# Identifying Security-Critical Components with Attack Path Planning

SCS Honors Thesis
2016

**Author:**
Brandon Lum
Carnegie Mellon University
`brandonlum@cmu.edu`

**Advisor:**
David Brumley
Carnegie Mellon University
`dbrumley@cmu.edu`

School of Computer Science
Carnegie Mellon University

## Abstract

The security of any system is only as strong as its weakest link. Ideally, we would prefer to ensure that every component of a system is secure. However, it is not feasible to put resources into inspecting every component of a system as the cost to check for security vulnerabilities does not scale well. The current approach to Identifying Security-Critical Components is through attack path planning - finding out how an attacker can violate the confidentiality, integrity or availability of the system. An attack plan consists of a series of step an attacker can take to compromise the system. These steps include exploiting vulnerabilities in components of the system. However, we do not know what vulnerabilities are present in the system. Our task is to hypothesize (based on a potential attack plan) which vulnerabilities are the most important to check for in a system (checklist items). Our current system generates facts about the system using analysis tools on system images, infers additional information from a set of rules, and identifies attack plans via a simple backward search algorithm.

This thesis presents a new logic-based approach to tackling the problem. We start with an introduction of the problem, and next proceed to define our logic-based formulation of the problem based on natural deduction and datalog. We then provide an overview of the design, limitations and implementation of the system and test our new approach against our previous system. In this thesis, we briefly discuss our previous attempt at using Planning Domain Definition Language (PDDL), another approach to the problem , and why it was not a good fit.

Our new system reports a 2-3x speedup over the old system (at the minimum search depth configuration for each test case). However, we note that the system finds all possible attack goals, but not all possible attack plans. We conclude that our current approach is promising in scaling to multi-system attack plans and we explore future possibilities that may eliminate existing limitations of the system.

# Contents

# 1 Introduction

## 1.1 Motivation

The area of research focuses on the identification of weak points in a system. The security of any system is only as strong as its weakest link. Ideally, we would prefer to ensure that every component of a system is secure. However, it is not feasible to put resources into inspecting every component of a system as the cost to check for security vulnerabilities does not scale well.

## 1.2 Problem

One approach to Identifying Security-Critical Components is the use of attack path planning [1–5]. This involves identifying potential attack plans which could violate the confidentiality, integrity or availability of the system. From these attack plans, we determine the highest priority components to inspect for security vulnerabilities.

This research project was part of the DARPA Vetting Commodity IT Software and Firmware (VET) program, which focuses on the same goals on Commercial Off-the-Shelf (COTS) commodity Information Technology (IT) devices [6].

## 1.3 Attack Plans

More specifically, we define an attack plan [4] as a series of steps an attacker could carry out to compromise the system. These steps can include general deductions about the system (i.e. an ELF is a binary) and exploiting vulnerabilities in components of the system (i.e. a binary is memory corruptible).

We represent an attack plan as a single-source directed acyclic graph (DAG), where the root represents a state where an attacker has successfully gain unauthorized privilege of a system (this may be in the form of getting an interactive shell on the system, or reading a confidential file). The nodes in the tree represent facts about the system and what the attacker can achieve, and edges would represent deductions drawn from these facts.

Each node in the DAG is either

- A fact about the system (i.e. `/bin/ls` is a ELF file)
- A bug that can be exploited (i.e. `/bin/ls` is memory corruptible
- A conclusion from other facts or bugs in the system (i.e. `/bin/ls` executes code because it is a binary, `/bin/ls` executes shell commands because it is memory corruptible and it is a binary).

An example of an attack plan would be as follows:

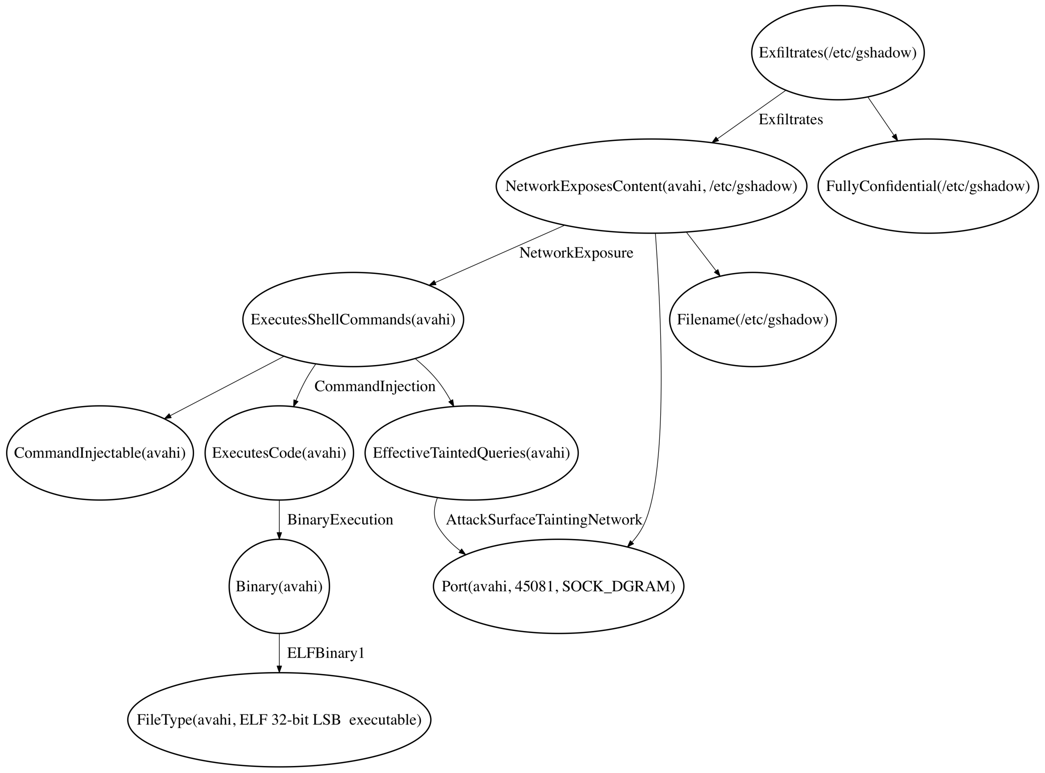
Exfiltrates(/etc/gshadow)
Exfiltrates
NetworkExposesContent(avahi, /etc/gshadow)
FullyConfidential(/etc/gshadow)
NetworkExposure
ExecutesShellCommands(avahi)
Filename(/etc/gshadow)
CommandInjection
CommandInjectable(avahi)
ExecutesCode(avahi)
EffectiveTaintedQueries(avahi)
BinaryExecution
AttackSurfaceTaintingNetwork
Binary(avahi)
Port(avahi, 45081, SOCK_DGRAM)
ELFBinary1
FileType(avahi, ELF 32-bit LSB executable)

Figure 1: Part of Tree of an attack plan

This attack plan shows that an attacker can exfiltrate a confidential file `/etc/gshadow` through service `/usr/sbin/avahi-daemon` using a command injection vulnerability.

A walk through of the attack plan is as follows. We prefix each explanation with the deduction on the edge label.

