for information must be in compliance with the Data Protection Act and that the crime risk-level is *3*. The risk-level is then calculated using the methodology described in Chapter **??**. As illustrated in Table **??**,

the crime severity score of a physical injury to a child is 3894. This score correlates to crime risk-level *3*, as shown in Table **??**.

The described policy initially appears complex but can be effectively modelled using the syntax outlined in Listing 3. The result is illustrated in Listing 5 where the‘<’ and ‘>’ symbols denote the beginning and end, respectively, of a relevant field in the policy:

Listing 5: Example of an information sharing policy.

```
This policy <permits> a <Sergeant> from the <
   Domestic Violence Unit> of <Police Force A>,
   with a <Chief Investigator> relation in a <Child
    Protection Investigation>, to request to  <read
   > a <child> <Health History Record> from the
<Records Admin> of the <Records Unit> of <Child
   Protection Agency B> under compliance of the <
   Data Protection Act> with risk-level <3>
```

2.3 Binary Decision Diagrams (BDDs)

2.3.1 Boolean Expressions

The examples which follow in this chapter refer to the classical calculus definitions for Boolean expressions. These consist of Boolean variables (x, y, ...), constants, *true* (1) and *false* (0), and operators *negation* ($\neg$), *conjunction* ($\wedge$), *disjunction* ($\vee$), *implication* ($\Rightarrow$) and *bi-implication* ($\Leftrightarrow$). Henrik Andersen provides an abstract syntax to Boolean expressions in [10], and provides details on the formal grammar used to generate the above expressions. Truth tables for the Boolean operators negation, conjunction, disjunction, implication and bi-implication are given in Tables 1, 2, 3, 4 and 5 respectively.

Table 1: Truth table for the negation ($\neg$) Boolean operator.

Input p	Output $\neg p$
0	1
1	0

Table 2: Truth table for the conjunction ($\wedge$) Boolean operator.

Input p	Input q	Output $p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

It should be noted that the expression $(p \Rightarrow q)$, as denoted in Table 4, is logically equivalent to the expression $(\neg p \vee q)$. This relationship is formalised in Equation 1. Further, it should also be noted that the expression $(p \Leftrightarrow q)$, as denoted in Table 5, is logically equivalent to the expression $((p \wedge q) \vee (\neg p \wedge \neg q))$. This relationship is formalised in Equation 2.

$$p \Rightarrow q \equiv \neg p \vee q \tag{1}$$

$$p \Leftrightarrow q \equiv (p \wedge q) \vee (\neg p \wedge \neg q) \tag{2}$$

Table 3: Truth table for the disjunction ($\vee$) Boolean operator.

Input p	Input q	Output $p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

Table 4: Truth table for the implication ($\Rightarrow$) Boolean operator.

Input p	Input q	Output $p \Rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

Table 5: Truth table for the bi-implication ($\Leftrightarrow$) Boolean operator.

Input p	Input q	Output $p \Leftrightarrow q$
0	0	1
0	1	0
1	0	0
1	1	1

2.3.2 Binary Decision Diagrams (BDDs) and Canonicity

Binary Decision Diagrams (BDDs) are rooted, directed, acyclic graphs originally proposed by Lee [11] in 1959 and Akers [12] in 1978 to graphically represent Boolean functions. BDDs originate from binary decision trees which are rooted, directed trees that can be used to represent Boolean functions. For example, the decision tree illustrated in Figure 3 represents the Boolean function $f(x, y) = (x \vee y)$. The concept behind this form of representation is that each non-terminal node (circle) in the decision tree denotes a variable. In the example illustrated in Figure 3, the variables are x and y. The node refers to a test of the variable's binary value, 0 or 1, with the edges of the node representing the paths taken by either possible value. The path represented by the dashed (low) edge corresponds to the case where the variable is assigned a 0, and the path represented by the solid (high) edge corresponds to the case where the variable is assigned a 1. The bottom (square) terminal-nodes of the tree represent the Boolean constants 0 and 1. Hence, the value of any Boolean function may be evaluated for any given number of variables by starting at the root (top) of the tree and following the path at each node, as determined from the value of the variable that the node represents. This process is repeated until a terminal-node (bottom) is reached. The value of the Boolean function, either a 0 or a 1, is represented by the value of the terminal node.

A difficulty with representing Boolean functions with decision trees is that if the function contains a large number of variables, then the decision tree representing that function will also be very large. Figure 4, for example, represents the binary decision tree for the function $f(x, y, z) = (x \vee y \vee z)$. A comparison of Figure 3, which represents a Boolean function with two variables, x and y, with Figure 4, which represents a Boolean function with three variables, x, y, and z, illustrates that there is an exponential relationship between the number of variables in a function and the number of nodes in the decision tree which represents that function. With increasing numbers of variables, therefore, the size of the decision trees used to represent functions increases

exponentially. As can be expected, the decision trees of complex Boolean functions can quickly become very large and difficult to use.

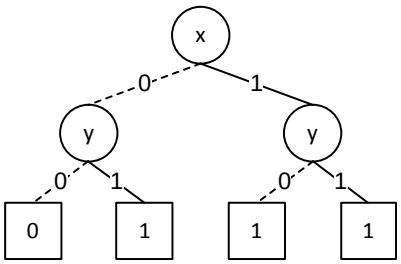

Figure 3: Binary decision diagram for the function $f(x, y) = (x \vee y)$.

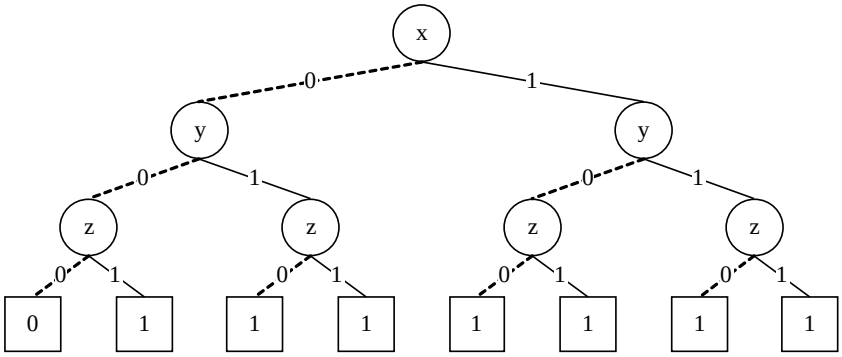

Figure 4: Binary decision diagram for the function $f(x, y, z) = (x \vee y \vee z)$.

2.3.3 Reduced Ordered Binary Decision Diagrams (ROBDDs)

In 1986, Randal Bryant proposed a solution to this problem in [13] by introducing algorithms for reducing binary trees and ordering the variables in a function. The process of reduction consists of merging any isomorphic sub-graphs for the decision tree. Any parent node which has child-nodes that are isomorphic is considered redundant and is removed. Applying this process to the decision tree for the Boolean function $f(x, y) = (x \vee y)$, as illustrated in Figure 3, it is evident that if the first node, x, is 1, then the value of the second node, y, has no effect on the terminal node value of the Boolean function: whether y is 0 or 1, the value of the terminal nodes is 1. This means that where the node x is 1, child-nodes of y are isomorphic. Node y can then be considered redundant here and removed. The result is the reduced decision tree as illustrated in Figure 5. Similarly, applying the reduction process to the decision tree for the Boolean function $f(x, y, z) = (x \vee y \vee z)$, illustrated in Figure 4, yields the reduced decision tree shown in Figure 6. Reduced decision trees allow a more compact representation of Boolean expressions than non-reduced decision trees.

Bryant also highlighted in [13] that the size of a decision tree for a given function is dependent on the ordering of the variables in that decision tree. For example, the

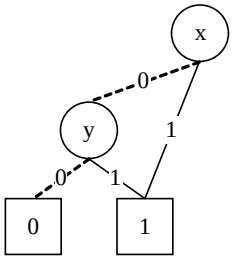

Figure 5: Reduced Binary Decision Diagram for the function $f(x, y) = (x \vee y)$.

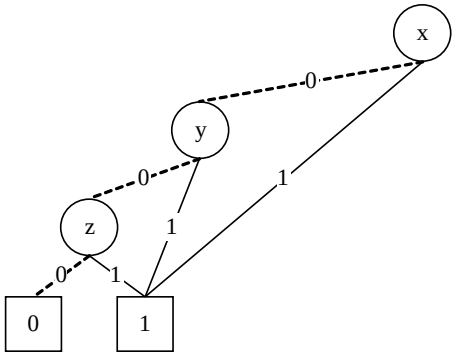

Figure 6: Reduced Binary Decision Diagram for the function $f(x, y, z) = (x \vee y \vee z)$.

decision tree for the Boolean function $f(w, x, y, z) = (w \wedge x) \vee (y \wedge z)$, given a variable ordering of w, x, y, z, is illustrated in Figure 7. If the variable ordering for the same function was now changed to w, z, y, x, the resultant decision tree will be more complicated, as illustrated in Figure 8. Hence, an optimal variable ordering will produce the simplest, and therefore smallest, decision tree for a given function, while sub-optimal orderings will produce larger and more complex decision trees for the same function. However, as shown by Bollig and Wegener in [14], determining the optimal variable ordering for a Boolean function is an NP-complete problem that often requires trial and error or expert knowledge of domain-specific ordering strategies.

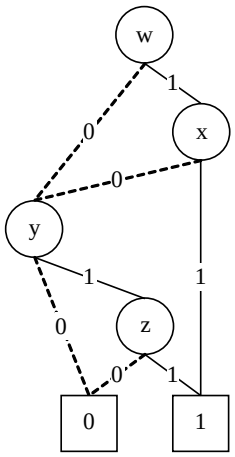

Figure 7: Reduced Binary Decision Diagram for the function $f(w, x, y, z) = (w \wedge x) \vee (y \wedge z)$ with variable ordering of w, x, y, z.

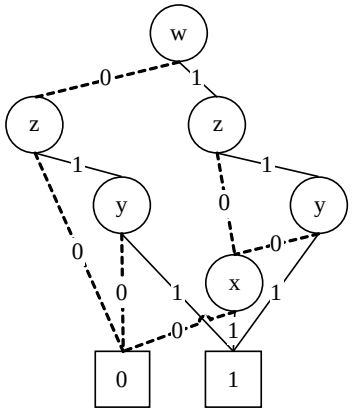

Figure 8: Reduced Binary Decision Diagram for the function $f(w, x, y, z) = (w \wedge x) \vee (y \wedge z)$ with variable ordering of w, z, y, x.

Decision trees which have been reduced and ordered are referred to as Reduced Ordered Binary Decision Diagrams (ROBDDs), or commonly shortened to just Binary Decision Diagrams (BDDs). A key property of the reduction and ordering restrictions introduced by Bryant is that the resulting BDDs are canonical [15]. This means that

the BDD for any Boolean function, for a defined variable ordering, will always be isomorphic. This property has made BDDs ideal for use in formal equivalence checking. In the electronic design automation process, for example, BDDs are frequently used to formally prove that two circuit design representations exhibit the same behaviour.

2.4 Modelling Sets of Rules and Policies Using BDDs

A novelty of this paper is to exploit the unique properties of Binary Decision Diagrams (BDD) to model complex sets of policies, in a form that is readily machine-executable, and to extend these to the information sharing domain. The work of Hazelhurst et al. [7] with firewalls identified key constituent fields in access-list rules and translated these into bit vectors representing BDD variables. This paper applies a similar methodology to information sharing where a set of information sharing policies can be modelled as a decision diagram, once a specific variable ordering scheme has been selected. The modelling of a set of policies as a BDD provides a number of significant advantages, including providing an efficient lookup mechanism for an information sharing request, as well as providing a graphical representation of the overall policy set. As rule sets become larger and more complex, they become difficult to interpret and maintain [16]. Modification of the rule set, by either adding new rules or removing existing ones, or even changing the order of rules has a significant impact on the behaviour of the policy-based system. Hence, analysis and validation of large, complex rule sets is essential in ensuring that high-level directives are enforced. Further, exploiting the formal equivalence checking ability of BDDs, and the fact that they can canonically represent Boolean functions, multiple sets of policies can be compared to ensure that they have the same behaviour or identify areas where they behave differently. Large and complex rule sets, represented as BDDs, can, therefore, be efficiently modelled, analysed and validated.

