STAIRS 2012
K. Kersting and M. Toussaint (Eds.)

doi:10.3233/978-1-61499-096-3-192

Tools for Finding Inconsistencies in Real-world Logic-based Systems

Kevin MCAREAVEY [a,1] Weiru LIU [a] Paul MILLER [a] Chris MEENAN [b]
[a] *Queen's University Belfast, Northern Ireland*
[b] *Q1 Labs/IBM, Northern Ireland*

Abstract. Currently there is extensive theoretical work on inconsistencies in logic-based systems. Recently, algorithms for identifying inconsistent clauses in a single conjunctive formula have demonstrated that practical application of this work is possible. However, these algorithms have not been extended for full knowledge base systems and have not been applied to real-world knowledge. To address these issues, we propose a new algorithm for finding the inconsistencies in a knowledge base using existing algorithms for finding inconsistent clauses in a formula. An implementation of this algorithm is then presented as an automated tool for finding inconsistencies in a knowledge base and measuring the inconsistency of formulae. Finally, we look at a case study of a network security rule set for exploit detection (QRadar) and suggest how these automated tools can be applied.

Keywords. MISes; MUSes; inconsistency measure; network security.

Introduction

Inconsistencies in any knowledge based system can have a significant, negative impact on how the system will perform and on the conclusions that it will reach. Recent developments in formal Artificial Intelligence (AI) approaches for inconsistency handling in logic-based systems have demonstrated the benefit of identifying the minimal number of formulae needed to create an inconsistency. From this work, a number of measures have been proposed to quantify the degree of blame associated with each formula for the inconsistency of the knowledge base, i.e., how responsible a formula is for the overall inconsistency in the knowledge base. The intuition being that identifying the most problematic formulae in an inconsistent knowledge base is a first-step towards resolving inconsistencies.

There are many real-world systems where a logic-based interpretation is appropriate and which would benefit from this approach to inconsistency handling. In network security for example, it has been demonstrated that intrusion detection systems [1] are suitable for this interpretation and so finding the most problematic rules by applying inconsistency measures is possible. Similarly, in the area of Requirements Engineering (RE) a logic-based approach has also been applied to software requirements specifications [2], while other system such as firewalls, Access Control Lists (ACLs) and access rights would be equally applicable.

[1]Corresponding Author: Centre for Secure Information Technologies, Queen's University Belfast, Belfast, BT3 9DT, Northern Ireland; e-mail: kmcareavey01@qub.ac.uk.

Currently there is little practical application of (or implemented systems for) these logic-based approaches, both in terms of algorithms for finding inconsistencies in a knowledge base, and for measuring the blame of inconsistent formulae. To address these issues, we propose a new algorithm for finding the inconsistencies in a knowledge base using existing algorithms for finding inconsistent clauses in a single formula. An implementation of this algorithm is presented as an automated tool for finding inconsistencies in a knowledge base and measuring the inconsistency of formulae. Finally, we look at a case study of a network security rule set for exploit detection and suggest how these automated tools can be applied.

The paper is organized as follows: in Sec. 1 we introduce notations; in Sec. 2 we describe a process for finding inconsistencies in a knowledge base and its implementation in a tool called MIMUS; in Sec. 3 we discuss measures for calculating the degree of blame associated with each formulae for the inconsistency of a knowledge base and their implementation in a tool called MINC; in Sec. 4 we present a case study of a network security rule set in terms of how inconsistencies can be automatically identified; and in Sec. 5 we conclude the paper.

1. Preliminaries

Let $\mathcal{L}$ denote the propositional language obtained from a finite set of propositional atoms $\mathcal{P} = \{a, b, c, ...\}$, using logical connectives $\{\vee, \wedge, \neg, \rightarrow\}$. We denote formulae from $\mathcal{L}$ as α, β, γ, etc and set inclusion (resp. strict) by $\subseteq$ (resp. $\subset$). Let $\bot$ denote an inconsistent knowledge base. We define a knowledge base K as the union of a rule base (rule set) and a fact base, though either can be empty. A rule base is a set of 'if ... then ...' rules (using material implication) of the form $r : \mathcal{C} \rightarrow \mathcal{A}$, where $r^{\mathcal{C}}$ (resp. $r^{\mathcal{A}}$) denotes the condition (resp. action) of a rule r. If K entails ϕ we denote this as $K \vdash \phi$ where ϕ is the consequence of K.

We define inconsistency (contradiction) as logical inconsistency in the classical sense, i.e., a knowledge base that is unsatisfiable under any interpretation. The term inconsistency measure [3] may refer to: a measure of the overall inconsistency of a knowledge base (knowledge base-level); or a measure of the degree to which a formula contributes to the inconsistency of a knowledge base (formula-level). In this paper we use the formula-level interpretation since this is more relevant for resolving inconsistencies within a single knowledge base. We consider the term inconsistency measure as synonymous with the terms inconsistency value [4], blame measure and degree of blame [5].