- `[ELFBinary1]` `/usr/sbin/avahi-daemon` has filetype `"ELF 32-bit LSB executable"`, therefore we can conclude that it is a Binary.
- `[BinaryExecution]` Since `/usr/sbin/avahi-daemon` is a binary, it executes code.
- `[AttackSurfaceTaintingNetwork]` Since `/usr/sbin/avahi-daemon` is accessible via port 45081 on TCP, an attacker can taint queries to the service by sending data to the socket.
- `[CommandInjection]` We note that `/usr/sbin/avahi-daemon` has a command injection vulnerability, executes code, and takes in tainted input from an attacker. Therefore, by exploiting the command injection vulnerability in the service, the attacker can execute command shells.
- `[NetworkExposure]` Since an attacker can execute shell commands on the system and there is a readable file `/etc/gshadow` on the system, the file `/etc/gshadow` can be exposed on the network through the shell.
- `[Exfiltrate]` Since an attacker is able to expose `/etc/gshadow` on the network and `/etc/gshadow` is a confidential file, an attacker can exfiltrate a confidential file `/etc/gshadow`.

## 1.4 Bug Hypotheses and Checklist Items

We note that an attack plan can only be valid with the assumption that we know the vulnerabilities in a system. Ironically, we do not know what the vulnerabilities are and finding those potential vulnerabilities is exactly our goal.

What we can do instead is create a hypothesis on the possible vulnerabilities in a system. Using the hypothesized vulnerabilities, we can show how an attack plan can be achieved. The output of the system would then be the hypothesized vulnerabilities needed to generate attack plans. We call these hypothesized vulnerabilities “bug hypotheses”. We use the term “checklist items” synonymously as these are the bugs we want to check for in a system.

Example of checklist items as follows:

- Check that `/usr/sbin/avahi-daemon` does not have a memory corruption vulnerability
- Check if there is a backdoor in iptables that allows access to port 776
- Check that `/usr/sbin/rpcbind` does not have a command injection vulnerability

In this paper, we focus on generating the bug hypothesis and attack plans, while continuing to examine we will visit the idea of prioritization of checklist items in future work.

## 1.5 Our Work

In this paper, we formalize the problem by presenting a logic formulation of the problem based on model checking of attack graphs [4, 5, 7]. Using this, we designed, implemented and evaluated a system to generate attack plans and checklist items.

We discuss the results from our evaluation as well as a previous attempt to solving the problem with a classical planning approach using Planning Domain Definition Language (PDDL) [8]. Concluding, we suggest some ideas for further development of the system.

# 2 Related Work

We review two areas of related work. 1) Attack model representations - as they are the foundation of attack plans. 2) Other work in obtaining attack plans and corresponding bug hypothesis/checklist items in a system or network of systems.

## 2.1 Attack Model Representations

There are many ways to represent an attack model. In attack path planning, attack graphs and model checking are a popular choice of attack model representation. In addition, attack grammars are a fairly new inclusion. We will explore these 3 model representations.

Attack Graphs are graph-based approach to network vulnerability analysis that “show the way an attacker can compromise a network or host [9].” In an attack graph, each node represents a possible attack state and each edge represents an action by the attacker [10, 11]. Reachability of a privilege node entails a successful attack.

Model Checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system [12]. Model checking is used in various fields of security [13–16]. Relevant to our context, Ou et al. shows that model checking can be used to represent attack graphs, by directly illustrating logical dependencies among attack goals and configuration information [4, 5, 7]. In this thesis, we use model checking as our attack model representation.

Attack grammars are grammars that “model and analyze network attack sequences” [17], and are very suitable for modeling attacks in Intrusion Detection Systems (IDSes). An advantage of these over attack graphs are that visualization of attack graphs can be rather complex and hard to understand [18].

## 2.2 Attack Plan Generation/Analysis

There are several existing approaches and systems which perform similar (if not identical) tasks.

MulVal (by Ou et al.) [4, 5], which is very closely related to our system, is an end-to-end framework and reasoning system that conducts multihost, multistage vulnerability analysis on a network. It uses model checking to represent attack graphs and does hypothetical analysis (similar to our bug hypothesis/checklist items).

Z. Liu et al. [2] presents graph generation method for complex network using an attack graph. Key nodes are first identified and the algorithm combines greedy policy, forward exploration and backward searching to generate the attack graph.

Raytheon BBN [3] created Theseus, which automatically maps and explores the firmware/software architecture of a system to generate attack plans, accompanied by a prioritized list checkliste items, for the device. Theseus combines static program analysis, attack graph generation algorithms, and a Boolean satisfiability solver to automate the checklist generation workflow.

O. Sheyner et al. [1] presents automatic analysis of attack graphs through a technique called minimization analysis. The analysis is used to help an analyst determine a minimal set of atomic attacks that must be prevented to stop intrusion by treating the attack graph as an Markov Decision Process (MDP) and finding an optimal policy using value iteration on attack graphs.

# 3 Logic Formulation

We will now look at the formulation of the model for our approach. We model our system based on ideas in logic.

## 3.1 Model: Facts and Rules

Our model is made out of 4 main components (actually 3, since the 4th being a special case of the rest). We first define each component, followed by a formal type definition.

### 3.1.1 Model Description

- An *atom* $a$ is a component of a system. All atoms are assigned a unique identifiable ID. Examples of atoms are files, binaries, kernel modules, etc.

- A *fact* $f$ is a predicate which describes a property about the system and/or attack. For example, the fact `Filename (301, /bin/ls)` describes that atom 301 has a filename of `"/bin/ls"`. Another example `Filetype (301, ELF)` says that atom 301 has a filetype of `"ELF"`. A fact may take on variables as parameters i.e. `Filename(X,Y)`, where `X` and `Y` are variables. These variables can be used as bindings or wildcards. We note that each parameter of a fact has a parameter type that specifies domain of values it can take on. Examples of these domains include the domain of all atoms (`hID`), integers (`hIntegers`), booleans (`hBool`), strings (`hString`), etc.

- A *rule* $r$ takes in a set of facts and concludes an additional fact about the system. I.e. Given a precondition, consisting of a set of facts, we can make deductions about a system. The purpose of rules are to deduce additional (and more meaningful) facts about the system.

  We write rules in the following format:

$$\frac{F_1, F_2, F_3, \ldots, F_n}{C_1}$$

  Where $F_i$s are facts (which may contain variables) and $C_1$ is the concluded fact. Let us look at some examples.

$$\frac{\texttt{Filetype(X, "ELF")}}{\texttt{Binary(X)}}$$

(ELFBinary)

This rule states that if there is a file $X$ that has a filetype `"ELF"`, then $X$ is a binary.

$$\frac{\texttt{Binary(X)}}{\texttt{ExecutesCode(X)}}$$

(BinaryExecution)

This rule states that if there is a binary $X$ then $X$ executes code.

Given the above 2 rules, and the fact `Filetype (301, ELF)`, we can derive the fact `ExecutesCode(301)`. We can invoke ELFBinary which gives us the fact `Binary(301)`, which we can then invoke BinaryExecution to get `ExecutesCode(301)`.

$$\frac{\texttt{ServesDirectory(X, Z, S)} \quad \texttt{DirectoryMember(Y, Z, S)}}{\texttt{ServesFile(X,Y)}}$$

(DirectoryToFileServing)

The above rule describes that if $X$ serves directory $Z$ on system $S$, and $Y$ is a directory member of directory $Z$ on system $S$, then $X$ serves file $Y$.

- A *bug hypothesis b* is a fact (that is associated with an atom), that indicates that there is a flaw in the system. These special facts are key to the generation of the attack plans.

  Bug hypothesis are facts like the following `MemoryCorruptable(X)`

  I.e. File X is memory corruptable - can be buffer overflowed..