2.4.1 Translation of Firewall Rules

Each field in the firewall rule is first translated into a bit sequence in order to be represented as a BDD. The following list describes how firewall rule fields for the example in Listing 1 are translated into bit sequences:

- Protocol: Only one Boolean variable is needed to represent this field since it can only be either TCP or UDP. TCP will be denoted by 1 and UDP by 0.
- Source IP address: The source IP address field consists of four binary octets represented in decimal by a number in the range 0 to 255 and, hence, requires 32 Boolean variables to translate to bit sequence. The four octets of the source IP address in the example, 140.192.37.20, are translated as follows:
  - 140: 1000 1100
  - 192: 1100 0000
  - 37: 0010 0101
  - 20: 0001 0100

- Source port: The source port field represents ports ranging from 0 to 65535, hence, 16 Boolean variables are required to translate this field. The source port field in the example, port 80, is translated as 0000 0000 0101 0000.
- Destination IP address: The destination IP address field has the same format as the Source IP address field and also requires 32 Boolean variables. The four octets of the destination IP address in the example, 192.168.21.20, are translated as follows:
  - 192: 1100 0000
  - 168: 1010 1000
  - 21: 0001 0101
  - 20: 0001 0100
- Destination port: The destination port field has the same format as the source port field and also requires 16 Boolean variables. The destination port field in the example, port 21, is translated as 0000 0000 0001 0101.
- Action: The action field can either be 'Permit', denoted by 1, or 'Deny', denoted by 0.

The firewall rule shown in Listing 1 can, hence, be translated into a Boolean vector of 97 variables and, therefore, be represented as a BDD. Similarly, as Hazelhurst et al. describe in [17], the process used to convert a single firewall rule into a Boolean function, shown above, can be recursively applied to all the rules in a rule set. The resulting Boolean functions corresponding to each individual rule can then be linked together using a logical disjunction operator ($\vee$) for *permit* rules or a logical conjunction ($\wedge$) operator for *deny* rules, yielding a single Boolean function which represents the entire firewall rule set. This function can then be represented as a BDD which models the complete rule set. A firewall can then classify and filter packets using the BDD by simply extracting relevant fields in the packet header and checking if they satisfy the BDD.

2.4.2 Translation of Information Sharing Policies

Each field in an information sharing policy is first translated into a bit sequence in order to be represented as a BDD. The following list describes how policy fields for the example in Listing 5 are translated into bit sequences:

- Permission: A single Boolean variable is needed to represent this field since it can either be *permit* and denoted by 1, or *deny* and denoted by 0.
- Requester, Owner: The Requester field, similar to the Owner field, comprises a hierarchical, X.500-inspired form. It can be tailored to any organisational hierarchy but, for the purposes of illustration, represents a *Domain.Organisation.Unit.Role* structure in this example, representing the respective fields for the *Requester* and *Owner*. Since this simplified example has a single Requester, and a single Owner, two binary variables are needed to represent the domain, organisation, unit and roles of each:
  - Requester Domain: 01

  - Requester Organisation: 01
  - Requester Unit: 01
  - Requester Role: 01
  - Owner Domain: 10
  - Owner Organisation: 10
  - Owner Unit: 10
  - Owner Role: 10

- Relation: The relation field is represented as flat in this example, solely for illustration. Since this example contains only one relation, it is simply represented by a single 1. In a 'live' system, this field will usually be hierarchically defined to reflect the desired granularity. The structure and specific hierarchy used for the definition of this field is entirely flexible, and can be tailored to the needs of the organisation.
- Context: Similar to the relation field described above, the context field is also represented as flat in this example, solely for illustration. Since this example contains only one context, it is simply represented by a single 1. In a live system, this field will also be defined in a hierarchical fashion to reflect desired granularity. The structure and specific hierarchy used for the definition of this field is entirely flexible and can be tailored to the needs of the organisation.
- Action: The action field is used to differentiate between possible actions, such as create, read, update and delete, which a requester may be allowed to perform. Since this example contains the *Read* action, it is simply represented by a single 1.
- Object: Similar to the relation field described above, the object field is also represented as flat in this example, solely for illustration. Since this example contains only one object, it is simply represented by a single 1. In a live system, this field will also be defined in a hierarchical fashion to reflect desired granularity. The structure and specific hierarchy used for the definition of this field is entirely flexible and can be tailored to the needs of the organisation.
- Attribute: Similar to the relation field described above, the attribute field is also represented as flat in this example, solely for illustration. Since this example contains only one attribute, it is simply represented by a single 1. In a live system, this field will also be defined in a hierarchical fashion to reflect desired granularity. The structure and specific hierarchy used for the definition of this field is entirely flexible and can be tailored to the needs of the organisation.

The information sharing policy shown in Listing 5 can, hence, be translated into a Boolean vector of 97 variables, as shown in Listing 6.

Listing 6: information sharing policy from Listing 5 translated into Boolean vector.

```
<permission=1> <requester=01010101> <relationship
   =1> <action=1> <attribute=1> <object=1> <context
   =1> <owner=10101010> <compliance=1>
```

Since the policy is now in a Boolean vector form, it can be represented as a BDD. Similarly, as Hazelhurst et al. describe in [17] regarding firewall rules, the process used to convert a single information sharing rule into a Boolean function, shown above, can be recursively applied to all the rules in a rule set. The resulting Boolean functions corresponding to each individual rule can then be linked together using a logical disjunction operator ($\vee$) for *permit* rules or a logical conjunction ($\wedge$) operator for *deny* rules, yielding a single Boolean function which represents the entire set of policies. This function can then be represented as a BDD which then models the entire policy set. The policy-filtering agent can then classify and filter information using the BDD by simply extracting relevant fields described above and checking if they satisfy the BDD.

2.5 Rule and Policy Translation Examples

2.5.1 Firewall Rule Set Translation Example

This example describes in detail the steps needed to translate an access list into a BDD. Table 6 shows a portion of a sample access list, adapted from [18]. A '*' or 'Any' is used in the access list to denote redundant fields, or redundant portions of fields, for IP address octets and ports, respectively. Redundant fields are not translated into binary as they represent variables that are not evaluated by the BDD and, hence, do not form part of the Boolean function. Where an entire field is redundant, it is entirely excluded from binary representation and where only a portion of a field is redundant, only the relevant portion is translated while the redundant portions are shown using Xs. If no matching rule is found, the firewall defaults to a deny policy.

In this example, the access list rules shown in Table 6 are initially translated into their binary form and then expressed as logical conjunctions of their relevant fields. Each rule is also expressed as an *if-then* statement to clearly define how it will be processed. Finally, each rule is represented as a BDD. It must be noted that the motivation in expressing each rule as a BDD is simply to illustrate the relatively linear form that the diagram takes, indicating that each field in the rule is processed in a sequential fashion. This is contrasted by the final BDD representing the access list as a whole. This BDD is not linear and, hence, illustrates how some fields are logically prioritised over others.

Table 6: Portion of access list adapted from [18]

Rule	Protocol	Source Address	Source Port	Destination Address	Destination Port	Action
1	TCP	*.*.*.*	Any	161.120.33.41	25	Permit
2	TCP	140.192.37.30	Any	*.*.*.*	21	Deny
3	TCP	*.*.*.*	Any	161.120.33.*	21	Deny
4	TCP	140.192.37.*	Any	*.*.*.*	21	Permit
Default Policy: Deny						

Rule1

Protocol (TCP)	:	1
Source IP (*.*.*.*)	:	not checked by BDD
Source Port (Any)	:	not checked by BDD
Destination IP (161.120.33.41)	:	161 : 1010 0001
	:	120 : 0111 1000
	:	33 : 0010 0001
	:	41 : 0010 1001
Destination Port (25)	:	11001
Action (Permit)	:	1

The Boolean function corresponding to Rule1, ignoring redundant fields, is a logical conjunction of all of the above fields in the format shown in Listing 7. Listing 8 represents Rule1 expressed logically as an *if-then* conditional statement and Figure 9 illustrates Rule1 represented as a BDD.

Listing 7: Logical format of Rule1.

```
Permit: Protocol ∧ Destination IP ∧ Destination
   Port
```

Listing 8: Rule1 expressed as an if-then conditional statement.

```
if      (Protocol = 1) ∧
        (Destination IP =
           10100001011110000010000100101001) ∧
        (Destination Port = 11001),
then    (Action = Permit)
```

Rule2

Protocol (TCP)	:	1
Source IP (140.192.37.30)	:	140 : 1000 1100
	:	192 : 1100 0000
	:	37 : 0010 0101
	:	30 : 0001 1110
Source Port (Any)	:	not checked by BDD
Destination IP (*.*.*.*)	:	not checked by BDD
Destination Port (21)	:	10101
Action (Deny)	:	0

The Boolean function corresponding to Rule2, ignoring redundant fields, is a logical conjunction of all of the above fields, in the format shown in Listing 9. Listing 10 represents Rule2 expressed logically as an if-then conditional statement and Figure 10 illustrates Rule2 represented as a BDD.

Listing 9: Logical format of Rule2.

```
Deny: Protocol ∧ Source IP ∧ Destination Port
```

Listing 10: Rule2 expressed as an if-then conditional statement.

```
if      (Protocol = 1) ∧
```

```
        (Source IP =
           10001100110000000010010100011110) ∧
        (Destination Port = 10101),
then    (Action = Deny)
```

Rule3

Protocol (TCP)	:	1
Source IP (*.*.*.*)	:	not checked by BDD
Source Port (Any)	:	not checked by BDD
Destination IP (161.120.33.*)	:	161 : 1010 0001
	:	120 : 0111 1000
	:	33 : 0010 0001
	:	* : XXXX XXXX
Destination Port (21)	:	10101
Action (Deny)	:	0

The Boolean function corresponding to Rule3, ignoring redundant fields, is a logical conjunction of all of the above fields, in the format shown in Listing 11. Listing 12 represents Rule3 expressed logically as an if-then conditional statement and Figure 11 illustrates Rule3 represented as a BDD.

Listing 11: Logical format of Rule3.

```
Deny: Protocol ∧ Destination IP ∧ Destination Port
```

Listing 12: Rule3 expressed as an if-then conditional statement.

```
if      (Protocol = 1) ∧
        (Destination IP = 101000010111100000100001
           XXXXXXXX) ∧
        (Destination Port = 10101),
then    (Action = Deny)
```

Rule4

Protocol (TCP)	:	1
Source IP (140.192.37.30)	:	140 : 1000 1100
	:	192 : 1100 0000
	:	37 : 0010 0101
	:	* : XXXX XXXX
Source Port (Any)	:	not checked by BDD
Destination IP (*.*.*.*)	:	not checked by BDD
Destination Port (21)	:	10101
Action (Permit)	:	1

The Boolean function corresponding to Rule4, ignoring redundant fields, is a logical conjunction of all of the above fields, in the format shown in Listing 13. Listing 14 represents Rule4 expressed logically as an if-then conditional statement and Figure 12 illustrates Rule4 represented as a BDD.

Listing 13: Logical format of Rule4.

```
Deny: Protocol ∧ Source IP ∧ Destination Port
```

Listing 14: Rule4 expressed as an if-then conditional statement.

```
if      (Protocol = 1) ∧
```

```
        (Source IP = 10001100110000000100101
           XXXXXXXX) ∧
        (Destination Port = 10101),
then    (Action = Permit)
```

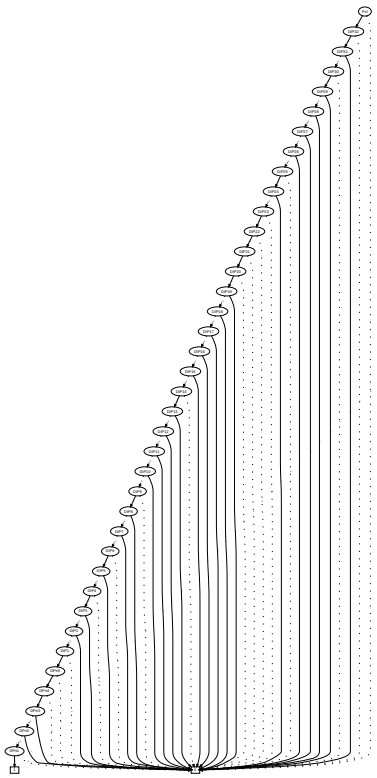

Figure 9: BDD representing Rule1.