Definition 1 (MC). *A Maximal Consistent Subset* Γ *of a knowledge base* K *is*

$$(1)\ \Gamma \subseteq K,\ (2)\ \Gamma \nvdash \bot,\ (3)\ \forall \Omega \subseteq (K \setminus \Gamma)\ s.t.\ \Omega \neq \emptyset,\ \Gamma \cup \Omega \vdash \bot.$$

Definition 2 (MI). *A Minimal Inconsistent Subset (MIS)* Γ *of a knowledge base* K *is*

$$(1)\ \Gamma \subseteq K,\ (2)\ \Gamma \vdash \bot,\ (3)\ \forall \Omega \subset \Gamma, \Omega \nvdash \bot.$$

Then $MI(K)$ is the set of MISes of K and $MC(K)$ the set of Maximal Consistent Subsets of K.

2. Finding MISes from MUSes

While the task of identifying MISes is computationally hard, algorithms for identifying Minimally Unsatisfiable Subformulas (MUSes) [6,7] mean that it is practi-

cally possible. In [4], these MUSes are said to be the same as MISes, however this is not strictly true since a MIS is a set of inconsistent formulae in a knowledge base while a MUS is an unsatisfiable set of clauses from a single formula in conjunctive normal form (CNF). The difference being that MUSes are inconsistent clauses (a disjunction of literals) while MISes are inconsistent formulae. Saying this however, if the model-based view of knowledge bases is taken, i.e., that the knowledge base is equivalent to the conjunction of its formulae, then the knowledge base can be converted to a single formula representing the whole knowledge base. This formula can then be converted to CNF, from which MUSes can be identified. A set of MISes can therefore be determined from each MUS, e.g. given formulae α and β where α results in clauses $\{\alpha_1, \alpha_2\}$ and β results in clauses $\{\beta_1, \beta_2, \beta_3\}$, if there is a MUS $\{\alpha_1, \alpha_2, \beta_2\}$ then the resulting MIS would be $\{\alpha, \beta\}$. This process of finding MISes from MUSes is explained in the remainder of this section and has been implemented in an automated tool called MIMUS.

An algorithm for identifying MUSes, called CAMUS, is described in [6]. This algorithm is based on the relationship between Maximally Satisfiable Subformulas (MSSes) and MUSes, i.e., from the set complement of the MSSes of a formula (called CoMSSes and later Minimal Correction Sets (MCSes) [8,9]) the MUSes can be deduced because a MUS is an irreducible hitting set of the set of MCSes (see Example. 3 and 4). A modification of the CAMUS algorithm, called HYCAM, has also been developed [7]. Implementations of both algorithms are available [10,11] incorporating a SAT solver for determining the MSSes of a formula. We will demonstrate finding MISes from MUSes on a knowledge base Δ from [3].

Definition 3 (MSS [7]). *A MSS* Γ *of a set of clauses* Σ *is a set of clauses s.t.*

(1) $\Gamma \subseteq \Sigma$, (2) Γ *is satisfiable,* (3) $\forall \Omega \subseteq (\Sigma \setminus \Gamma)$ *s.t.* $\Omega \neq \emptyset$, $\Gamma \cup \Omega$ *is unsatisfiable.*

Definition 4 (MCS or CoMSS [7]). *The MCS of a MSS* Γ *of a set of clauses* Σ *is given by* $\Sigma \setminus \Gamma$.

Definition 5 (MUS [7]). *A MUS* Γ *of a set of clauses* Σ *is a set of clauses s.t.*

(1) $\Gamma \subseteq \Sigma$, (2) Γ *is unsatisfiable,* (3) $\forall \Omega \subset \Gamma$, Ω *is satisfiable.*

Definition 6 (Hitting set [8]). *Let* Ω *denote a set of sets from some finite domain* D. *A hitting set* H *of* Ω *is* $H \subseteq D$ *such that* $\forall S \in \Omega, H \cap S \neq \emptyset$.

Let $MSS(\Lambda)$ denote the MSSes of a set of clauses Λ. Let $MCS(\Lambda_{MSS})$ denote the MCSes of a set of MSSes Λ_{MSS}. Let $HIT(S)$ denote the irreducible hitting sets of a set of sets S. Then, as discussed previously, the MUSes of a set of clauses Λ is $MUS(\Lambda) = HIT(MCS(MSS(\Lambda)))$ [8].

Example 1. Let Δ_R be a rule base:

$f_1 : red \rightarrow fast$ $\quad$ $f_2 : fast \rightarrow \neg fuelEfficient$
$f_3 : offRoad \rightarrow expensive$ $\quad$ $f_4 : sporty \rightarrow (expensive \wedge (black \vee red \vee white))$
$f_5 : \neg expensive \rightarrow under\$20K$
$f_6 : cabriolet \rightarrow \neg bigCapacity$ $\quad$ $f_7 : fuelEfficient \rightarrow \neg offRoad$

Let Δ_F be a fact base:

$f_8 : red$ $\quad$ $f_9 : offRoad$ $\quad$ $f_{10} : \neg expensive$ $\quad$ $f_{11} : fuelEfficient$
$f_{12} : sporty$ $\quad$ $f_{13} : cabriolet$ $\quad$ $f_{14} : bigCapacity$

Let Δ be a knowledge base where $\Delta = \Delta_R \cup \Delta_F$.