In our model, a bug hypothesis can be derived by a rule. We call these bug hypothesis rules. The bug hypothesis facts are not concluded unless we choose to use a bug hypothesis. For example, it only make sense that an atom is `MemoryCorruptable(X)` if it is a binary. Thus a bug hypothesis rule would be:

$$\frac{\texttt{Binary(X)}}{\texttt{MemoryCorruptable(X)}}$$

(MemoryCorruptableGen)

These bug hypothesis can then be used to derive facts about what an attacker can do to a system with rules such as the following:

$$\frac{\texttt{Binary(X)} \quad \texttt{MemoryCorruptable(X)} \\ \texttt{AttackerEffectiveTaintedQueries(Y, X, Z)}}{\texttt{AttackerExecutesShellCommands(X, Y)}}$$

(MemoryCorrupt1)

### 3.1.2 Model Type Definitions

```
type ptype = hString | hInteger | hID | hSystemID | hBool
type pvalue = string | Variable of string

type fact = string * (pvalue * ptype list)
type rule = fact list -> fact option

type bughyp = fact
type bughyp_rule = rule
```

## 3.2 Problem: Identify Attack Plans from Bug Hypothesis

Given a problem instance $P$, we want to find attack plans and the corresponding bug hypothesis required to execute those plans. A *problem instance* $P$ is defined by a tuple $(I, G, R, R_B)$. Where $I$ is the initial state, $G$ is the goal of the problem instance and $R, R_B$ are the respective set of rules and bug hypothesis rules for the problem.

### 3.2.1 Initial State

We define the initial state as consisting of raw base facts as well as initial attacker facts. Intuitively this should correspond to the original state of the system and the initial position/status of the attacker. The initial state $I = F_{raw} \cup F_{atk}$. Where $F_{raw}$ are the raw base facts and $F_{atk}$ are the initial attacker facts.

**Raw Base Facts:** Base facts consists of a set of facts $F_{raw}$ which represent the state of the original system (without bug hypotheses). *Raw base facts* are facts gathered via processing of the images of the system to get information such as files, permissions, binaries, etc. This is done using linux tools as well as disassembly and program analysis tools such as IDA. For some information that may not be able to be detected by automated tools, we are able to manually add them to the database of raw base facts.

**Initial Attacker Facts**: We note that in a given system, we can make different assumptions about an attacker. The *initial attacker facts* are a set of facts $F_{atk}$ which encapsulate the initial position/status of the attacker. For example, an attacker may be internal (having access to the local network, server access, etc.). Alternatively, an attacker may be from the outside and thus, would only have access to the public network. These assumptions about the attacker can be encoded as facts such as the following:

```
AttackerNetworkAccess(N)
AttackerReads(X)
AttackerWrites(X)
AttackerPortAccessible(P,S)
AttackerControlsSystem(X)
```

For example, `AttackerNetworkAccess(40002)` where 40002 represents the internal network, gives us an initial state where the attacker has access to the internal network.

We note that all facts above should not contain any variables.

### 3.2.2 Goal

A goal of a problem instance is made of 2-tuple $(f_G, R_G)$, a *goal fact* $f_G$ is a fact representing the goal, as well as a set of rules $R_G$ describing how to derive the goal fact $f_G$.

We note that a goal fact should represent some sort of malice the attacker can perform. For example, we say the fact `Exfiltrate(X)` represents that an attacker can exfiltrate an important file $X$ that he should not be able to access.

Accompanying this goal fact domain would be a set of rules. in this example, we have just one rule describing possible ways an attacker can exfiltrate a file. An example for the `Exfiltrates(X)` goal fact is as follows:

Goal Fact: `Exfiltrates(X)`

$$\frac{\texttt{AttackerReads(Y)} \quad \texttt{FullyConfidentialFile(Y)} \quad \texttt{Filename(Y,X)}}{\texttt{Exfiltrates(X)}}$$

(ExfiltratesRule)

The above rule says that an attacker exfiltrates filename $X$ if there is an atom $Y$ with filename $X$, and that atom is confidential, and an attacker can read it.

#### 3.2.3 Finding Bug Hypotheses and Attack Plans

Given a problem instance $(I, G, R, R_B)$, where $G = (f_G, R_G)$. We let $R_*$ be the set of all the rules, defined by $R_* = R \cup R_G \cup R_B$. Very broadly, we want to ask the question, "Given facts $I$ and rules $R_*$, give all derivations of $f_G$".

$$\frac{\vdots}{f_G}$$

## 4 System Design and Implementation

Our system design and implementation was largely influenced by our previous system. The following is an overview of the entire system design and workflow. The input of the system is a disk image of the system to analyze. We label the disk image as the "Image".

We note that in this diagram, the soft rectangular blocks represent the procedures, the database symbols represent collections of facts and the multiple documents symbols represent logic facts/rules definitions.

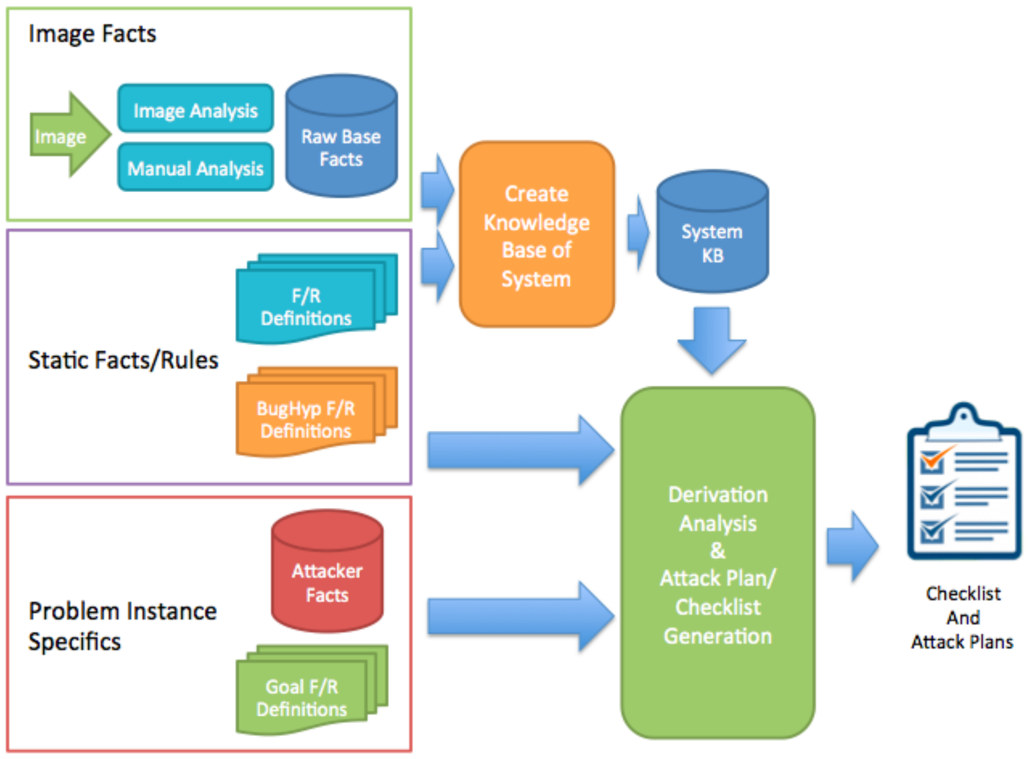

Figure 2: System Design Overview

## 4.1 Building Blocks of our System

For each problem instance $P$, there are many sub-parts of the problem. We organize the different parts of the problem into 3 groups of components in our system: Image Facts, Static Fact/Rules and Problem Instance Specifics. The purpose for this segregation is to 1) enable convenient abstraction of interfaces and 2) allow staging of computation in our system based on the modification frequency and patterns across problem instances.

We call these groups the building blocks of our system, as we will use them as part of our core procedures in generating the checklist items and attack plans.