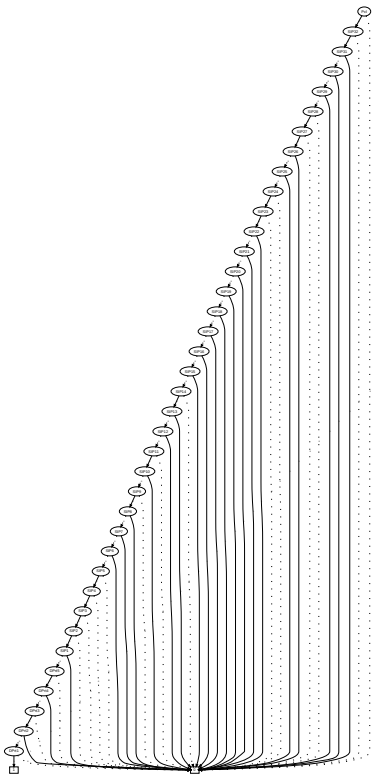

Figure 10: BDD representing Rule2.

2.5.2 Modelling the Complete Firewall Rule Set

The security policy modelled by the example access list shown in Table 6 is enforced through a combination of Rule1, Rule2, Rule3 and Rule4. Listing 15 illustrates the logical sequence in which the rules are enforced.

Listing 15: Logical sequence of enforced rules.

```
if      Rule1 ∨ (¬Rule2 ∧ ¬Rule3 ∧ Rule4)
then    Permit
else    Deny
```

As shown, a packet is permitted if its header fields match Rule1. If Rule1 is not matched, the packet can only be permitted if Rule2 *and* Rule3 are *not* matched and Rule4 *is* matched. If no rules are matched, the default policy is applied and the packet is denied. Since Rule2 and Rule3 are Deny rules, they are prefixed with a '¬' to show that these rules must *not* be matched in order for the packet to be accepted. The BDD of

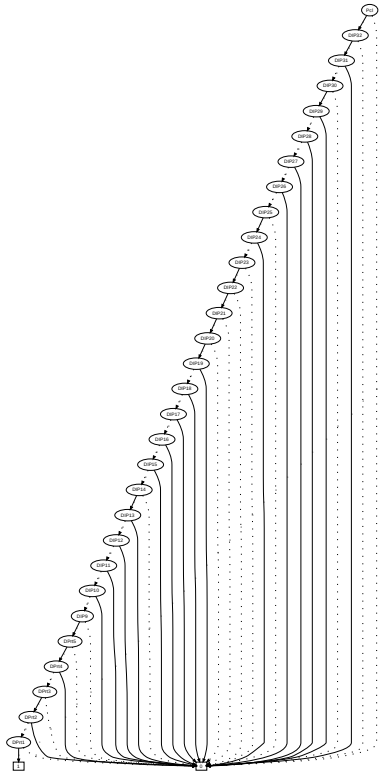

Figure 11: BDD representing Rule3.

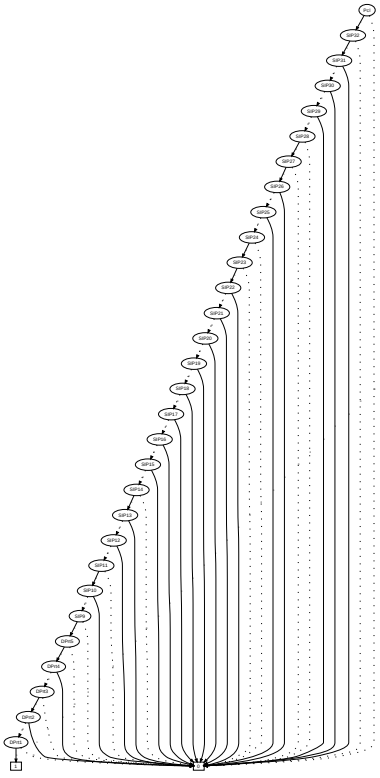

Figure 12: BDD representing Rule4.

the combined rules is illustrated in Figure 13. Since this BDD is not linear, it illustrates how some fields are prioritised over others and where alternate paths can be taken to arrive at the same decision.

2.5.3 information sharing Policy Set Translation Example

This example describes in detail the steps needed to translate a set of information sharing policies to a BDD. List 16 shows a sample list of policies. A '*' or 'Any' is used to denote redundant fields, or redundant portions of fields. Redundant fields are not translated into binary as they represent variables that are not evaluated by the BDD and, hence, do not form part of the Boolean function. Where an entire field is redundant, it is entirely excluded from the binary representation and where only a portion of a field is redundant, only the relevant portion is translated while the redundant portions are shown using 'Xs'.

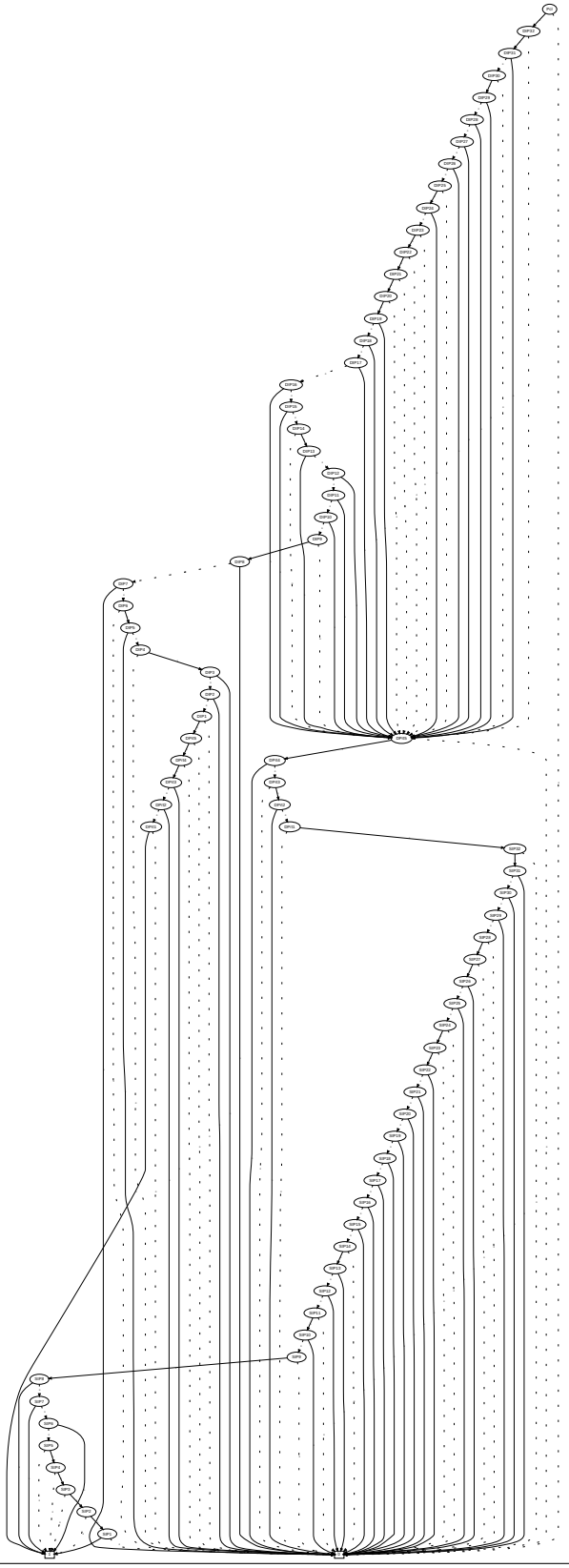

Figure 13: Rule1 ∨ (¬Rule2 ∧ ¬Rule3 ∧ Rule4).

Listing 16: Sample list of information sharing policies.

```
Policy 1: This policy <permits> <ANY> requester,
   with <ANY> relation in <ANY> context, to request
    to <read> a <child>  <Health History Record>
   from the <Records Admin> of the    <Records Unit
   > of <Child Protection Agency B> under
   compliance of the <Data Protection Act>

Policy 2: This policy <denies> a <Sergeant> from
   the <Domestic Violence Unit> of <Police Force A
   >, with <ANY> relation in <ANY> context, to
   request to <read> a <child> <Case File Record>
   from <ANY> owner under compliance of the <Data
   Protection Act>

Policy 3: This policy <denies> <ANY> requester,
   with <ANY> relation in <ANY> context, to request
    to <read> a <child>  <Case File Record> from <
   ANY> role of the <Records Unit> of <Child
   Protection Agency B> under compliance of the <
   Data Protection Act>

Policy 4: This policy <permits> <ANY> role from the
    <Domestic Violence Unit> of <Police Force A>,
   with <ANY> relation in <ANY> context, to request
    to <read> a <childs> <Case File Record> from <
   ANY> owner under compliance of the <Data
   Protection Act>
```

Policy1

Compliance (DPA)	:	1
Requester (Any)	:	not checked by BDD
Relation (Any)	:	not checked by BDD
Context (Any)	:	not checked by BDD
Object (Child)	:	1
Attribute (Health History Record)	:	01
Owner (SocCare.CPA-B.RecUnit.RecAdmin)	:	SocCare : 10
	:	CPA-B: 10
	:	RecUnit : 10
	:	RecAdmin : 10
Action (Permit)	:	1

The Boolean function corresponding to Policy1, ignoring redundant fields, is a logical conjunction of all of the above fields in the format shown in Listing 17. Listing 18 represents Policy1 expressed logically as an if-then conditional statement and Figure 14 illustrates Policy1 as a BDD. Figure 15 represents a simplification of Figure 14, highlighting the effective constituent parts of Policy1.

Listing 17: Logical format of Policy1.

```
Permit: Compliance ∧ Owner ∧ Object ∧ Attribute
```

Listing 18: Rule1 expressed as an if-then conditional statement.

```
if      (Compliance = 1) ∧
        (Owner = 10101010) ∧
        (Object = 1) ∧
        (Attribute = 01),
then    (Action = Permit)
```

Policy2

Compliance (DPA)	:	1
Requester (Police.Force-A.DVU.Sergeant)	:	Police : 01
	:	Force-A : 01
	:	DVU : 01
	:	Sergeant : 01
Relation (Any)	:	not checked by BDD
Context (Any)	:	not checked by BDD
Object (Child)	:	1
Attribute (Case File Record)	:	10
Owner (Any)	:	not checked by BDD
Action (Deny)	:	0

The Boolean function corresponding to Policy2, ignoring redundant fields, is a logical conjunction of all of the above fields, in the format shown in Listing 19. Listing 20 represents Policy2 expressed logically as an if-then conditional statement and Figure 16 illustrates Policy2 as a BDD. Figure 17 represents a simplification of Figure 16, highlighting the effective constituent parts of Policy2.

Listing 19: Logical format of Policy2.

```
Deny: Compliance ∧ Requester ∧ Object ∧ Attribute
```

Listing 20: Policy2 expressed as an if-then conditional statement.

```
if      (Compliance = 1) ∧
        (Requester = 01010101) ∧
        (Object = 1) ∧
        (Attribute = 10),
then    (Action = Deny)
```

Policy3

Compliance (DPA)	:	1
Requester (Any)	:	not checked by BDD
Relation (Any)	:	not checked by BDD
Context (Any)	:	not checked by BDD
Object (Child)	:	1
Attribute (Case File Record)	:	10
Owner (SocCare.CPA-B.RecUnit.Any)	:	SocCare : 10
	:	CPA-B: 10
	:	RecUnit : 10
	:	Any : X
Action (Deny)	:	0

The Boolean function corresponding to Policy3, ignoring redundant fields, is a logical conjunction of all of the above fields, in the format shown in Listing 21. Listing 22 represents Policy3 expressed logically as an if-then conditional statement and Figure 18 illustrates Policy3 represented as a BDD. Figure 19 represents a simplification of Figure 18, highlighting the effective constituent parts of Policy3.