We denote the set of clauses in the CNF equivalent of a propositional formula α as $CNF(\alpha)$. The set of clauses for a knowledge base K is $\bigcup_{\alpha \in K} CNF(\alpha)$, i.e., a knowledge base is represented as a set of clauses interpreted as a single formula by the conjunction of these clauses.

Example 2. Given the knowledge base Δ then $CNF(\alpha)$ for $\alpha \in \Delta$ is (clauses (resp. formulae) are denoted c_n (resp. f_n) where n is a unique identifier):

$f_1 : \{c_1 : \neg red \vee fast\}$
$f_2 : \{c_2 : \neg fast \vee \neg fuelEfficient\}$
$f_3 : \{c_3 : \neg offRoad \vee expensive\}$
$f_4 : \{c_4 : \neg sporty \vee expensive,$
$f_5 : \{c_6 : expensive \vee under\$20K\}$ $\quad c_5 : \neg sporty \vee black \vee red \vee white\}$
$f_6 : \{c_7 : \neg cabriolet \vee \neg bigCapacity\}$
$f_7 : \{c_8 : \neg fuelEfficient \vee \neg offRoad\}$
$f_8 : \{c_9 : red\}$
$f_9 : \{c_{10} : offRoad\}$
$f_{10} : \{c_{11} : \neg expensive\}$
$f_{11} : \{c_{12} : fuelEfficient\}$
$f_{12} : \{c_{13} : sporty\}$
$f_{13} : \{c_{14} : cabriolet\}$
$f_{14} : \{c_{15} : bigCapacity\}$

Converting a knowledge base to a set of clauses is necessary for applying existing methods for finding MUSes since a MUS is an inconsistent set of clauses.

Example 3. Let $\Theta = MSS(CNF(\Delta))$. There are 69 MCSes in Θ (of which 10 sets of clauses are shown below). A sample clause from each MCS is underlined to highlight a single MUS:

$$MCS(\Theta) = \left\{ \begin{array}{c} \{c_7, c_{11}, \underline{c_{12}}\}, \{c_{11}, \underline{c_{12}}, c_{15}\}, \{c_{11}, \underline{c_{12}}, c_{14}\}, \{\underline{c_2}, c_4, c_7, c_{10}\}, \\ \{\underline{c_2}, c_{10}, c_{11}, c_{14}\}, \{\underline{c_1}, c_8, c_{11}, c_{14}\}, \{\underline{c_9}, c_{10}, c_{11}, c_{14}\}, \\ \{\underline{c_2}, c_8, c_{11}, c_{14}\}, \{\underline{c_1}, c_{10}, c_{11}, c_{14}\}, \{c_8, \underline{c_9}, c_{11}, c_{14}\}, \ldots \end{array} \right\}.$$

Example 4. There are 5 MUSes from $MCS(\Theta)$. The sample MUS from Example. 3 is again underlined:

$$HIT(MCS(\Theta)) = \left\{ \begin{array}{c} \{\underline{c_{12}}, \underline{c_1}, \underline{c_2}, \underline{c_9}\}, \{c_{12}, c_8, c_{10}\}, \\ \{c_{11}, c_3, c_{10}\}, \{c_{11}, c_4, c_{13}\}, \{c_7, c_{14}, c_{15}\} \end{array} \right\}.$$

The underlined variables in Example. 3 and the underlined MUS in Example. 4 demonstrates the relationship between MCSes and MUSes, i.e., the MUS $\{c_{12}, c_1, c_2, c_9\}$ can be determined from the MCSes of Θ because the clauses c_{12}, c_1, c_2 and c_9 are the minimal set of clauses whose elements appear at least once in all MCSes. Therefore, a MCS contains at least one clause from each MUS.

Since a MUS is a set of clauses and a MIS is a set of formulae, it is necessary to determine the formulae from which each clause in the MUS originated (the set of FoMUSes of a MUS). Then, the set of MISes from an MUS is an irreducible hitting set of the set FoMUSes for that MUS. Therefore, the MISes of a MUS can be found using the same method used for finding a set of MUSes from a set of MCSes, i.e., $HIT(FoMUS)$ for each MUS.

Example 5. The FoMUSes for the 4th MUS $\{c_4, c_{11}, c_{13}\}$ are $\{\{f_4 : sporty \rightarrow (expensive \wedge (black \vee red \vee white))\}, \{f_{10} : \neg expensive\}, \{f_{12} : sporty\}\}$. So, the MISes for these FoMUSes can be determined by $HIT(\{\{f_4\}, \{f_{10}\}, \{f_{12}\}\})$:

$$\{\{sporty \rightarrow (expensive \wedge (black \vee red \vee white)), \neg expensive, sporty\}\}$$

2.1. Implementation

The main tasks performed when finding the MISes of a knowledge base are: converting the knowledge base to CNF; finding the set of MCSes from a set of clauses; and finding the set of MUSes from a set of MCSes (which is the same task performed when finding the set of MISes from a set of FoMUSes). In terms of computational complexity, conversion of an arbitrary propositional formula to CNF has complexity $O(2^n)$ in the worst case while SAT is a classic NP-complete problem. Additionally, the irreducible hitting set problem encountered in the tasks of finding MUSes from MCSes and MISes from FoMUSes is also NP-complete [8].