### 4.1.1 Image Facts

The image facts consists of the raw base facts of the system using 2 different processes. The first process is Image Analysis, which utilizes linux tools, as well as disassembly and program analysis tools such as IDA, to generate information like file reads, references, file permissions, file types, etc. However, not all important facts about the system can be generated autonomously. For facts that cannot be generated autonomously, we have a web interface that allows us to manually add facts to the raw facts database (which we call Manual Analysis). An example of a scenario of this is the tagging of certain files as confidential based on the written/verbal specifications of the systems.

### 4.1.2 Static Facts/Rules

Our system is statically configured with a list of fact/rule definitions for performing deductions as defined in the Section 3, and a list of fact/rule definitions for the bug hypotheses. We do not expect frequent changes to these definitions.

### 4.1.3 Problem Instance Specifics

Our system is configured with multiple problem instances, each indicating the type of malice that we are trying to identify. These encompass the initial attacker facts, which are part of the initial state, and the goal facts/rules definitions. We expect to continuously refine and add/modify problem instances to detect different types of malice.

## 4.2 System Procedures

The procedures to obtain the image facts have been covered earlier. As seen, we will examine the other two procedures in the system design, which make up the majority components. To be able to do this, we need to address how we intend to solve the question we defined in Section 3.2.3.

### 4.2.1 Solving the Problem

We will approach this problem using natural deduction [19]. More specifically, we aim infer the goal state $f_G$, given our initial state facts $I$ , and rules $R_*$. We will make use of logic machinery called horn clauses [20]. Horn clauses are logical formulas of a particular rule-like form. By setting up our problem as a set of horn clauses, we can use logical resolution to perform inference of the goal.

We are able to take advantage of the way the problem is set up to directly map our problem instance to horn clauses (facts are the initial state $I$, rules are $R_*$ and the goal is $f_G$). Therefore, being able to infer the goal state $f_G$ is equivalent to a horn clause resolution.

More specifically, we want to use a decidable fragment of Horn logic called Datalog [21]. We note that Datalog arises from Horn logic via two restrictions and an extension. It is thereby beneficial that our facts and rules meet this requirement on restrictions.

- **Restriction 1: Finite Domains:** Function symbols are disallowed, terms must be variables or drawn from a fixed set of constant symbols. Favorably, our facts and rules do not use function symbols (i.e. we only draw from a fixed set of constant symbols - which are system atoms and miscellaneous labels), and therefore we have a finite domain.
- **Restriction 2: Variables:** Any variable in the head of a clause also appears in the body. Our rules follow this property where every variable in the postcondition of a rule is in the precondition.

Being able to represent our problem in Datalog is beneficial since the complexity of a Datalog program is known to be polynomial in the size of input (since our datalog program is fixed) [22].

### 4.2.2 Knowledge Base Creation

We note that the Image facts and the static facts/rules do not change often. Therefore we can generate a collection of "processed facts" by applying the static rules on the raw base facts of the system. This can be done by putting all the facts and rules in datalog and performing forward chaining [20]. This set of facts, which we call the system knowledge base (KB), can be re-used across problem instances.

### 4.2.3 Derivation Analysis and Attack Plan Checklist Generation

We note that in order obtain the knowledge base of the problem instance, we need to augment the system knowledge base with 1) the attacker initlal state, and 2) the goal rules. Thereafter, finding out if there is a derivation $f_G$ is just a simple query of $f_G$. This would return the set of goal fact instances possible.

However, this is insufficient. We need to additionally generate an attack plan and checklist items. We note that the attack plan is equivalent to the derivation of an instance of the goal fact, and checklist items would be the corresponding bug hypotheses used in the derivation to prove that goal fact.

We obtain the derivation by first asking for a proof from the Datalog engine. However, we note that the proofs provided by an automatic theorem prover usually contains many facts that were not required in the actual deduction [23]. Therefore, we proceed by converting the proof of facts into a attack plan and perform attack plan minimization through reachability analysis on the attack plan (DAG). From this attack plan, we filter the bug hypothesis facts in the graph to determine the checklist items. This leaves us with the attack plan and the checklist items, which is the information we require.

## 4.3 Limitations of System

We note that a problem instance may have multiple derivable goals, and for each goal, multiple attack plans/graphs (that may be associated with different checklist items).

We note that the system will find all possible derivable goals, but will not generate all possible attack plans for each derivable goal. This is because the system relies on the derivation of the goal (proof) provided by the Datalog engine to create an attack plan. However, the Datalog engine needs only to provide one possible derivation for the proof to be valid. Thus, we are not guaranteed that our system will produce all derivations of the goals. Getting all possible attack plans will require more work, which will be discussed further in Section 7.

## 4.4 Implementation Specifics

For image analysis, we used a suite of tools consisting of IDA, readelf, strings and python. We processed each file in the image and dumped the results in a postgres database. For manual analysis, we used an application written with python flask.

For the Knowledge Base Creation and Derivation Analysis, we used pyDatalog in python for the Datalog engine. We obtained the list of facts from the goal deduction via verbosity of pyDatalog logging and we used a combination of python and PostgreSQL to generate an attack plan and perform reachability analysis.

In total, the system consisted of a total of 15,000 lines of code.

# 5 Evaluation

## 5.1 Testing

We now define the testing dataset and test cases that we will be using. The testing environment is a modified `ubuntu-trusty-64` vagrant box running on a 1.3 Ghz Core i5, solid state disk, with a memory provision of 4GB and host operating system Mac OS X. Unless specified otherwise, the testing setup will be the same for all approaches.

### 5.1.1 Testing Dataset

We used the data from the DARPA VET-TA1 engagement 3 day 2 for all results in this thesis. More specifically, we use system 1 of the given images, which was named “console”, as part of a 2 system setup with the following network diagram (a bigger version can be found in Appendix C).

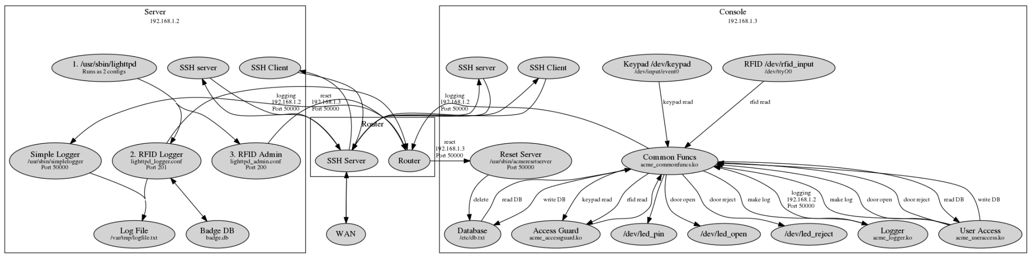

Figure 3: Network Diagram for VET Engagement 3 Day 2

Several details about the system include:

- 3015 atoms
- 48 config files
- 35 file types
- 232 unique files [1]

### 5.1.2 Test cases

The test cases were chosen based on the two main types of problem instances. The first is a generic test case: `info_leak` targets a broad purpose and is defined to have a deeper derivation/search tree. On the other hand, the other test case: `atk_shell` targets a rather specific action, and the rules that have been defined with a shallow derivation/search tree (i.e. most preconditions are base facts or direct bug hypothesis).

**Test Case:** `info_leak`

This source of malice comprises attacks where an attacker exfiltrates sensitive information from the target system. This is designed to cover "confidentiality" in the CIA taxonomy.