Listing 21: Logical format of Policy3.

```
Deny: Compliance ∧ Owner ∧ Object ∧ Attribute
```

Listing 22: Policy3 expressed as an if-then conditional statement.

```
if      (Compliance = 1) ∧
        (Owner = 101010X) ∧
        (Object= 1) ∧
        (Attribute = 1),
then    (Action = Deny)
```

Policy4

Compliance (DPA)	:	1
Requester (Police.Force-A.DVU.Any)	:	Police : 01
	:	Force-A : 01
	:	DVU : 01
	:	Any : X
Relation (Any)	:	not checked by BDD
Context (Any)	:	not checked by BDD
Object (Child)	:	1
Attribute (File Record)	:	10
Owner (Any)	:	not checked by BDD
Action (Permit)	:	1

The Boolean function corresponding to Policy4, ignoring redundant fields, is a logical conjunction of all of the above fields, in the format shown in Listing 23. Listing 24 represents Policy4 expressed logically as an if-then conditional statement and Figure 20 illustrates Policy4 represented as a BDD. Figure 21 represents a simplification of Figure 20, highlighting the effective constituent parts of Policy4.

Listing 23: Logical format of Policy4.

```
Permit: Compliance ∧ Requester ∧ Object ∧ Attribute
```

Listing 24: Rule4 expressed as an if-then conditional statement.

```
if      (Compliance = 1) ∧
        (Requester = 010101X) ∧
        (Object = 1) ∧
        (Attribute = 10),
then    (Action = Deny)
```

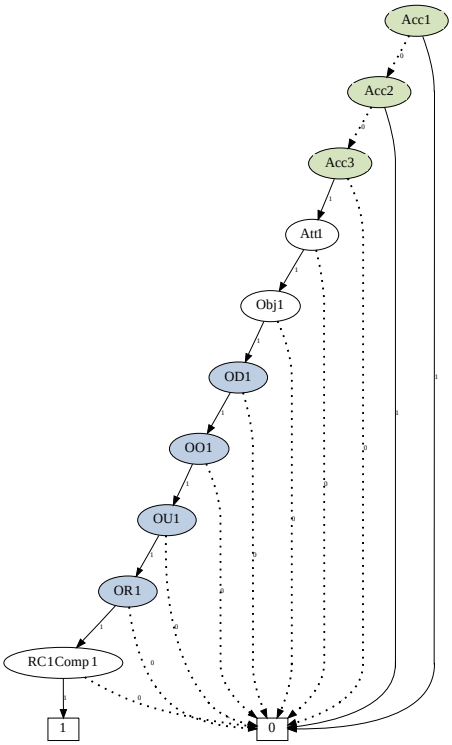

Figure 14: BDD representing Policy1.

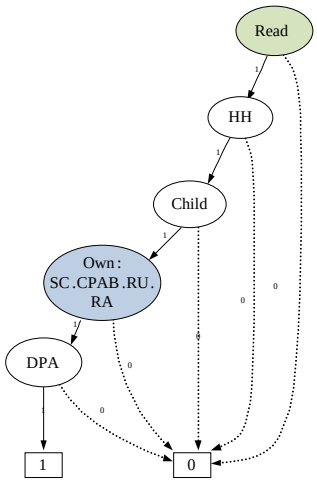

Figure 15: Simplification of Policy1 BDD.

2.5.4 Modelling the Complete information sharing Policy Set

The information sharing policy modelled by the sample list shown in List 16 is enforced through a combination of Policy1, Policy2, Policy3 and Policy4. Listing 25 illustrates the logical sequence in which the rules are enforced.

Listing 25: Logical sequence of enforced rules.

```
if      Policy1 ∨ (¬ Policy2 ∧ ¬ Policy3 ∧ Policy4
   )
then    Permit
else    Deny
```

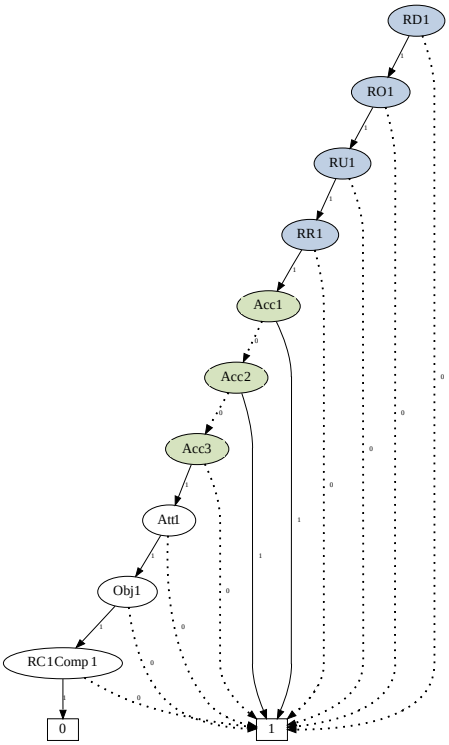

Figure 16: BDD representing Policy2.

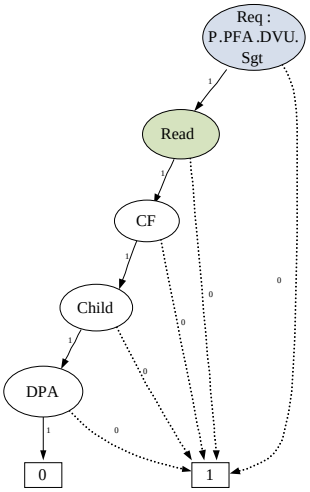

Figure 17: Simplification of Policy2 BDD.

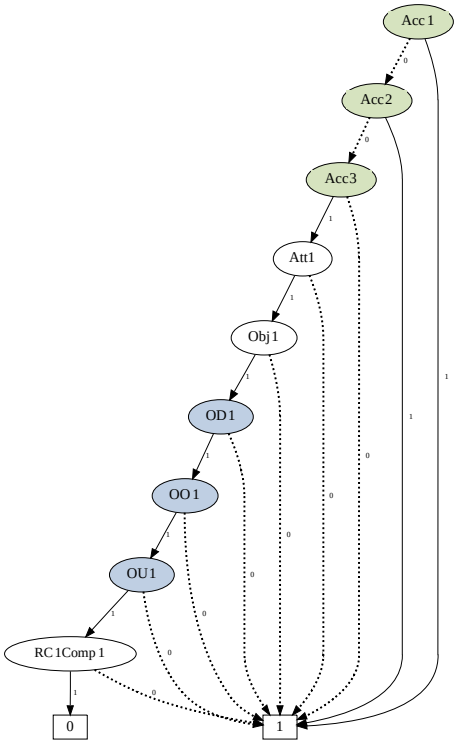

Figure 18: BDD representing Policy3.

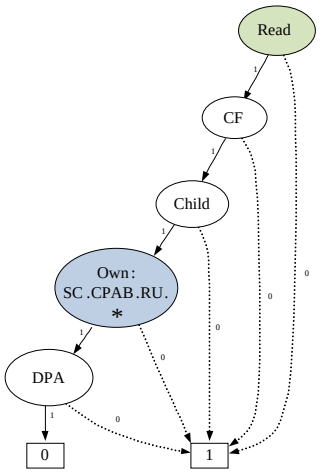

Figure 19: Simplification of Policy3 BDD.

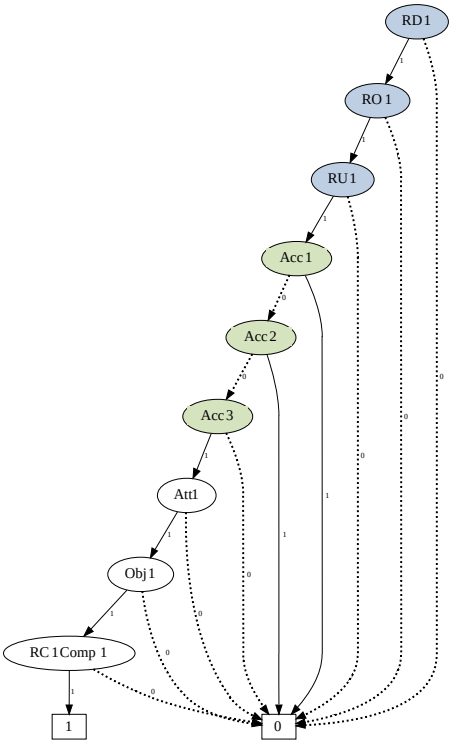

Figure 20: BDD representing Policy4.

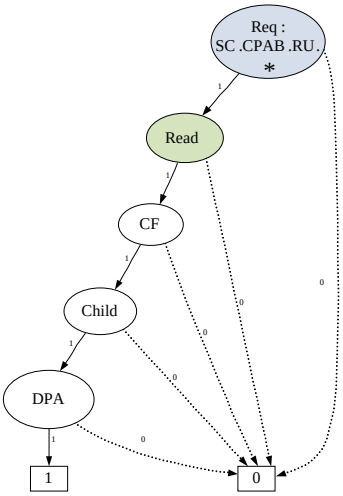

Figure 21: Simplification of Policy4 BDD.

As shown, information can be shared if a request matches Policy1. If Policy1 is not matched, the request can only be permitted if Policy2 *and* Policy3 are *not* matched and Policy4 *is* matched. If no matches are found, the default policy is applied and no information is shared. Since Policy2 and Policy3 have 'deny' actions, they are prefixed with a '¬' to show that these policies must *not* be matched in order for information sharing to take place. The BDD of the combined policies is illustrated in Figure 22.

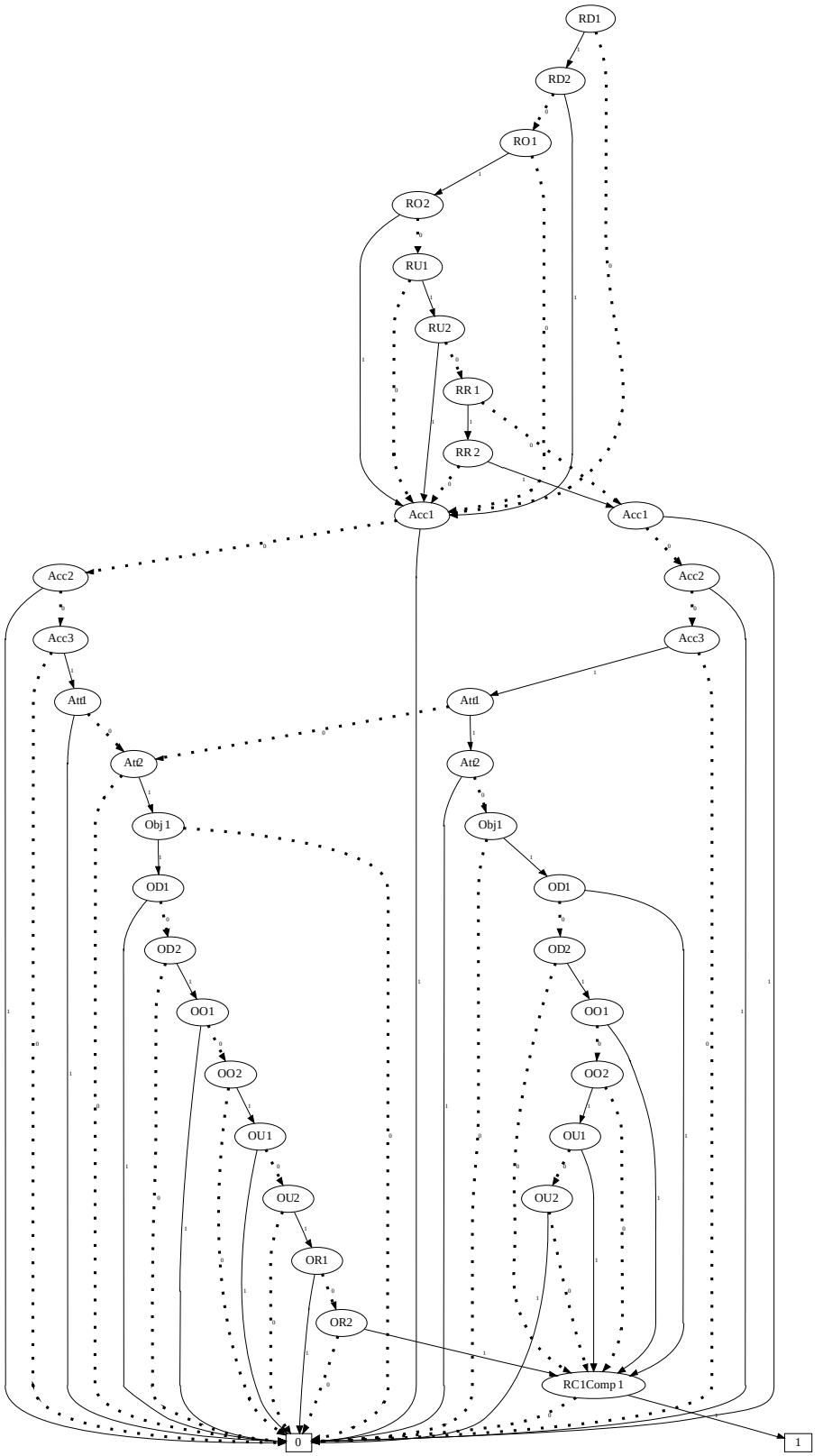

Figure 22: Policy1 $\vee$ ($\neg$ Policy2 $\wedge$ $\neg$ Policy3 $\wedge$ Policy4).