Finding the set of MUSes (resp. MISes) from a set of MCSes (resp. FoMUSes), denoted HIT, works by looping through each MCS and adding a clause (resp. formula) to a forming MUS, then any remaining MCSes containing this clause are removed. When the clause is added to a MUS it is forced to be 'essential' (removing it would leave at least one MCS unrepresented in the final MUS) by removing all clauses in the current MCS from all remaining MCSes. All MUSes for a set of MCSes are found recursively in this way. Either CAMUS or HYCAM algorithms can be used for these tasks and both have been shown to perform well in many complex cases [7,8]. We do not explain the other algorithms in detail because of space limitations.

Algorithm 1 Finding MISes in a knowledge base

Require: Knowledge base K
Ensure: Set of MISes
  $clauses \leftarrow \emptyset$
  for $formula \in K$ **do**
    for $clause \in CNF(formula)$ **do**
      if $mapping(clause)$ exists **then**
        $mapping : clause \mapsto mapping(clause) \cup \{formula\}$
      else
        $mapping : clause \mapsto \{formula\}$
      end if
      $clauses \leftarrow clauses \cup \{clause\}$
    end for
  end for
  $MUSes \leftarrow MUS(clauses)$
  $MISes \leftarrow \emptyset$
  for $MUS \in MUSes$ **do**
    $FoMUSes \leftarrow \emptyset$
    for $clause \in MUS$ **do**
      $FoMUSes \leftarrow FoMUSes \cup \{mapping(clause)\}$
    end for
    $MISes \leftarrow MISes \cup HIT(FoMUSes)$
  end for
  return $MISes$

Algorithm 1 is the formal process for finding the MISes of a knowledge base using existing algorithms for finding the MUSes of a single CNF formula [6,7]. In summary, each formula in the knowledge base is converted to CNF, the formula (or formulae) are mapped to each unique clause and a final set of clauses representing the complete knowledge base is produced. The MUSes for this set of clauses

are found using either CAMUS or HYCAM and, for each MUS, the originating formulae for each clause are found (FoMUSes). Then by applying the algorithm for finding the irreducible hitting set of a set of sets (HIT), the set of MISes for each MUS can be determined from the set of FoMUSes producing the MISes for the knowledge base. This process has been implemented in the MIMUS tool which can incorporate either CAMUS or HYCAM implementations and a combined analysis of the performance of this tool along with a tool for measuring inconsistency (MINC) is included in the following section.

3. Measuring the Inconsistency of Formulae

In the previous section we have shown that a practical implementation for finding MISes is possible. However identifying inconsistencies is only the first step in resolving conflicting knowledge, i.e., the reason it is important to identify inconsistencies is so that they can be handled in some way. One such method for inconsistency handling is to attempt to resolve inconsistencies, for example by deletion, weakening or splitting of individual formulae [12]. However in cases of large knowledge bases, or knowledge bases which have a large number of inconsistencies, there may be unmanageable numbers of inconsistent formulae so it then becomes a question of which formula should be addressed first. For this purpose, a number of inconsistency measures have been proposed [3,4] which quantify the responsibility of each formula for the inconsistency of the overall knowledge base. Intuitively, the more responsible a formula is for inconsistency, the sooner it should be resolved. Furthermore, when formulae have varying degrees of importance this can also be taken into account [5]. The practical benefit of these measures in relation to a network intrusion detection rule set is discussed in [1].

3.1. Inconsistency Measures

The Scoring function presented in [3] is the truest formula-level inconsistency measure since it is closely related to the concept of MISes, i.e., that an MIS can be resolved by removing any formula. The Scoring function therefore assigns a value to each formula based on the number of MISes that would be resolved if the formula were removed from the knowledge base. Looking at Fig. 1 it is clear, for the purpose of deletion-based resolution, that removing a formula which is involved in multiple MISes is more efficient than removing a formula which is involved in only one. By this intuition, the more inconsistencies a formula is involved in, the more problematic it is in relation to the knowledge base.

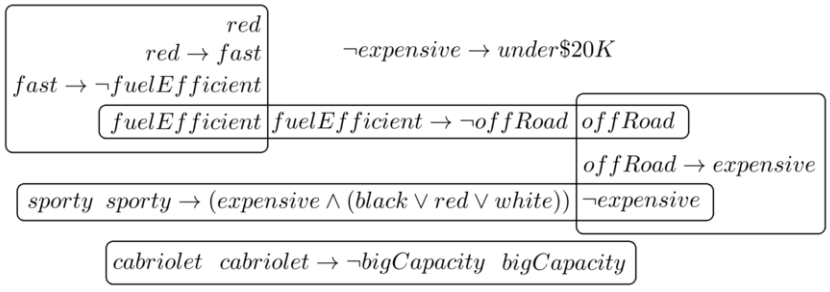

Figure 1. MISes in Δ where $fuelEfficient$, $offRoad$, $\neg expensive$ are most problematic.