Goal Fact: `Exfiltrates(EXFILTRATED_FILENAME)`

[1]Unique files are files that are not commonly known. I.e. from daemons or operating systems

$$\frac{\begin{array}{c}\texttt{NetworkExposesContent(SERVER, PORT\_NUM, EXFILTRATED\_FILE)}\\ \texttt{FullyConfidentialFile(EXFILTRATED\_FILE)}\\ \texttt{Filename(EXFILTRATED\_FILE, EXFILTRATED\_FILENAME)}\end{array}}{\texttt{Exfiltrates(EXFILTRATED\_FILENAME)}}$$

(Solver1)

$$\frac{\begin{array}{c}\texttt{Port(SERVER, SERVER\_NAME, PORT\_NUM, PROTOCOL, SYSTEM)}\\ \texttt{PortAccessible(PORT\_NUM, SYSTEM)}\\ \texttt{ServesFileUnderAuthentication(SERVER, EXFILTRATED\_FILE)}\\ \texttt{AuthenticatedOnlyConfidentialFile(EXFILTRATED\_FILE)}\\ \texttt{Filename(EXFILTRATED\_FILE, EXFILTRATED\_FILENAME)}\end{array}}{\texttt{Exfiltrates(EXFILTRATED\_FILENAME)}}$$

(Solver2)

$$\frac{\begin{array}{c}\texttt{ReadsHardwareInput(READER, DEVICE)}\\ \texttt{HardwareExposesContent(READER, EXFILTRATED\_FILE)}\\ \texttt{FullyConfidentialFile(EXFILTRATED\_FILE)}\\ \texttt{Filename(EXFILTRATED\_FILE, EXFILTRATED\_FILENAME)}\end{array}}{\texttt{Exfiltrates(EXFILTRATED\_FILENAME)}}$$

(Solver3)

**Test Case:** `atk_shell`

Source of malice is for control flow hijacks, which may be any binary in the system.

Goal Fact: `AttackerGetsShell(CFH_FILENAME)`

$$\frac{\begin{array}{c}\texttt{Binary(CFH\_FILEID)}\\ \texttt{Port(CFH\_FILEID, SERVER\_NAME, PORT\_NUM, PROTOCOL, SYSTEM)}\\ \texttt{MemoryCorruptable(CFH\_FILEID)}\\ \texttt{Filename(CFH\_FILEID, CFH\_FILENAME)}\end{array}}{\texttt{AttackerGetsShell(CFH\_FILENAME)}}$$

(Solver1)

## 5.2 Test Methodology

In our experiment, we test our new system against our old system. We run both systems with the same test cases and measure the time taken. We then compare the checklist output of the two systems. For reference, we first give a brief description of the old system and the selection of parameter configuration for the tests.

### 5.2.1 Old System: Backwards Goal Search with Unification

The backwards search algorithm is defined recursively. It takes in the fact that it tries to prove. Next, it iterates through each rule that could prove the fact, and for each of the preconditions, it recursively calls on the fact to prove the fact for the rule. The recursively call returns the subset of values that are provable for that fact we asked for. Using this return value, the algorithm does some optimization by unifying the variables of the rule. At some point, the algorithm may face a situation where it requires a bug hypothesis to be present to prove a fact. The algorithm can then create a bug hypothesis to help prove the fact in question. The base case of the algorithm is when a fact is already in the database (in the initial state), or when we use a bug hypothesis.

The algorithm is configured with a maximum depth, `PLANNING_DEPTH`, of recursion and a maximum number of bug hypotheses, `MAX_CHECKLIST`, that can be created.

### 5.2.2 Old System Time Measurements

To account for the configurable parameters in the old system (`PLANNING_DEPTH` and `MAX_CHECKLIST`), we measure two timings. One is the minimum (Min.) timing, which is defined as the time taken for the system to get all checklist items with the minimum depth/checklist setting for each test case. The second is the saturated (Sat.) timing, which is defined as the time taken for the system to complete a run that explores the entire search tree of each test case.

We determine the saturation setting/time by incrementally increasing the depth and checklist items until we no longer see an increase in the time to complete running the algorithm.

### 5.2.3 New System Measurements

We measure the time taken to obtain all derivations of the goal (and its respective proof), and the time taken to generate the minimized attack plans separately. We add the two time measurements together to get the total time taken.

## 5.3 Results

### 5.3.1 Timing Results/Analysis

We first compare the timing of the systems:

| Test Case | New | Old (Min.) | Old (Sat.) |
|---|---|---|---|
| `info_leak` | 24s | 75s | 579s |
| `atk_shell` | 16s | 34s | 34s |

Figure 4: Timing Data across systems

For more information on the test data in determining the Min. and Sat. timing of the old system, refer to appendix A.

We note that for the new system, we measured the time taken for the goal derivation and the time taken for generating the minimized attack plan separately. For each case, the timings are 21.2s/2.2s (`info_leak`) and 15.5s/1.2s (`atk_shell`) respectively.

We present the following visualization of the differences in times:

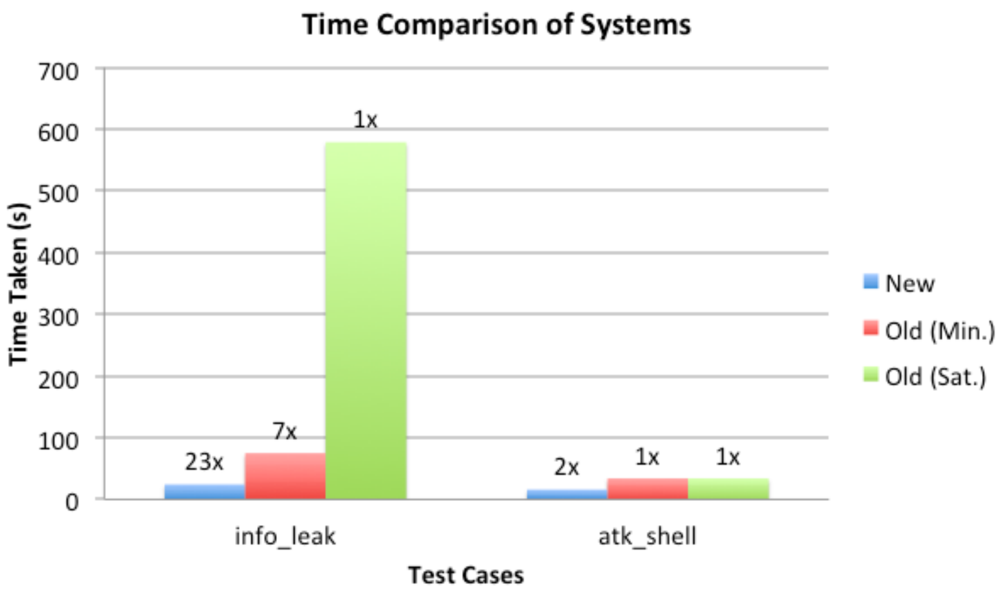

Figure 5: Time comparisons across systems (Multipliers above bars denotes the speed-up with respect to Old (Sat.))

### 5.3.2 Checklist Items Results/Analysis

The following are the results for the comparison of checklist items found. We note the following legend:

- Cmd Inj. - `CommandInjectable` Bug
- Mem. - `MemoryCorruptable` Bug
- Iptables - `IptablesBackdoor` Bug

| | New System | | | Old System | | |
|---|---|---|---|---|---|---|
| Test Case | Cmd Inj. | Mem. | Iptables | Cmd Inj. | Mem. | Iptables |
| `info_leak` | 0 | 4 | 6 | 4 | 2 | 6 |
| `atk_shell` | 0 | 4 | 0 | 0 | 4 | 0 |

Figure 6: Comparison of bug hypotheses found with systems

For the `info_leak` test case, both systems found the same 6 `IptablesBackdoor` bug. The new system found 2 additional `MemoryCorruptable` bugs over the old system, but did not find the 6 `CommandInjectable` bugs that the old system found. For the `atk_shell` test case, both systems found the 4 `MemoryCorruptable` bugs.