3 Framework Implementation

3.1 Introduction

THIS chapter builds upon the information sharing policy verification framework, as outlined in Chapter 2, and describes the details of its implementation. Specifically, it extends the work of Al-Shaer et al. in [19] and that of Hamed and Al-Shaer in [20]. In these, the authors concentrated on developing formal definitions of possible anomalies between rules in a network firewall rule set. This chapter focuses on extending the principles of formal anomaly definitions to the area of information sharing policies. It outlines a structured approach for the detection of possible anomalies in sets of information sharing policies, and defines the details of each step required for this approach.

3.2 Policy Verification Process

This section describes the mode of operation of the policy verification framework as illustrated in Figure 23. The process used to verify a proposed policy against possible anomalies uses:

- Definition of policy syntax structure and declaration of policy field elements.
- Syntactic verification of the proposed policy.
- Ontological verification of the proposed policy.
- Functional verification of the proposed policy.

3.2.1 Definition

The definition stage comprises a two-step process where the first step requires the definition of the syntax to be used to describe information sharing policies, and the second step requires a declaration of all possible elements which can occur as part of the fields of a policy. Thus, the definition stage defines the *universe* of all possible policy fields, as well as the possible elements of each field, and forms the basis of any subsequent policy verification processes. The following example illustrates the definition process.

Policy Definition Example This example assumes the scenario that two organisations, Police Force A and Child Protection Agency B, have initiated an information sharing agreement, a policy from which is shown in Listing 26.

Listing 26: Example of a policy in an information sharing agreement.

```
The Records Admin from the Records Unit of
   Child Protection Agency B permits a Sergeant
   from the Domestic Violence Unit of
   Police Force A to read the Unique Identifier of
   a Child, whilst complying with the
   Human Rights Act, 1998, as long as it is for an
```

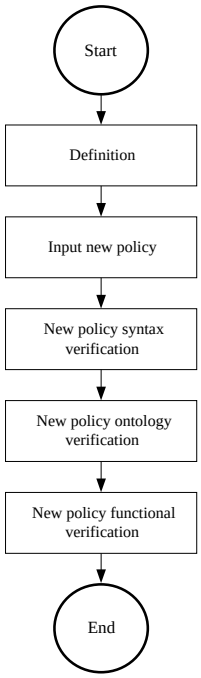

Figure 23: Mode of operation of the policy verification framework.

```
abuse investigation and the Sergeant is the
Investigating Officer.
```

The first step of the definition stage requires the specification of the information sharing policy syntax. For the purposes of this example, information sharing policies are defined as having a nine-field syntax, where each field is enclosed within square brackets, '[' and ']', as illustrated in Listing 27. Detailed explanation of the structure of this syntax is given in Section 2.2.2.

Listing 27: Nine-field syntax used to define information sharing policies.

```
[permission] [requester] [relationship] [action] [
   attribute]   [object] [context] [owner] [
   compliance]
```

Once the information sharing policy syntax has been specified, it can be used to represent polices from the information sharing agreement. For the purposes of this example, the `requester` and `owner` fields are defined as hierarchical, while the remainder of the fields are non-hierarchical. The `requester` and `owner` fields are, both, subdivided into `domain`, `organisation`, `unit` and `role`, with a full stop (‘`.`’) used to delineate between each field sub-division. Listing 28 shows the example information sharing policy from Listing 26 using the syntax from Listing 27.

Listing 28: Information sharing policy from 26 espressed in syntax from 27.

```
[Permit] [Police.Police_Force_A.
   Domestic_Violence_Unit.Sergeant] with [
   Investigating_Officer] relationship [R]    [
   Unique_Identifier] of [Child] with [
   Abuse_Investigation] context from [Social_Care.
   Child_Protection_Agency_B.Records_Unit.
   Records_Admin] with Compliance                [
   Human_Rights_Act_1998]
```

The second step of the definition stage requires a declaration of all possible elements which can occur within the fields of a policy. Table 7 illustrates all possible elements from an example information sharing agreement. In fact, elements used to define the example policy shown in Listing 28 have all been selected from Table 7. It should be noted that a policy field can also be defined using the ‘*’ wildcard, which symbolises that an element has not been declared for a specific field in a policy. Further, as illustrated in Table 7, the elements of the hierarchical `requester` and `owner` fields are declared with respect to their specific higher-level fields. This means that the elements of the highest-level field, `domain`, would be declared by themselves. For example, in Table 7, the domains `Police` and `Social_Care` would be declared by themselves. The elements of the `organisation` field, however, are declared in relation to their respective domains. For example, the organisation `Police_Force_A` is declared in relation to its specific domain, as `Police.Police_Force_A`. Similarly, the organisation `Child_Protection_Agency_B` is declared in relation to its specific domain, as `Social_Care.Child_Protection_Agency_B`. The same principle applies to the subsequent lower-level fields of `unit` and `role`.

3.2.2 Syntax Verification

Syntax verification is the initial stage of the policy verification process. During this stage, a proposed policy is checked to verify that it satisfies the defined syntax criteria for information sharing policies, as specified previously during the definition stage, illustrated in Listing 27. The following example illustrates the syntax verification process. If the proposed policy does not comply with this syntax structure, the testing

Table 7: Possible elements of example information sharing policy.

Policy Field	Declared Elements
Permission	Permit Deny
Domain	Social_Care Police
Organisation	(Social_Care) + Child_Protection_Agency_B (Police) + Police_Force_A
Unit	(Social_Care.Child_Protection_Agency_B) + Records_Unit (Police.Police_Force_A) + Domestic_Violence_Unit
Role	(Social_Care.Child_Protection_Agency_B.Records_Unit) + Records_Admin (Police.Police_Force_A.Domestic_Violence_Unit) + Sergeant
Relationship	Investigating_Officer
Action	Read
Attribute	Health_Record Unique_Identifier
Object	Child
Context	Abuse_Investigation
Compliance	Data_Protection_Act

process is terminated, as other tests only need to be carried out if a policy meets the required syntax criteria.

Syntax Verification Example This example assumes an information sharing policy, shown in Listing 29, is proposed to be added to an existing set of policies.

Listing 29: Example policy used for syntax verification example.

```
[Permit] [Police.Police_Force_A.*.Sergeant] with
   [*] relationship [R] [Unique_Identifier] of [
   Child] with       [Abuse_Investigation] context
   from [Social_Care.Child_Protection_Agency_B.
   Records_Unit.Records_Admin] with Compliance [
   Human_Rights_Act_1998]
```

The syntax verification stage checks that the number of fields specified in the proposed policy, as well as the syntax structure of the proposed policy, correspond with the syntax and number of fields defined earlier in the definition stage, as shown in Listing 27. In this example, the policy shown in Listing 29 is parsed to extract its constituent fields, as enclosed within square brackets, '[' and ']'. This process checks that the number of fields in the proposed policy corresponds correctly with the number of fields defined earlier in the definition stage. Further, the `requester` and `owner` fields from the proposed policy, that is fields two and eight, respectively, are checked to

ensure that they correspond correctly with the hierarchical structure defined for these fields. This entails ensuring that fields two and eight contain four sub-divisions which are delineated using full stops, (‘.’). Since the example policy shown in Listing 29 is expressed correctly using the defined syntax, the syntax verification is successful and the next stage of policy verification commences. If the proposed policy had not complied with the defined syntax structure, the testing process would be terminated, as other tests only need to be carried out if a policy meets the required syntax criteria.

3.2.3 Ontology Verification

Ontology verification is the second stage of the policy verification process, following syntax verification. During this stage, a proposed policy is checked to verify that each field of the policy statement comprises valid elements. An element is designated as valid if it has been previously declared in the definition stage. The following example illustrates this process.

Ontology Verification Example This example assumes a scenario where policy field elements, as shown in Table 8, are specified as part of the definition stage in an information sharing agreement.

Table 8: Defined policy field elements for Ontology Verification Example.

Policy Field	Declared Elements
Permission	Permit Deny
Domain	Social_Care Police
Organisation	(Social_Care) + Child_Protection_Agency_B (Police) + Police_Force_A
Unit	(Social_Care.Child_Protection_Agency_B) + Records_Unit (Police.Police_Force_A) + Domestic_Violence_Unit
Role	(Social_Care.Child_Protection_Agency_B.Records_Unit) + Records_Admin (Police.Police_Force_A.Domestic_Violence_Unit) + Sergeant
Relationship	Investigating_Officer
Action	Read (R) Create (C) Update (U) Delete (D)
Attribute	Health_Record Unique_Identifier
Object	Child
Context	Abuse_Investigation
Compliance	Data_Protection_Act

Four information sharing policies, R_w, R_x, R_y and R_z, are proposed to be added to an existing set of policies. Listings 30, 31, 32 and 33 show policies R_w, R_x, R_y and R_z, respectively.

Listing 30: Policy R_w used for ontology verification example.

```
[Permit] [Police.Police_Force_A.
   Domestic_violence_Unit.Sergeant] with [*]
   relationship [R] [Unique_Identifier] of [Child]
   with [Abuse_Investigation] context from
        [Social_Care.Child_Protection_Agency_B.
   Records_Unit.Records_Admin] with Compliance [
   Human_Rights_Act_1998]
```

Listing 31: Policy R_x used for ontology verification example.

```
[Permit] [Police.Police_Force_A.
   Domestic_violence_Unit.*] with [*] relationship
   [R] [Unique_Identifier] of [Child] with  [
   Abuse_Investigation] context from [Social_Care.
   Child_Protection_Agency_B.Records_Unit.
   Records_Admin] with Compliance [
   Human_Rights_Act_1998]
```

Listing 32: Policy R_y used for ontology verification example.

```
[Permit] [Police.Police_Force_A.
   Domestic_violence_Unit.Constable] with [*]
   relationship [R] [Unique_Identifier] of [Child]
   with [Abuse_Investigation] context from
        [Social_Care.Child_Protection_Agency_B.
   Records_Unit.Records_Admin] with Compliance [
   Human_Rights_Act_1998]
```

Listing 33: Policy R_z used for ontology verification example.

```
[Permit] [Police.Police_Force_A.
   Domestic_violence_Unit.Records_Admin] with [*]
   relationship [R]                   [
   Unique_Identifier] of [Child] with [
   Abuse_Investigation] context from [Social_Care.
   Child_Protection_Agency_B.Records_Unit.
   Records_Admin] with Compliance               [
   Human_Rights_Act_1998]
```

The initial step of the ontology verification stage comprises parsing of the proposed policies in order to extract their constituent field elements, as enclosed within square brackets, '[' and ']'. Table 9 shows field elements extracted from the example proposed policies.

It must be noted here that where the '*' wildcard is used instead of a field element, the corresponding field is not checked for against an entry in the definition. Comparison between elements of the proposed policies, as shown in Table 9, and the declared elements shown in Table 8, illustrates that each element in the proposed policy, R_w, exists as a valid declared element, in its respective declared field. Therefore, for policy R_w, the ontology verification process is successful, and the next stage of verification can commence. Similar to policy R_w, each element in the proposed policy, R_x, also

Table 9: Field elements of proposed information sharing policies R_w, R_x, R_y and R_z for Ontology Verification Example 3.2.3.