Since the Scoring function is unable to discriminate in terms of the complexity and size of an inconsistency, a proportional inconsistency measure was proposed. The Shapley Inconsistency Value (SIV) [4] takes an inconsistency measure as a payoff function in coalitional form and, using the Shapley value from coalitional game theory, determines the proportional inconsistency for each formula in a knowledge base. In a knowledge base K, the SIV for $\alpha \in K$ calculates the sum of the inconsistency of every subset of the power set of K for which α is involved. As a logical property, when the Scoring function is applied as a Shapley inconsistency measure, denoted $\mathrm{S}_{I_{MI}}$ or MI Shapley, the result is to sum the proportion of each MIS from which a formula is a member.

Both the Scoring function and the MI Shapley measure are limited by the fact that they are unable to consider the relative degree of importance of a formula, i.e., if two formulae have the same inconsistency values but one formula is considered more important than the other, then it is logical that they should not be considered as equally problematic. For this reason, the Blame_v measure for prioritized knowledge bases was introduced. While this measure is also based on MISes, rather than assigning each formula with a single inconsistency value, it instead assigns a vector value representing the blame at each priority level for a formula. In order to demonstrate the benefit of prioritized measures on the example knowledge base, it is necessary to produce a prioritized version of Δ.

Example 6. Let Δ be prioritized as a Type-II[2] prioritized knowledge base Δ^P where $f_1, \ldots, f_{14}$ denote formulae from Δ:

$$\Delta^P = \langle \{f_1, f_2, f_3, f_4, f_5\}, \{f_6, f_7, f_8, f_9, f_{10}\}, \{f_{11}, f_{12}, f_{13}, f_{14}\} \rangle.$$

Therefore Δ^P has three priority levels where, for example, formulae $f_1 : red \rightarrow fast$ and $f_2 : fast \rightarrow \neg fuelEfficient$ are more important than $f_8 : red$, which is more important than $f_{11} : fuelEfficient$. By taking the relative priority of formulae into account, the Blame_v measure can produce an ordering for the degree of blame associated with each formula where the value is based on both the formulas involvement in inconsistency and its relative importance.

Finally, an extension of the Blame_v measure was introduced in [1], denoted Blame_l which takes into account the normalized number of atoms involved in inconsistency at each priority level. The rational being that if two formulae are of varying degrees of complexity, i.e., $f_4 : sporty \rightarrow (expensive \wedge (black \vee red \vee white))$ is more complex than $f_{12} : sporty$, then they should not be treated as equally problematic. The Blame_l measure therefore provides a deeper inspection of formulae involved in inconsistency.

3.2. Implementation and Performance

The Scoring, MI Shapley, Blame_v and Blame_l measures have been implemented in an automated tool, MINC (Measuring INConsistencies), which incorporates the MIMUS tool for finding MISes and which will be downloadable soon [13]. Given a propositional knowledge base, MINC can identify the inconsistencies (using MIMUS) and determine the degree of blame associated with each formula. The

[2] Let K be a knowledge base and K^P be the Type-II prioritized [5] equivalent with n priority levels. Let $K^P = \langle K_1, \ldots, K_n \rangle$ where $K_1, \ldots, K_n \subseteq K$ assigned to priority levels $1, \ldots, n$ respectively, where K_1 has the most important formulae, and K_n the least important.

result of applying MINC to the prioritized knowledge base Δ^P is shown in Table. 1 where f_{10} is identified as most problematic by Blame_v and Blame_l.

Formula	Priority	Scoring	MI Shapley	Blame_v	Blame_l
f_1	1	1	0.25	(0.02, 0.02, 0.02)	(0.00, 0.00, 0.01)
f_2	1	1	0.25	(0.02, 0.02, 0.02)	(0.00, 0.00, 0.01)
f_3	1	1	0.33	(0.00, 0.11, 0.00)	(0.00, 0.02, 0.00)
f_4	1	1	0.33	(0.00, 0.06, 0.06)	(0.00, 0.01, 0.01)
f_5	1	0	0.00	(0.00, 0.00, 0.00)	(0.00, 0.00, 0.00)
f_6	2	1	0.33	(0.00, 0.00, 0.11)	(0.00, 0.00, 0.03)
f_7	2	1	0.33	(0.00, 0.06, 0.06)	(0.00, 0.01, 0.01)
f_8	2	1	0.25	(0.04, 0.00, 0.02)	(0.01, 0.00, 0.01)
f_9	2	2	0.67	(0.06, 0.11, 0.06)	(0.01, 0.02, 0.01)
f_{10}	2	2	0.67	(0.11, 0.06, 0.06)	(0.04, 0.01, 0.01)
f_{11}	3	2	0.58	(0.04, 0.13, 0.00)	(0.01, 0.02, 0.00)
f_{12}	3	1	0.33	(0.06, 0.06, 0.00)	(0.03, 0.01, 0.00)
f_{13}	3	1	0.33	(0.00, 0.06, 0.06)	(0.00, 0.02, 0.01)
f_{14}	3	1	0.33	(0.00, 0.06, 0.06)	(0.00, 0.02, 0.01)

Table 1. Comparison of inconsistency measures for Δ^P rounded to 2 decimal places.