## 6 Discussion

In this section, we start by discussing some interesting results with regards to checklist items and timings. We then discuss one of the approaches we had taken prior to coming up with our new formulation/system and why it was not a good fit.

### 6.1 Checklist Results Analysis

Two interesting details from the `info_leak` results that we gathered were 1) Missing Checklist Items - our new system did not get the entire class of `CommandInjectable` bugs and 2) Additional Checklist Items - our new system got 2 additional `MemoryCorruptable` bugs.

#### 6.1.1 Missing Checklist Items

Missing checklist items were due to the limitation of the system as explained in Section 4.3. In this specific instance, it was due to the exploitation of the `CommandInjectable` bug leading to the same goal fact as the exploitation of the `MemoryCorruptable` bug. Therefore, the datalog system only provided the derivation of the goal that used the `MemoryCorruptable` bug instead of the `CommandInjectable` bug.

#### 6.1.2 Additional Checklist Items

We note that not finding the other 2 `MemoryCorruptable` bugs may be due to a potential bug in the old system. However, from this, we also noticed some interesting observations about the way the pyDatalog engine proves derivations. Specifically, we note that although only one `MemoryCorruptable` bug was needed to reach the goals. It found all 4 of them.

We hypothesize that this is due to the way the pyDatalog engine performs resolution. When the engine invokes a rule, it applies it to the current universe of facts, generating all possible facts deduced by the rule. Therefore, if it derives a goal fact from a particular derivation tree, it would create all possible instances of that derivation.

## 6.2 Timing Results Analysis

For the `info_leak` test case, the new system ran 3 times faster than the old system (at minimum settings required to get the 14 checklist items (MD:3, MC:2)). Compared to the old system at saturation, the new system ran 22 times faster. For the `atk_shell` test case, the new system ran 2 times faster than the old system (at minimum settings required to get the 4 checklist items) and at saturation (MD:3, MC:1).

This reflects the difference in complexity between the backward search method which runs into a combinatorial explosion, and the polynomial data complexity of Datalog [22] on our test set. In addition, performing the minimization on the attack plan in the new system is just a reachability analysis, which is also polynomial.

We also note that there are areas of improvement that can be done with regards to processing the attack plan, although this was not the main bottleneck. In particular, we could use an in-memory database such as memsql [24]. In addition, we could perform database indexing to speed up query times of the database.

## 6.3 Planning Domain Definition Language (PDDL)

In this section, we explore an approach we tried prior to our current formulation of the problem, and explain why it was not a good fit.

### 6.3.1 Motivation for PDDL

Planning, a branch of artificial intelligence that concerns the realization of strategies or action sequences, is a well researched area in Computer Science. We note that a classical planning problem includes an initial state, durationless and deterministic actions, and a single agent [25].

At first glance, our problem seems like a good fit for classical planning. We have an initial state, rules (actions) and an attacker (a single agent). In addition, we note that classical planning methods were used in similar tasks in another area of security, malware analysis. Sohrabi, S. et al. [26] presented Hypothesis exploration for malware detection using planning and showed a significant improvement over prior work focused on finding a single optimal plan.

Thus, we decided to explore the approach of translating our problem into a planning language. We used the Planning Domain Definition Language (PDDL) [8] for the formulation for our problem.

### 6.3.2 PDDL Solvers Used

We explored various PDDL Solvers: 1) Blackbox [27], 2) AppPlan.jar [28], 3) The LAMA Planner [29] and 4) Fast Downward [30]. In addition, we created two small PDDL toy examples and ran it on the various planners. These toy examples had a plan path length of 5 steps, and less than 10 actions and objects.

We eliminated Blackbox due to an unresolvable segfault in one of the examples, while AppPlan.jar could not find the short 5 step plan. The LAMA Planner and Fast Downward planner ran perfectly on the examples. We chose to use Fast Downward planner as it was an newer, extended version of the LAMA Planner.

### 6.3.3 Translation to PDDL

Using the same dataset from the evaluation section, we noted that it was impossible to hand-transcribe the PDDL problem without error. Therefore, we implemented a translator which would take the current data structures and generate a PDDL problem given a problem instance.

We did the translation in the following way:

- Domain PDDL Types followed our model ptypes

- Domain Consts were pre-processed from the set of rules
- Domain Actions were translated from rules where the preconditions were the body and the postcondition was the head of the rule.
- Problem Objects were pre-processed from the set of initial facts
- Problem Initial State were facts from forward chaining of initial facts [2]
- Problem Goal state was set to a unified goal (see below)
- A metric `total-cost` was created to minimize representing the number of bug hypotheses used
- Problem was specified to minimize `total-cost`
- Bug hypotheses were represented as actions with the empty precondition and the postcondition consisted of the bug hypothesis fact and an increase in `total-cost` by 1.

Goal State Specification: We create an additional empty goal predicate to resolve all instances of the goal fact. An example of this is as follows (for our `info_leak` example):

```
(:action ExfiltratesSolverResolve
   :parameters (?EXFILTRATED_FILENAME - hString)
    :precondition (and (Exfiltrates ?EXFILTRATED_FILENAME))
    :effect (and (ExfiltratesX ))
)
```

An example excerpt of the translated PDDL domain and problem can be viewed in Appendix D.

### 6.3.4 Grounding (Out Of Memory) Issues

Running the PDDL solver on `info_leak` resulted in a segfault caused by running out of memory after 2 minutes. We investigated and found the issue to be a "grounded representation of the problem, i.e., the total number of

[2]The forward chaining is only done once. We note that doing forward chaining first would help to reduce the amount of work done by the PDDL solver.

ground facts, ground actions and ground axiom rules that can be reached by a relaxed reachability analysis from the initial state" [31]. One of the ways to combat the grounding issue was to perform some kind of goal-related relevance analysis before grounding. This led to the idea of creating bug hypothesis rules, part of the logic formulation of the problem presented in this thesis. By implementing such restrictions, we were able to resolve the grounding problem.

### 6.3.5 Running the solver

After fixing the bug hypothesis by integrating the bug hypothesis rules, we were able to successfully get a plan (refer to Appendix D for example output). The results of the timings are as follows:

| Test Case | Forward Chaining | PDDL Translation | Find single plan |
|---|---|---|---|
| `info_leak` | 18s | 30s | 25s |
| `atk_shell` | 18s | 22s | 24s |

Figure 7: PDDL Test Results

We note that for finding a single plan, the PDDL solver generates a response faster than our original approach. However, due to the nature of planning algorithms, they are designed to exit once a single goal is reached. Therefore, in order to generate more attack plans and corresponding bug hypotheses, we have to run multiple instances of the planners with additional restrictions (so that the program does not give us the same plan).

We created the following algorithm (Refer to Appendix B) to do this, adapted from a similar algorithm from Sohrabi, S. et al. [26]. We note however, that because planners are not designed to keep previous work and look for more goals, having such a long run time for just one iteration makes this approach not the best fit. Based on the run times and analysis of the algorithm, we expect this approach to do about as well as our initial approach. Therefore, we discontinued work using this approach.

# 7 Future Work

## 7.1 Extending the System for more Attack Plans

As discussed in Section 4.3 and Section 6.1.1, we note that our system does not find all possible checklist items and attack plans. Therefore, further work is required to enable the system to generate all possible attack plans.