Policy Field	Elements of Policy R_w	Elements of Policy R_x	Elements of Policy R_y	Elements of Policy R_z
Permission	Permit	Permit	Permit	Permit
RD	Police	Police	Police	Police
RO	(Police)+ PFA	(Police) + PFA	(Police) + PFA	(Police) + PFA
RU	(Police)+ (PFA)+ DVU	(Police)+ (PFA)+ DVU	(Police)+ (PFA)+ DVU	(Police)+ (PFA)+ DVU
RR	(Police)+ (PFA)+ (DVU)+ Sergeant	(Police)+ (PFA)+ (DVU)+ *	(Police)+ (PFA)+ (DVU)+ Constable	(Police)+ (PFA)+ (DVU)+ RA
Relationship	*	*	*	*
Action	R	R	R	R
Attribute	Unique Identifier	Unique Identifier	Unique Identifier	Unique Identifier
Object	Child	Child	Child	Child
Context	Abuse Investigation	Abuse Investigation	Abuse Investigation	Abuse Investigation
OD	SC	SC	SC	SC
OO	(SC)+ CPAB	(SC)+ CPAB	(SC)+ CPAB	(SC)+ CPAB
OU	(SC)+ (CPAB)+ RU	(SC)+ (CPAB)+ RU	(SC)+ (CPAB)+ RU	(SC)+ (CPAB)+ RU
OR	(SC)+ (CPAB)+ (RU)+ RA	(SC)+ (CPAB)+ (RU)+ RA	(SC)+ (CPAB)+ (RU)+ RA	(SC)+ (CPAB)+ (RU)+ RA
Compliance	Data Protection Act	Data Protection Act	Data Protection Act	Data Protection Act

Field or Element		Abbreviation
Requester/Owner Domain	:	RD/OD
Requester/Owner Organisation	:	RO/OO
Requester/Owner Unit	:	RU/OU
Requester/Owner Role	:	RR/OR
Police_Force_A	:	PFA
Domestic_Violence_Unit	:	DVU
Social_Care	:	SC
Child_Protection_Agency_B	:	CPAB
Records_Unit	:	RU
Records_Admin	:	RA
Data_Protection_Act	:	DPA

exists as a valid declared element, in its respective declared field. Since R_x shows the `Requester_Role` element as the wildcard '*', this field is not checked against the respective field in the definition, and, hence, the ontology verification process for policy R_x is also successful.

In the case of policy R_y, however, the element for the `Requester_Role` field is shown as `Constable`. Since `Constable` is not a defined element for the `Requester_Role` field, and, hence, does not appear as a defined element in the column in Table 8, policy R_y will fail the ontology verification stage. In the case of policy R_z, although the `Requester_Role` field, shown as `Records_Admin`, exists as a declared element in the definitions in Table 8, it does not belong to the `Police.Police_Force_A.Domestic_violence_Unit` hierarchy. Therefore, policy R_z will also fail the ontology verification stage. In the case where a policy fails the ontology verification stage, the testing process would be terminated as other tests only need to be carried out if a policy meets the required ontology criteria.

3.2.4 Functional Verification

The functional verification stage is the final stage of the policy verification process and identifies any potential anomalies which may exist between a proposed policy and those present in an existing set of policies. This stage uses the anomaly definitions of redundancy, shadowing, generalisation and correlation, as detailed in Section **??**. Therefore, functional verification is split into four stages, each to check for a specific category of anomaly. The logical state diagram for anomaly classification is based on the work of Al-Shaer and Hamed in [9], illustrated in Figure 24.

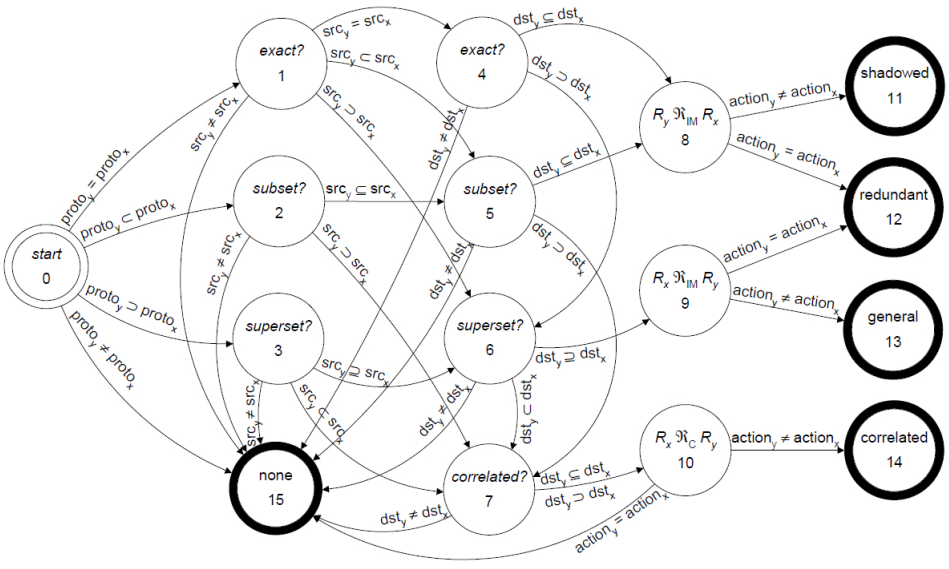

Figure 24: Anomaly state diagram from [9] for rules R_x and R_y, where R_x precedes R_y in the rules list.

A simplified version of the anomaly classification method, derived from the work of Al-Shaer and Hamed in [9], is shown in Figure 25. As indicated, anomalies are detected by logically comparing the permissions, P_x and P_y, and conditions, C_x and C_y, for any two policies, R_x and R_y.

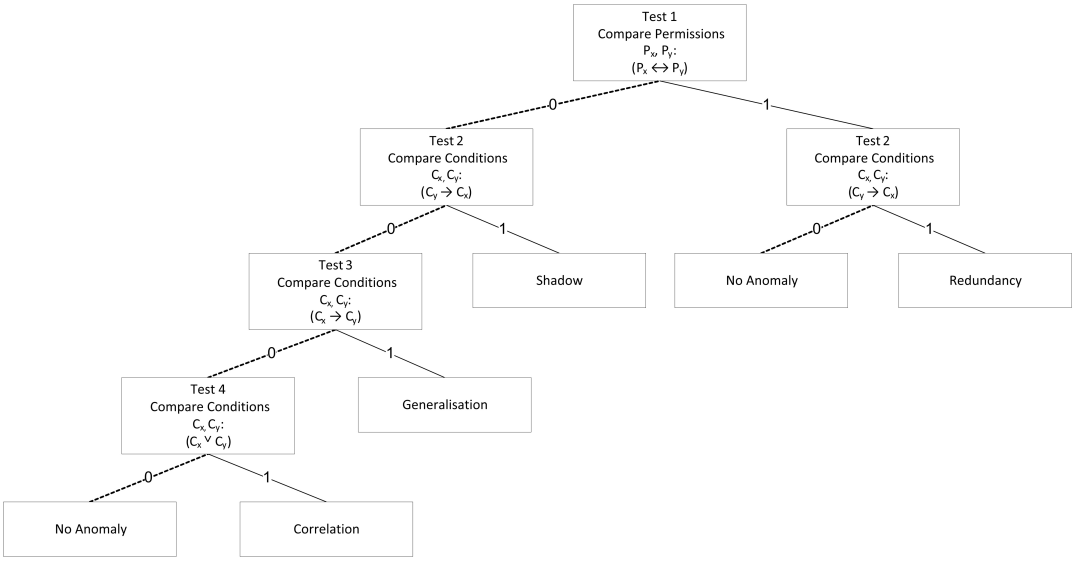

Figure 25: Simplified anomaly classification method for functional verification.

The comparison process, with respect to anomaly detection, entails identification of a subset, superset or equivalence relation between respective fields of the policies, R_x and R_y, which is achieved by carrying out bitwise logical operations between comparative fields. When operating on non-hierarchical fields, the operation simply involves using the entire field in the comparison. For hierarchical fields, however, this involves the definition of the fields as logical conjunctions ($\wedge$), of all of their sub-fields. The logical operation is then performed for the entire hierarchical field. For example, the `Requester` and `Owner` fields can both be defined as logical conjunctions of their constituent `Domain`, `Organisation`, `Unit` and `Role` sub-fields:

$$\begin{array}{rcl} Requester_x, Owner_x & : & Domain_x \wedge Organisation_x \wedge Unit_x \wedge Role_x \\ Requester_y, Owner_y & : & Domain_y \wedge Organisation_y \wedge Unit_y \wedge Role_y \end{array}$$

In practice, the logical comparisons are carried out using Binary Decision Diagrams (BDDs), as implemented for firewall rules by Hazelhurst et al. in [17] and access list modelling by Hazelhurst in [16]. However, the logical 'AND' operation, or conjunction ($\wedge$), is used here to identify subset, superset or equivalence relations. This allows a generic method of illustrating logical relationships, which is independent of the internal computations of any specific Binary Decision Diagram (BDD) software package.

Since BDDs perform bitwise logical operations, the permissions, P_x and P_y, and conditions, C_x and C_y, of policies R_x and R_y must first be represented in binary bits. However, unlike modelling firewall rules and access lists, information sharing policies can have fields of varying lengths. This is due to the fact that there is no limit on the number of possible elements which may be declared as part of a field in an information

sharing agreement. Hence, since the number of possible elements in a field can vary, where each field must be adjusted for the number of bits before any bitwise operations can be performed on it. The adjustment process involves assigning an element identifier, Element ID, to each element in a field and then representing the element identifier in binary form. Hence, the total number of binary variables needed to represent a field element is dependent on the total number of possible field elements declared during the definition stage. Table 10 illustrates the assignment of Element IDs to elements from Table 8, from the Ontology Verification Example, and their binary representations.

Table 10: Assignment of Element IDs to elements from Table 8 and their binary representation.

Policy Field	Elements from Definition	Element ID	Binary Representation
Permission	Deny	0	0
	Permit	1	1
Domain	Social Care	1	01
	Police	2	10
Organisation	Child Protection Agency B	1	01
	Police Force A	2	10
Unit	Records Unit	1	01
	Domestic Violence Unit	2	10
Role	Records Admin	1	01
	Sergeant	2	10
Relationship	Investigating Officer	1	1
Action	Read (R)	1	001
	Create (C)	2	010
	Update (U)	3	011
	Delete (D)	4	100
Attribute	Health Record	1	01
	Unique Identifier	2	10
Object	Child	1	1
Context	Abuse Investigation	1	1
Compliance	Data Protection Act	1	1

Listing 34 shows an example information sharing policy with field elements populated from elements defined in Table 10. Listing 35 illustrates the example policy from Listing 34 in its binary representation.

Listing 34: Example information sharing policy with field elements populated from elements defined in Table 10.

R_x **:**

```
[Permit] [Police.Police_Force_A.
   Domestic_Violence_Unit.Sergeant] with [
   Investigating_Officer] relationship [R]    [
   Health_Record] of [Child] with [
   Abuse_Investigation] context from [Social_Care.
   Child_Protection_Agency_B.Records_Unit.
   Records_Admin] with Compliance               [
   Data_Protection_Act]
```

Listing 35: Example information sharing policy from Listing 34 in binary representation.

R_x :
```
[1] [10.10.10.10] [1] [001] [01] [1] [1]
    [01.01.01.01] [1]
```

The binary representation in Listing 35 shows that an information sharing policy populated from elements defined in Table 10 will consist of 26 binary bits. If, however, the Role field is now updated to include two additional roles, Constable and Analyst, the number of bits required to represent the policy will change. Table 11 shows the Role field from Table 10 updated with the additional roles.

Table 11: Role field from Table 10 updated with Constable and Analyst roles.

Policy Field	Elements from Definition	Element ID	Binary Representation
Role	Records Admin	1	001
	Sergeant	2	010
	Constable	3	011
	Analyst	4	100

The same information sharing policy, R_x from Listing 34, now expressed using the updated Role field from Table 11, will have the binary form as shown in Listing 36. As shown, 28 binary bits are now required to represent R_x whereas previously only 26 binary bits were required. This illustrates the sensitivity of binary expressions and, hence, any binary calculations, to modifications made to the set of elements registered for an information sharing agreement.