The complexity of creating efficient SAT solvers has resulted in extensive SAT benchmarks. Since the process of finding MISes described in this paper is based on using SAT solvers to determine the satisfiability of sets of clauses in a CNF formula, these benchmarks are particularly suitable for testing the performance of MINC. However since each SAT benchmark is just a single CNF formula, and the purpose of MINC is to measure inconsistent formulae in a knowledge base, it is necessary to convert this CNF formula into a knowledge base. For the purpose of these experiments, the SAT benchmark was assumed to be a knowledge base where each clause was assumed to be a formula, i.e., each formula is a single clause and each MUS is directly equivalent to a MIS.

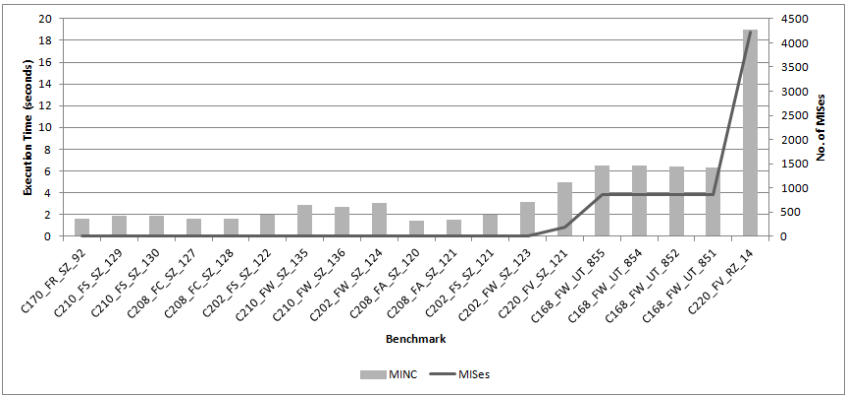

Figure 2. Measuring inconsistency of Benz benchmark set using MINC.

Figure. 2 shows the execution time of running MINC on knowledge base versions of the Benz [14] benchmark set. The test system was an Ubuntu Linux 10.04 Virtualbox virtual machine on a Windows 7 desktop PC with a 3.20GHz Intel Xeon quad core processor and 3GB of RAM. The execution time includes the

time it took MIMUS to find all MISes and MINC to calculate all four inconsistency measures. The performance of MIMUS is closely related to the performance of the implementation for finding MUSes so these results are omitted. In [15] it is argued that the ratio of clauses to variables is more useful for measuring the performance of SAT solvers than the total number of variables or clauses. No correlation was found with the total number of variables or formulae or with the ratio of formulae to variables. As expected however, the overall results for MINC show a clear correlation between the number of MISes in a knowledge base and the time it takes to calculate the inconsistency values of each formula. In this case, MINC performed well against the Benz benchmark set: the longest duration was 19.004 seconds for a knowledge base equivalent to C220_FV_92_14 with 1728 variables, 4508 formulae and 4224 MISes.

4. A Case Study: QRadar

QRadar [16] is an intelligent network security architecture developed by Q1 Labs/IBM, which provides exploit detection based on log data (e.g., application, operating system and firewall logs) as well as network traffic. Log data is analyzed by the QRadar system and normalized events, called *Events*, are generated. For example, if an application on the network produces a log message then QRadar generates an event with all the relevant information such as time, source address, destination address and message contents. Similar to log data, network traffic is also analyzed by QRadar and normalized events, called *Flows*, are generated.

For the purpose of exploit detection, the QRadar system is shipped with around 200 default rules which are used to classify log *Events* and network *Flows*. The default rules are designed to be customized for each unique network and users can create new rules to introduce new functionality. However, with this ability to modify and create rules comes the possibility of introducing inconsistencies into the rule set and these inconsistencies could have potentially serious implications for the systems functionality. Also, the larger a rule set becomes the more likely it is for inconsistencies to appear. For this reason, a method to automatically validate the rule set (after changes), and suggest a method to resolve any inconsistencies which may occur, would be an important feature.

In QRadar, knowledge is represented initially by these rules which are reasoned with only when facts are learned, i.e., rules are used to classify *Events* and *Flows* as they occur where facts are the *Events* and *Flows* themselves. This is true of many similar systems. However since a rule set is only a set of conditional statements, rules are seldomly inconsistent in isolation before facts are known. Often it is only when rules are applied to a set of facts that the rules themselves will create logical inconsistencies (see Θ-inconsistency [1] and rule inconsistency [17]). For the purpose of this case study, we add the minimal set of facts needed to trigger all rules in the rule set in order to produce a complete knowledge base.