A possible approach would be to create additional constraints and rerun the system to generate other plans. This is similar to the algorithm we proposed in Section 6.3.5, where subsets of the checklist items are removed until we can no longer reach the goal. We note that the observation from Section 6.1.2 may hint at a possibility of optimizing this process by taking advantage of the implementation specifics of the datalog engine.

## 7.2 Extending Rules for Multi-System

In addition, we should experiment with adding rules to support multi-system attack plans and test our new system with the extended ruleset. Examples of multi-system rules can be seen from MulVal [5].

# 8 Conclusion

In this paper we presented a new logic-based formulation for the problem of generating attack plans and checklist items. Further to that, we designed and implemented a system for our new approach, and evaluated it against our old system. Our new system had a 2-3x speedup over the old system running with the minimum search depth for each test case. With this speedup, we envision that this approach will scale to more complex multi-system networks. At the same time, we note that the new system does not generate all possible attack plans and checklist items. This is a current limitation that we will have to address in future iterations of our system.

## References

[1] S. Jha, O. Sheyner, and J. Wing, "Two formal analyses of attack graphs," in *Computer Security Foundations Workshop, 2002. Proceedings. 15th IEEE*, pp. 49–63, IEEE, 2002.

[2] Z. Liu, S. Li, J. He, D. Xie, and Z. Deng, "Complex network security analysis based on attack graph model," in *Proc. Second Int Instrumentation, Measurement, Computer, Communication and Control (IMCCC) Conf*, pp. 183–186, Dec. 2012.

[3] S. Jilcott, "Securing the supply chain for commodity it devices by automated scenario generation," in *Proc. IEEE Int Technologies for Homeland Security (HST) Symp*, pp. 1–6, Apr. 2015.

[4] X. Ou and A. W. Appel, *A logic-programming approach to network security analysis*. Princeton University Princeton, 2005.

[5] X. Ou, S. Govindavajhala, and A. W. Appel, "Mulval: A logic-based network security analyzer.," in *USENIX security*, 2005.

[6] T. Fraser, "Vetting commodity it software and firmware (vet)," 2013.

[7] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, (New York, NY, USA), pp. 336–345, ACM, 2006.

[8] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "Pddl-the planning domain definition language," tech. rep., Yale Center for Computational Vision and Control, 1998.

[9] K. Ingols, R. Lippmann, and K. Piwowarski, "Practical attack graph generation for network defense," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pp. 121–130, Dec 2006.

[10] C. Phillips and L. P. Swiler, "A graph-based system for network-vulnerability analysis," in *Proceedings of the 1998 Workshop on New Security Paradigms*, NSPW '98, (New York, NY, USA), pp. 71–79, ACM, 1998.

[11] L. P. Swiler, C. Phillips, D. Ellis, and S. Chakerian, "Computer-attack graph generation tool," in *DARPA Information Survivability Conference &amp; Exposition II, 2001. DISCEX'01. Proceedings*, vol. 2, pp. 307–321, IEEE, 2001.

[12] "Model checking @cmu," 2002.

[13] R. W. Baldwin, "Rule based analysis of computer security.," tech. rep., MIT LCS Lab, 1988.

[14] D. Farmer and E. H. Spafford, "The cops security checker system," tech. rep., Purdue University, 1990.

[15] W. L. Fithen, S. V. Hernan, P. F. O'Rourke, and D. A. Shinberg, "Formal modeling of vulnerability," *Bell Labs technical journal*, vol. 8, no. 4, pp. 173–186, 2004.

[16] C. Ramakrishnan and R. Sekar, "Model-based analysis of configuration vulnerabilities," *Journal of Computer Security*, vol. 10, no. 1, 2, pp. 189–209, 2002.

[17] Y. Zhang, X. Fan, Y. Wang, and Z. Xue, "Attack grammar: A new approach to modeling and analyzing network attack sequences," in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pp. 215–224, IEEE, 2008.

[18] S. Noel and S. Jajodia, "Managing attack graph complexity through visual hierarchical aggregation," in *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pp. 109–118, ACM, 2004.

[19] F. Pfenning, "Natural deduction," 2004.

[20] E. Davis, "Horn clause logic," 2002.

[21] F. Pfenning, "Datalog," 2006.

[22] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, "Complexity and expressive power of logic programming," *ACM Computing Surveys (CSUR)*, vol. 33, no. 3, pp. 374–425, 2001.

[23] J. Vyskoil, D. Stanovsk, and J. Urban, "Automated proof compression by invention of new definitions," in *Logic for Programming, Artificial Intelligence, and Reasoning*, Springer, Jan. 2010.

[24] "Memsql," 2015.

[25] P. T. Dana Nau, Malik Ghallab, *Automated Planning.* Elsevier Science & Technology, 2004.

[26] S. Sohrabi, O. Udrea, and A. V. Riabov, "Hypothesis exploration for malware detection using planning," *Edited By: Nicola Policella and Nilufer Onder*, p. 29, 2013.

[27] H. Kautz and B. Selman, "Unifying sat-based and graph-based planning," in *IJCAI*, vol. 99, pp. 318–325, 1999.

[28] G. Wickler, "Planapp," 2010.

[29] S. Richter and M. Westphal, "The lama planner: Guiding cost-based anytime planning with landmarks," *Journal of Artificial Intelligence Research*, vol. 39, no. 1, pp. 127–177, 2010.

[30] M. Helmert, "The fast downward planning system.," *J. Artif. Intell. Res.(JAIR)*, vol. 26, pp. 191–246, 2006.

[31] "Google groups (fast downward): Understanding the output of fast downward," 2015.

[32] S. Russell, P. Norvig, and A. Intelligence, "A modern approach," *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, p. 27, 1995.

## A Old System Test: Min/Sat Timing Determination

We ran the original system with the following results:

- Test Case - Test case used
- MD - Max depth of tree search (PLANNING_DEPTH setting)
- MC - maximum number of checklist items used in chain (MAX_CHECKLIST setting)
- Time - Time taken to run system
- CI - Number of bug hypotheses (checklist items) identified to check from attack chains

| Test Case | MD | MC | Time | CI |
|---|---|---|---|---|
| `info_leak` | 3 | 1 | 31s | 0 |
| `info_leak` | 3 | 2 | 75s | 14 |
| `info_leak` | 5 | 2 | 74s | 14 |
| `info_leak` | 10 | 3 | 166s | 14 |
| `info_leak` | 10 | 5 | 579s | 14 |
| `info_leak` | 20 | 5 | 526s | 14 |
| `atk_shell` | 3 | 1 | 34s | 4 |
| `atk_shell` | 3 | 2 | 32s | 4 |
| `atk_shell` | 5 | 2 | 34s | 4 |
| `atk_shell` | 10 | 3 | 33s | 4 |
| `atk_shell` | 10 | 5 | 32s | 4 |
| `atk_shell` | 20 | 5 | 33s | 4 |

Figure 8: Results from running old system

We note that at a certain point, each test case goes into saturation (i.e. increasing the depth of the search does not make the search any longer). Our results are consistent with our expectation since we defined `info_leak` to have a deeper tree (saturates slower) and `atk_shell` to have a shallow one (saturates faster). Based on the results, the saturation point for `info_leak` is at about (MD:10, MC:5) and for `atk_shell` at (MD:3, MC:1).