Listing 36: Example information sharing policy from Listing 34 represented in binary form using updated Role field from Table 11.

R_x :
```
[1] [10.10.10.010] [1] [001] [01] [1] [1]
    [01.01.01.001] [1]
```

The following examples provide details on each stage of the functional verification process. The methods used for functional verification are as illustrated in Figure 25. Each example assumes an information sharing agreement between two organisations, *Police Force A* and *Child Protection Agency B*. Only the `requester` and `owner` fields are manipulated in the following anomaly verification examples for reasons of brevity.

3.2.5 Example of Redundancy Anomaly Verification

Elements from Table 10 are used to define two information sharing policies. Listing 37 shows R_x, an existing policy in the agreement and Listing 38 shows R_y, a proposed policy to be added. Listing 39 shows the binary representations of the two policies.

Listing 37: Existing policy set for redundancy anomaly functional verification example.

```
R_x : [Permit] [Police.Police_Force_A.
Domestic_Violence_Unit.*] with [*] relationship
[R] [*] of [Child] with [*] context from [
Social_Care.Child_Protection_Agency_B.
Records_Unit.*] with Compliance [*]
```

Listing 38: Proposed policy for redundancy anomaly functional verification example.

```
R_y : [Permit] [Police.Police_Force_A.
Domestic_Violence_Unit.Sergeant] with [*]
relationship [R] [*] of [Child] with [*] context
 from [Social_Care.Child_Protection_Agency_B.
Records_Unit.*] with Compliance [*]
```

Listing 39: Binary representation of existing policy R_x from Listing 37 and proposed policy R_y from Listing 38 for redundancy anomaly functional verification example.

```
R_x : [1] [10.10.10.*] with [*] relationship [R]
[*] of [1] with [*] context from [01.01.01.*]
with Compliance [*]
R_y : [1] [10.10.10.10] with [*] relationship [R]
 [*] of [1] with [*] context from [01.01.01.*]
with Compliance [*]
```

Test 1: As illustrated in Figure 25, the first comparison in the functional verification stage, Test 1, is to compare the permissions, P_x and P_y, from policies R_x and R_y to check if they are the same. This operation is illustrated in the computation below using the logical relationship, from Equation 2, that if the permissions P_x and P_y are the same, $(P_x \Leftrightarrow P_y)$ is TRUE, then the expression $((P_x \wedge P_y) \vee (\neg P_x \wedge \neg P_y))$ must also be TRUE.

$$\begin{aligned} P_x &: 1 \\ P_y &: 1 \\ (P_x \Leftrightarrow P_y) &: ((P_x \wedge P_y) \vee (\neg P_x \wedge \neg P_y)) \\ &: ((1 \wedge 1) \vee (\neg 1 \wedge \neg 1)) \\ \therefore (P_x \Leftrightarrow P_y) &: \text{TRUE} \end{aligned}$$

Test 2: Since P_x and P_y are the same, the next comparison in the functional verification stage, Test 2, is to compare the conditions, C_x and C_y, to check if $(C_y \Rightarrow C_x)$ is TRUE. This test, if TRUE, indicates that C_y is either equal to, or a subset of, C_x which means that policy R_y is redundant to policy R_x. This operation is illustrated in the computation below using the logical relationship, from Equation 1, that if condition C_y implies condition C_x, $(C_y \Rightarrow C_x)$ is TRUE, then the expression $(\neg C_y \vee C_x)$ must also be TRUE.

$$
\begin{array}{rcc}
C_x & : & \text{10.10.10.XX} \\
C_y & : & \text{10.10.10.10} \\
C_y \Rightarrow C_x & : & \neg C_y \vee C_x \\
 & : & \text{01.01.01.01} \vee \text{10.10.10.XX} \\
\therefore C_y \Rightarrow C_x & : & \text{TRUE}
\end{array}
$$

Conclusion: As can be seen from Figure 25, if Test 1 and Test 2 result in TRUE, policy R_y is redundant to policy R_x.

3.2.6 Example of Shadow Anomaly Verification

Elements from Table 10 are used to define two information sharing policies. Listing 40 shows R_x, an existing policy in the agreement and Listing 41 shows R_y, a proposed policy to be added. Listing 42 shows the binary representations of the two policies.

Listing 40: Existing policy set for shadow anomaly functional verification example.

R_x : `[Deny] [Police.Police_Force_A. Domestic_Violence_Unit.*] with [*] relationship [R] [*] of [Child] with [*] context from [ Social_Care.Child_Protection_Agency_B. Records_Unit.*] with Compliance [*]`

Listing 41: Proposed policy for shadow anomaly functional verification example.

R_y : `[Permit] [Police.Police_Force_A. Domestic_Violence_Unit.Sergeant] with [*] relationship [R] [*] of [Child] with [*] context from [Social_Care.Child_Protection_Agency_B. Records_Unit.*] with Compliance [*]`

Listing 42: Binary representation of existing policy R_x from Listing 40 and proposed policy R_y from Listing 41 for shadow anomaly functional verification example.

R_x : `[0] [10.10.10.*] with [*] relationship [R] [*] of [1] with [*] context from [01.01.01.*] with Compliance [*]`

R_y : `[1] [10.10.10.10] with [*] relationship [R] [*] of [1] with [*] context from [01.01.01.*] with Compliance [*]`

Test 1: As illustrated in Figure 25, the first comparison in the functional verification stage, Test 1, is to compare the permissions, P_x and P_y, from policies R_x and R_y to check if they are the same. This operation is illustrated in the computation below using the logical relationship, from Equation 2, that if the permissions P_x and P_y are the

same, $(P_x \Leftrightarrow P_y)$ is TRUE, then the expression $((P_x \wedge P_y) \vee (\neg P_x \wedge \neg P_y))$ must also be TRUE.

$$\begin{array}{rcc} P_x & : & 0 \\ P_y & : & 1 \\ (P_x \Leftrightarrow P_y) & : & ((P_x \wedge P_y) \vee (\neg P_x \wedge \neg P_y)) \\ & : & ((0 \wedge 1) \vee (\neg 0 \wedge \neg 1)) \\ \therefore (P_x \Leftrightarrow P_y) & : & \text{FALSE} \end{array}$$

Test 2: Since P_x and P_y are different, the next comparison in the functional verification stage, Test 2, is to compare the conditions, C_x and C_y, to check if $(C_y \Rightarrow C_x)$ is TRUE. This test, if TRUE, indicates that C_y is either equal to, or a subset of, C_x which means that policy R_y is shadowed by policy R_x. This operation is illustrated in the computation below using the logical relationship, from Equation 1, that if condition C_y implies condition C_x, $(C_y \Rightarrow C_x)$ is TRUE, then the expression $(\neg C_y \vee C_x)$ must also be TRUE.

$$\begin{array}{rcc} C_x & : & \text{10.10.10.XX} \\ C_y & : & \text{10.10.10.10} \\ C_y \Rightarrow C_x & : & \neg C_y \vee C_x \\ & : & \text{01.01.01.01} \vee \text{10.10.10.XX} \\ \therefore C_y \Rightarrow C_x & : & \text{TRUE} \end{array}$$

Conclusion: As can be seen from Figure 25, if Test 1 results in FALSE and Test 2 results in TRUE, policy R_y is shadowed by policy R_x.

3.2.7 Example of Generalisation Anomaly Verification

Elements from Table 10 are used to define two information sharing policies. Listing 43 shows R_x, an existing policy in the agreement and Listing 44 shows R_y, a proposed policy to be added. Listing 45 shows the binary representations of the two policies.

Listing 43: Existing policy set for generalisation anomaly functional verification example.

R_x :
```
    [Deny] [Police.Police_Force_A.
Domestic_Violence_Unit.Sergeant] with [*]
relationship [R] [*] of [Child] with [*] context
 from [Social_Care.Child_Protection_Agency_B.
Records_Unit.*] with Compliance [*]
```

Listing 44: Proposed policy for generalisation anomaly functional verification example.

R_y :
```
    [Permit] [Police.Police_Force_A.
Domestic_Violence_Unit.*] with [*] relationship
[R] [*] of [Child] with [*] context from [
Social_Care.Child_Protection_Agency_B.
Records_Unit.*] with Compliance [*]
```

Listing 45: Binary representation of existing policy R_x from Listing 43 and proposed policy R_y from Listing 44 for generalisation anomaly functional verification example.

```
Rx : [0] [10.10.10.10] with [*] relationship [R]
     [*] of [1] with [*] context from [01.01.01.*]
     with Compliance [*]
Ry : [1] [10.10.10.*] with [*] relationship [R]
     [*] of [1] with [*] context from [01.01.01.*]
     with Compliance [*]
```

Test 1: As illustrated in Figure 25, the first comparison in the functional verification stage, Test 1, is to compare the permissions, P_x and P_y, from policies R_x and R_y to check if they are the same. This operation is illustrated in the computation below using the logical relationship, from Equation 2, that if the permissions P_x and P_y are the same, $(P_x \Leftrightarrow P_y)$ is TRUE, then the expression $((P_x \wedge P_y) \vee (\neg P_x \wedge \neg P_y))$ must also be TRUE.

$$\begin{array}{rcc} P_x & : & 0 \\ P_y & : & 1 \\ (P_x \Leftrightarrow P_y) & : & ((P_x \wedge P_y) \vee (\neg P_x \wedge \neg P_y)) \\ & : & ((0 \wedge 1) \vee (\neg 0 \wedge \neg 1)) \\ \therefore (P_x \Leftrightarrow P_y) & : & \text{FALSE} \end{array}$$

Test 2: Since P_x and P_y are different, the next comparison in the functional verification stage, Test 2, is to compare the conditions, C_x and C_y, to check if $(C_y \Rightarrow C_x)$ is TRUE. This test, if TRUE, indicates that C_y is either equal to, or a subset of, C_x which means that policy R_y is shadowed by policy R_x. This operation is illustrated in the computation below using the logical relationship, from Equation 1, that if condition C_y implies condition C_x, $(C_y \Rightarrow C_x)$ is TRUE, then the expression $(\neg C_y \vee C_x)$ must also be TRUE.

$$\begin{array}{rcc} C_x & : & 10.10.10.10 \\ C_y & : & 10.10.10.XX \\ C_y \Rightarrow C_x & : & \neg C_y \vee C_x \\ & : & 01.01.01.00 \vee 10.10.10.10 \\ \therefore C_y \Rightarrow C_x & : & \text{FALSE} \end{array}$$

Test 3: Since the result of Test 2 is FALSE, the next comparison in the functional verification stage, Test 3, is to compare the conditions, C_x and C_y, to check if $(C_x \Rightarrow C_y)$ is TRUE. This test, if TRUE, indicates that C_x is either equal to, or a subset of, C_y which means that policy R_y is a generalisation of policy R_x. This operation is illustrated in the computation below using the logical relationship, from Equation 1, that if condition C_x implies condition C_y, $(C_x \Rightarrow C_y)$ is TRUE, then the expression $(\neg C_x \vee C_y)$ must also be TRUE.

$$
\begin{array}{rcc}
C_x & : & 10.10.10.10 \\
C_y & : & 10.10.10.XX \\
C_x \Rightarrow C_y & : & \neg C_x \vee C_y \\
 & : & 01.01.01.01 \vee 10.10.10.11 \\
\therefore C_y \Rightarrow C_x & : & \text{TRUE}
\end{array}
$$

Conclusion: As can be seen from Figure 25, if Test 1 results in FALSE, Test 2 results in FALSE and Test 3 results in TRUE, policy R_y is a generalisation of policy R_x.

3.2.8 Example of Correlation Anomaly Verification

Elements from Table 10 are used to define two information sharing policies. Listing 46 shows R_x, an existing policy in the agreement and Listing 47 shows R_y, a proposed policy to be added. Listing 48 shows the binary representations of the two policies. It must be noted that, unlike in previous examples, in this example the conditions C_x and C_y of policies R_x and R_y have different `Requester` and `Owner` fields. Therefore, the conjunction of both these fields will be used in the tests below when comparing conditions.

Listing 46: Existing policy set for correlation anomaly functional verification example.