4.1. Existing Rules and Potential Inconsistencies

All QRadar rules are created in a natural language format using a rule builder tool. This tool provides the ability to build rules from an existing set of natural language conditions which can be customized and combined for the desired rule. While there are additional actions which may be assigned to a rule, e.g., dispatching a new event, we will focus on the core behavior only.

We have the following anomaly-based rule (rule r_1):

```
Apply Anomaly: Potential Honeypot Access on events which are
detected by the Local system and when an event matches any of
the following BB:NetworkDefinition: Honeypot like Addresses.
```

This calls the following rule (rule r_2):

```
Apply BB:NetworkDefinition: Honeypot like Addresses on events
or flows which are detected by the Local system and when the
destination IP is a part of any of the following Bogon.
```

Name	Description
$Event(x)$	denotes that x is an event
$Flow(x)$	denotes that x is a flow
$Dest\_ip(x, y)$	denotes that the destination IP of x is y
$Honeypot\_like\_address(x)$	denotes that x is a honeypot like address
$Potential\_honeypot\_access(x)$	denotes that x is a potential honeypot access
$Honeypot\_safe(x)$	denotes that x is honeypot safe

Table 2. Predicates for set of QRadar rules.

Propositional logic is not sufficiently expressive to accurately represent these rules so we will represent them in first-order logic. Using predicates from Table. 2, the rules can be represented as:

$$r_1 : (\forall x) \begin{pmatrix} Event(x) \wedge Honeypot\_like\_address(x) \\ \rightarrow Potential\_honeypot\_access(x) \end{pmatrix}$$

$$r_2 : (\forall x \exists y) \begin{pmatrix} (Event(x) \vee Flow(x)) \wedge Dest\_ip(x, y) \wedge y \in bogon \\ \rightarrow Honeypot\_like\_address(x) \end{pmatrix}$$

These can be read as:

- r_1: If x is an event and x is a honeypot like address then x is a potential honeypot access
- r_2: If x is an event or flow and the destination IP of x is part of Bogon then x is a honeypot like address

Since users can create their own rules, the following would be possible:

- r_3: If x is an event or flow and the destination IP of x is part of Bogon and x is not a honeypot like address then x is honeypot safe

As a QRadar rule it could be written as (rule r_3):

```
Apply Anomaly: Honeypot Safe on events or flows which are
detected by the Local system and when the destination IP is a
part of any of the following Bogon and when an event matches
none of the following BB:NetworkDefinition: Honeypot like
Addresses.
```

Using predicates from Table. 2, this can be represented in first-order logic:

$$r_3 : (\forall x \exists y) \begin{pmatrix} (Event(x) \vee Flow(x)) \wedge Dest\_ip(x, y) \wedge y \in bogon \\ \wedge \neg Honeypot\_like\_address(x) \rightarrow Honeypot\_safe(x) \end{pmatrix}$$

Name	Description
$event_1$	denotes an event with a destination IP 192.168.1.137
$bogon$	denotes $\{192.168.1.100, 192.168.1.101, \ldots, 192.168.1.150\}$

Table 3. Constants for QRadar system.

In order to determine inconsistency we must instantiate (ground) these universally quantified formulae by constants representing a certain scenario (propositional case). Using the constants from Table. 3, these rules can be grounded in a propositional rule set $Q = \{r_1, r_2, r_3, bogon\}$ where:

$$r_1 : Event(event_1) \wedge Honeypot\_like\_address(event_1) \rightarrow Potential\_honeypot\_access(event_1)$$

$$r_2 : (Event(event_1) \vee Flow(event_1)) \wedge Dest\_ip(event_1, 192.168.1.137) \wedge 192.168.1.137 \in bogon \rightarrow Honeypot\_like\_address(event_1)$$

$$r_3 : (Event(event_1) \vee Flow(event_1)) \wedge Dest\_ip(event_1, 192.168.1.137) \wedge 192.168.1.137 \in bogon \wedge \neg Honeypot\_like\_address(event_1) \rightarrow Honeypot\_safe(event_1)$$

Currently the rule set Q is consistent but if the condition for r_3 were ever met, i.e., $r_3^{\mathcal{C}} : (Event(event_1) \vee Flow(event_1)) \wedge Dest\_ip(event_1, 192.168.1.137) \wedge 192.168.1.137 \in bogon \wedge \neg Honeypot\_like\_address(event_1)$, then the system would be inconsistent. Formally, $MI(Q \cup \{r_3^{\mathcal{C}}\}) = \{\{bogon, r_2, r_3^{\mathcal{C}}\}\}$ since the inconsistency is between r_2 and the *condition* of r_3. In practice however, there are likely to be two outcomes from adding r_3 to the rule set (depending on the order of rule execution and assumptions made by the reasoning engine):

- the rule engine assumes $\neg Honeypot\_like\_address(event_1)$ until it can be inferred otherwise. In this case, depending on the order of execution, it can infer $Honeypot\_like\_address(event_1) \wedge Potential\_honeypot\_access(event_1)$ or $Honeypot\_safe(event_1)$. Alternatively,
- nothing is assumed by the rule engine but r_2 is applied first since the condition for r_3 encompasses the condition for r_2. In this case, it will infer $Honeypot\_like\_address(event_1) \wedge Potential\_honeypot\_access(event_1)$, so r_3 will never be triggered but will remain in the rule set.