Results: `info_leak` (Min. - MD:3, MC:2 / Sat. MD:10, MC:5) `atk_shell` (Min. MD:3, MC:1 / Sat. MD:3, MC:1)

## B PDDL Find-All Algorithm

```
####Initialization
1. Initialize ‘G‘ to be the set of goal states and ‘BS[g]‘ to
   be the bug hypothesis sets of a goal state ‘g‘. Let ‘BSS[g
   ]‘ be a dictionary of tuples of things needed to be checked
   .

#### ‘GENERATE_NEXT_PLAN‘
1. For each ‘g‘ in ‘G‘
2. Check if the number of checked values in the dict of ‘BSS[g]
    == size(BS[g]) +1‘, if so, then move to the next goal, or
   if there are no more goals in ‘G‘, then run the generic
   solver. If there are existing goals, pick the next ‘g‘ and
   well as the next ‘removed_bughyps‘ as indicated for ‘BSS[g]
    == None‘.
3. Run the planner with the new ‘g‘ added in as a goal state,
   and not ‘removed_bughyps‘ as a goal state.


####If planner generates Solution
1. Run Planner, if generates the solution, we take two things
   from it:
       - Bug hypothesis
       - Goal State
2. If the planner has a goal state, add this goal state ‘g‘ to
   ‘G‘
3. For bug hypothesis linked to the goal state, add the bug
   hypotheses to ‘BS[g]‘. For each bug hypothesis ‘b‘, we
   iterate through ‘BSS[g]‘ and add an entry for each ordered
   set of the previous bug hyps and the new ‘b‘. If the old
   one had no plan, then the new dict entry would have no plan
   .
4. Run ‘GENERATE_NEXT_PLAN‘
```

####If planner doesn’t generate Solution

1. If planner returns no solution, if was a generic solve, then end the algorithm, if not, it was linked to a goal ‘g‘ and set of removed bug hypotheses ‘removed_bughyps‘. In that case seg ‘BSS[g][removed_bughyps] = False‘. And repeat process ‘GENERATE_NEXT_PLAN‘.

## C VET E3 Day 2 Network Diagram

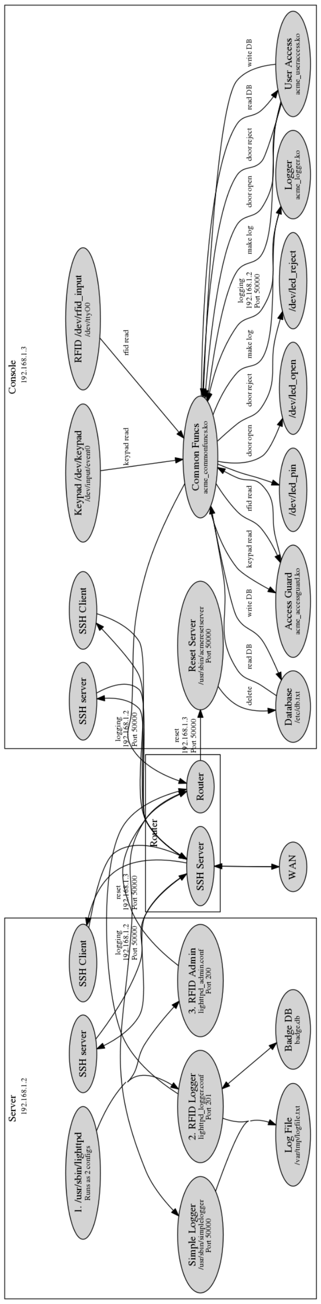

Figure 9: Network Diagram for VET Engagement 3 Day 2

## D PDDL Sample Input/Output

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This is the domain file for VET-TA1 Planning
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (domain vuln)
  (:requirements :strips :typing)
  ; PDDL Types

(:types
  hInteger ; hInteger
  hID ; hID
  hSystemID ; hSystemID
  hString ; hString
  hBoolean ; hBoolean
)

; PDDL Constants

(:constants
     hString-/etc/apache2/apache2.conf
     hString-ldap
     hString-auth.backend.htdigest.userfile ...
  )

  ; PDDL Predicates

(:predicates
  (binary ?id - hid)
  (availabilitycriticalfile ?id - hid)
  (effectivetaintedposts ?id - hid ?target_id - hid ?value - hstring)
  ...
  (MemoryCorruptable ?id - hID)
  (NetworkExposesObtainedContent ?id - hID ?file - hID)
  ...
  (Exfiltrates ?res - hString)
  (ExfiltratesX )
  )


  ; Metric value to optimize
  (:functions
    (total-cost)
  )

  ; PDDL Actions

  (:action Execution
   :parameters (?X - hID)
    :precondition (
      and (Executable  ?X)
    )
    :effect (
      and (ExecutesCode  ?X)
    )
  )


  (:action BinaryExecution
   :parameters (?X - hID)
    :precondition (
      and (Binary  ?X)
    )
    :effect (
      and (ExecutesCode  ?X)
    )
  )

...

  (:action ExfiltratesSolver
   :parameters (?EXFILTRATED_FILENAME - hString
   ?SERVER - hID
   ?EXFILTRATED_FILE - hID
   ?PORT_NUM - hInteger)
    :precondition (
      and (NetworkExposesContent  ?SERVER ?PORT_NUM ?EXFILTRATED_FILE)
            (FullyConfidentialFile  ?EXFILTRATED_FILE)
            (Filename  ?EXFILTRATED_FILE ?EXFILTRATED_FILENAME)
    )
    :effect (
      and (Exfiltrates  ?EXFILTRATED_FILENAME)
    )
  )

...

  (:action ExfiltratesSolverResolve
   :parameters (?EXFILTRATED_FILENAME - hString)
    :precondition (
      and (Exfiltrates  ?EXFILTRATED_FILENAME)
    )
    :effect (
      and (ExfiltratesX )
    )
  )

...

  (:action bughyp_MemoryCorruptable
   :parameters (?id - hID)
    :precondition (
      and (Binary ?id)
    )
    :effect (
      and (MemoryCorruptable ?id)
      (increase (total-cost) 1)
    )
  )
```

Figure 10: Sample VET-TA1 PDDL Domain

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; This is the problem file for VET-TA1 Planning
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (problem ta1-prob)
  (:domain vuln)
  ; PDDL Objects

  (:objects
     hBoolean---false hBoolean---true - hBoolean
     hString-/usr/share/terminfo/a/avt-w
     hString-/lib/systemd/systemd-fsck
     hString-/usr/share/terminfo/a/avt-s
     ...
  )

  ; PDDL Init

  (:init
     (Filename  hID-1 hString-/usr/share/vim/vim73/syntax/tcl.vim)
     (Basename  hID-1 hString-tcl.vim)
     (SystemMember  hID-1 hSystemID-1)
     (DirectoryMember  hID-1 hString-/ hSystemID-1)
     ...
  )

  ; PDDL Goal
  (:goal
    (and
      (ExfiltratesX )
    )
  )

  ; PDDL Metric minimization for minimizing the bug hyps used
  (:metric minimize (total-cost))
)
```

Figure 11: Sample VET-TA1 PDDL Problem

```
bughyp_commandinjectable hid-1108 hid-1108 hid-1108 (1)
commandinjection hid-1108 hid-1108 hid-1108 (0)
bughyp_iptablesbackdoor hinteger-50000 hsystemid-1 hid-1108
   hstring-/usr/sbin/acmeresetserver hstring-sock_stream (1)
brokeniptables hsystemid-1 hinteger-50000 (0)
shellread hid-1108 hinteger-50000 hstring-/usr/sbin/
   acmeresetserver hstring-/etc/db.txt hid-10004 hsystemid-1
   hstring-sock_stream hid-1108 (0)
exfiltratessolver hstring-/etc/db.txt hid-1108 hid-10004
   hinteger-50000 (0)
exfiltratessolverresolve hstring-/etc/db.txt (0)
```

Figure 12: Sample PDDL plan generated from test case `info_leak`