R_x :
```
[Deny] [Police.Police_Force_A.
Domestic_Violence_Unit.Sergeant] with [*]
relationship [R] [*] of [Child] with [*] context
 from [Social_Care.Child_Protection_Agency_B.
Records_Unit.*] with Compliance [*]
```

Listing 47: Proposed policy for correlation anomaly functional verification example.

R_y :
```
[Permit] [Police.Police_Force_A.
Domestic_Violence_Unit.*] with [*] relationship
[R] [*] of [Child] with [*] context from [
Social_Care.Child_Protection_Agency_B.
Records_Unit.Records_Admin] with Compliance [*]
```

Listing 48: Binary representation of existing policy R_x from Listing 46 and proposed policy R_y from Listing 47 for correlation anomaly functional verification example.

R_x :
```
[0] [10.10.10.10] with [*] relationship [R]
 [*] of [1] with [*] context from [01.01.01.*]
with Compliance [*]
```
R_y :
```
[1] [10.10.10.*] with [*] relationship [R]
[*] of [1] with [*] context from [01.01.01.01]
with Compliance [*]
```

Test 1: As illustrated in Figure 25, the first comparison in the functional verification stage, Test 1, is to compare the permissions, P_x and P_y, from policies R_x and R_y to check if they are the same. This operation is illustrated in the computation below using the logical relationship, from Equation 2, that if the permissions P_x and P_y are the same, $(P_x \Leftrightarrow P_y)$ is TRUE, then the expression $((P_x \wedge P_y) \vee (\neg P_x \wedge \neg P_y))$ must also be TRUE.

$$\begin{array}{rcc} P_x & : & 0 \\ P_y & : & 1 \\ (P_x \Leftrightarrow P_y) & : & ((P_x \wedge P_y) \vee (\neg P_x \wedge \neg P_y)) \\ & : & ((0 \wedge 1) \vee (\neg 0 \wedge \neg 1)) \\ \therefore (P_x \Leftrightarrow P_y) & : & \text{FALSE} \end{array}$$

Test 2: Since P_x and P_y are different, the next comparison in the functional verification stage, Test 2, is to compare the conditions, C_x and C_y, to check if $(C_y \Rightarrow C_x)$ is TRUE. This test, if TRUE, indicates that C_y is either equal to, or a subset of, C_x which means that policy R_y is shadowed by policy R_x. This operation is illustrated in the computation below using the logical relationship, from Equation 1, that if condition C_y implies condition C_x, $(C_y \Rightarrow C_x)$ is TRUE, then the expression $(\neg C_y \vee C_x)$ must also be TRUE.

		Requester Field	Owner Field
C_x	:	10.10.10.10	01.01.01.XX
C_y	:	10.10.10.XX	01.01.01.01
$C_y \Rightarrow C_x$	:	$\neg C_y \vee C_x$	$\neg C_y \vee C_x$
	:	01.01.01.00 $\vee$ 10.10.10.10	10.10.10.10 $\vee$ 01.01.01.11
	:	FALSE	TRUE
$\therefore C_y \Rightarrow C_x$	:	FALSE	

Test 3: Since the result of Test 2 is FALSE, the next comparison in the functional verification stage, Test 3, is to compare the conditions, C_x and C_y, to check if $(C_x \Rightarrow C_y)$ is TRUE. This test, if TRUE, indicates that C_x is either equal to, or a subset of, C_y which means that policy R_y is a generalisation of policy R_x. This operation is illustrated in the computation below using the logical relationship, from Equation 1, that if condition C_x implies condition C_y, $(C_x \Rightarrow C_y)$ is TRUE, then the expression $(\neg C_x \vee C_y)$ must also be TRUE.

		Requester Field	Owner Field
C_x	:	10.10.10.10	01.01.01.XX
C_y	:	10.10.10.XX	01.01.01.01
$C_x \Rightarrow C_y$	:	$\neg C_x \vee C_y$	$\neg C_x \vee C_y$
	:	01.01.01.01 $\vee$ 10.10.10.11	10.10.10.00 $\vee$ 01.01.01.01
	:	TRUE	FALSE
$\therefore C_x \Rightarrow C_y$	:	FALSE	

Test 4: Since the result of Test 3 is FALSE, the next comparison in the functional verification stage, Test 4, is to compare the conditions, C_x and C_y, to check for correlated fields. For this example, this requires checking that the `Requester` and `Owner` fields of one policy are the same as, or subsets of, the corresponding fields of the other policy

and that the remaining fields of the former policy are supersets of the corresponding fields of the latter policy. Formally, this means that for the fields being compared, if for certain fields $(C_x \Rightarrow C_y)$ is TRUE, then $(C_y \Rightarrow C_x)$ must be TRUE for the remaining fields being compared. This operation is illustrated in the computation below using the logical relationship, from Equation 1, that if condition C_x implies condition C_y, $(C_x \Rightarrow C_y)$ is TRUE, then the expression $(\neg C_x \vee C_y)$ must also be TRUE.

		Requester Field	Owner Field
C_x	:	10.10.10.10	01.01.01.XX
C_y	:	10.10.10.XX	01.01.01.01
$Requester_x \Rightarrow Requester_y$	:	$\neg Requester_x \vee Requester_y$	
	:	01.01.01.01 $\vee$ 10.10.10.11	
	:	TRUE	
$Owner_y \Rightarrow Owner_x$	:		$\neg Owner_y \vee Owner_x$
	:		10.10.10.10 $\vee$ 01.01.01.11
	:		TRUE
$\therefore Requester_x \Rightarrow Requester_y$ $\wedge Owner_y \Rightarrow Owner_x$	:	TRUE	

Conclusion: As can be seen from Figure 25, if Test 1 results in FALSE, Test 2 results in FALSE, Test 3 results in FALSE and Test 4 results in TRUE, policies R_x and R_y are correlated.

3.3 Conclusion

This chapter defined a structured methodology which is based on the foundational work of Al-Shaer et al. in [19] and that of Hamed and Al-Shaer in [20]. This includes a detailed discussion of each stage of the methodology, from the declaration of policy field elements, through to the syntax, ontology and functional verification stages. Detailed examples are provided throughout this chapter, in order to clearly illustrate the specifics of each stage of the methodology and demonstrate how it is applied to information sharing policies.

In their works of [19] and [20] the authors concentrated on developing formal definitions of possible anomalies between rules in a network firewall rule set. Their work is considered as the foundation for further works on anomaly detection, including those of Fitzgerald et al. [21], Chen et al. [22], Hu et al. [23], among others. Although these more recent works focus on extending and refining the original research, they are restricted in their focus to the operation of network firewalls. The methodology defined in this chapter abstracts from the original work the core principles relating to formal definitions of anomalies within a set of rules. These abstracted principles, through the use of the structured approach outlined in this chapter, allow anomalies between rules to be detected in any rule-based system. The methodology defined in this chapter integrates with the novel syntax for information sharing, as discussed in Section 2.2.2, and outlines a self-contained architecture for information sharing

4 Conclusion

This paper defined a structured methodology which is based on the foundational work of Al-Shaer et al. in [19] and that of Hamed and Al-Shaer in [20]. This includes a detailed discussion of each stage of the methodology, from the declaration of policy field elements, through to the syntax, ontology and functional verification stages. Detailed examples are provided throughout this chapter, in order to clearly illustrate the specifics of each stage of the methodology and demonstrate how it is applied to information sharing policies.

In their works of [19] and [20] the authors concentrated on developing formal definitions of possible anomalies between rules in a network firewall rule set. Their work is considered as the foundation for further works on anomaly detection, including those of Fitzgerald et al. [21], Chen et al. [22], Hu et al. [23], among others. Although these more recent works focus on extending and refining the original research, they are restricted in their focus to the operation of network firewalls. The methodology defined in this chapter abstracts from the original work the core principles relating to formal definitions of anomalies within a set of rules. These abstracted principles, through the use of the structured approach outlined in this chapter, allow anomalies between rules to be detected in any rule-based system. The methodology defined in this chapter integrates with the novel syntax for information sharing, as discussed in Section 2.2.2, and outlines a self-contained architecture for information sharing

Bibliography

[1] Bureau of Justice Assistance, "Intelligence-Led Policing: The New Intelligence Architecture," 2005. [Online]. Available: http://www.ncjrs.gov/pdffiles1/bja/210681.pdf

[2] P. M. Collier, "Policing and the intelligent application of knowledge," *Public Money & Management*, vol. 26, no. 2, pp. 109–116, 2006.

[3] A. Wool, "A quantitative study of firewall configuration errors," *Computer*, vol. 37, no. 6, pp. 62–67, june 2004.

[4] M. Sloman, "Policy driven management for distributed systems," *Journal of Network and Systems Management*, vol. 2, no. 4, pp. 333–360, December 1994. [Online]. Available: http://pubs.doc.ic.ac.uk/policy-distributed-systems/

[5] B. Meyer, "On formalism in specifications," *Software, IEEE*, vol. 2, no. 1, pp. 6 –26, Jan. 1985.

[6] Books LLC, *Computer Security: Information Security, Access Control List, Non-Repudiation, Trusted Client, Rfpolicy, Computer Surveillance*, B. Group, Ed. Books LLC, November 2010.

[7] S. Hazelhurst, A. Fatti, and A. Henwood, "Binary decision diagram representations of firewall and router access lists," University of the Witwatersrand, Johannesburg, South Africa, Tech. Rep. TR-Wits-CS-1998-3, 1998.

[8] O. Uthmani, W. Buchanan, A. Lawson, R. Scott, B. Schafer, L. Fan, and S. Uthmani, "Crime risk evaluation within information sharing between the police and community partners," *Information & Communications Technology Law*, vol. 20, no. 2, pp. 57–81, 2011.

[9] E. Al-Shaer and H. Hamed, "Firewall policy advisor for anomaly discovery and rule editing," in *Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on*, 2003, pp. 17 – 30.

[10] H. Andersen, "An introduction to binary decision diagrams," 1997.

[11] C. Lee, "Representation of switching circuits by binary decision programs," *Bell System Technical Journal*, vol. 38, pp. 985–999, 1959.

[12] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. C-27, no. 6, p. 509, 1978.

[13] R. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. C-35, no. 8, pp. 677–691, August 1986.

[14] B. Bollig and I. Wegener, "Improving the variable ordering of obdds is np-complete," *IEEE Trans. Comput.*, vol. 45, pp. 993–1002, September 1996.

[15] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Comput. Surv.*, vol. 24, pp. 293–318, September 1992.

[16] S. Hazelhurst, "Algorithms for analysing firewall and router access lists," University of the Witwatersrand, Johannesburg, South Africa, Tech. Rep. TR-WitsCS -1999-5, 2000.

[17] S. Hazelhurst, A. Attar, and R. Sinnappan, "Algorithms for improving the dependability of firewall and filter rule lists," in *Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, ser. DSN '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 576–585. [Online]. Available: http://portal.acm.org/citation.cfm?id=647881.737947

[18] H. Hamed and E. Al-Shaer, "Dynamic rule-ordering optimization for high-speed firewall filtering," in *ASIACCS '06: Proceedings of the 2006 ACM Symposium on Information, computer and communications security*. New York, NY, USA: ACM, 2006, pp. 332–342.

[19] E. Al-Shaer and H. Hamed, "Modeling and management of firewall policies," *IEEE Transactions on Network and Service Management*, vol. 1-1, pp. 2–10, 2004.

[20] H. Hamed and E. Al-Shaer, "Taxonomy of conflicts in network security policies," *Communications Magazine, IEEE*, vol. 44, no. 3, pp. 134 – 141, 2006.

[21] W. M. Fitzgerald, F. Turkmen, S. N. Foley, and B. O'Sullivan, "Anomaly analysis for physical access control security configuration," in *Risk and Security of Internet and Systems (CRiSIS), 2012 7th International Conference on*. IEEE, 2012, pp. 1–8.

[22] Z. Chen, S. Guo, and R. Duan, "Research on the anomaly discovering algorithm of the packet filtering rule sets," in *Pervasive Computing Signal Processing and Applications (PCSPA), 2010 First International Conference on*, 2010, pp. 362–366.

[23] H. Hu, G.-J. Ahn, and K. Kulkarni, "Detecting and resolving firewall policy anomalies," *Dependable and Secure Computing, IEEE Transactions on*, vol. 9, no. 3, pp. 318–331, 2012.