The significance of identifying this inconsistent set of rules is that depending on the reasoning engine, different conclusions may be reached ($event_1$ will be classified differently) or one rule will never be triggered. In relation to the QRadar system, it is obviously problematic that different conclusions can be reached from the same rule set. However, a non-triggering rule can also be considered problematic since there is likely to be a performance overhead. Other issues include the loss of functionality when a non-triggering rule is a prerequisite for another rule.

4.2. Implications for Automated Inconsistency Handling in QRadar

From this case study it is clear that a grounded first-order logic interpretation of the QRadar rule set is possible. This allows automated tools to be applied for identifying inconsistencies and formally measuring the degree to which each rule in the rule set is responsible for inconsistency. However the case study also raised an issue affecting any rule set (including QRadar) in relation to finding inconsistencies, i.e., that explicit inconsistencies may not occur until facts are learned. We demonstrated the performance of MINC in terms of complex synthetic knowledge bases which performed well in experimentation, i.e., it took 18 seconds to measure

the inconsistency of formulae in a knowledge base with 1728 variables, 4508 formulae and 4224 MISes. This suggests that the process of finding and measuring the inconsistency of QRadar rules can be automated with MINC after manually inserting the facts needed to trigger rules. However, if a tool was developed for this purpose then the entire process of detecting inconsistencies between rules in the QRadar system could be fully automated. A complete architecture is particularly justified because existing features for allowing users to create and modify rules do not include this type of inconsistency validation.

5. Conclusion

The main contributions of this paper can be summarized as:

- a new algorithm for finding the MISes of a knowledge base from the MUSes of a CNF formula has been proposed and implemented in a tool (MIMUS);
- a number of inconsistency measures, which identify an ordering for the degree of blame associated with each formula in a knowledge based, have been implemented in a tool (MINC); and
- a case study has been carried out on the existing QRadar network security rule set and it has been suggested how MIMUS and MINC could be applied in order to automate inconsistency handling.

Future work will focus on developing a tool to automatically insert triggering facts so that MIMUS and MINC can be used to identify and measure the inconsistency in a rule set such as QRadar. From there, we hope to propose methods to resolve inconsistencies using these tools on experimental data.

References

[1] K. McAreavey, W. Liu, P. Miller, K. Mu. Measuring inconsistency in a Network Intrusion Detection rule set based on Snort. *Int. J. Semantic Computing* 5(3): 281–322. 2011.

[2] K. Mu, Z. Jin, R. Lu, W. Liu. Measuring inconsistency in requirements specifications. In *Proc. of ECSQARU 05*:440-451. LNAI 3571, Springer, 2005.

[3] A. Hunter, S. Konieczny. Approaches to Measuring Inconsistent Information. In *Inconsistency Tolerance*, volume 3300 of LNCS, Springer, 189–234, 2004.

[4] A. Hunter, S. Konieczny. On the measure of conflicts: Shapley inconsistency values. *Artificial Intelligence* 174(14):1007–1026., 2010.

[5] K. Mu, W. Liu, Z. Jin. Measuring the Blame of each Formula for Inconsistent Prioritized Knowledge Bases. *Journal of Logic and Computation* (doi: 10.1093/logcom/exr002), 2011.

[6] M.H. Liffiton and K.A. Sakallah. On Finding All Minimally Unsatisfiable Subformulas. In *Proc. of SAT 05*, volume 3569 of LNCS, 173–186. Springer, 2005.

[7] É. Grégoire, B. Mazure, and C. Piette. Using local search to find MSSes and MUSes. In *European Journal of Operational Research*, volume 199, no. 3, 640–646, 2009.

[8] M. Liffiton and K. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. In *Journal of Automated Reasoning*, volume Online First. Springer, 2007.

[9] R. Malouf. Maximal Consistent Subsets. Comp. Lingu., vol. 33(2), 153–160. 2007.

[10] M. Liffiton. CAMUS source code. http://www.iwu.edu/~mliffito/camus/.

[11] C. Piette. HYCAM binaries. http://www.cril.univ-artois.fr/~piette/.

[12] J. Grant, A. Hunter. Measuring consistency gain and information loss in stepwise inconsistency resolution. In W. Liu (Ed.) ECSQARU 2011. LNAI 6717, 362–373. 2011.

[13] K. McAreavey. MINC source code. http://www.cs.qub.ac.uk/~kmcareavey01/

[14] M. Liffiton. CAMUS Results - Benz Benchmarks. http://www.iwu.edu/~mliffito/camus/results_benz.php.

[15] N. Gorogiannis, A. Hunter. Implementing semantic merging operators using binary decision diagrams. International Journal of Approximate Reasoning 49(1), 234–251. 2008.

[16] Q1 Labs/IBM. QRadar Overview. http://q1labs.com/products.aspx.

[17] P. Besnard. A Logical Analysis of Rule Inconsistency. *Int. J. Semantic Computing* 5(3): 271–280. 2011.