AGBuilder: An AI Tool for Automated Attack Graph Building, Analysis, and Refinement

Bruhadeshwar Bezawada, Indrajit Ray, Kushagra Tiwary

AGBuilder: An AI Tool for Automated Attack Graph Building, Analysis, and Refinement

Bruhadeshwar Bezawada[1], Indrajit Ray[2],[3], and Kushagra Tiwary[2]

[1] Mahindra École Centrale, Hyderabad, India
[2] Colorado State University, Fort Collins CO 80523, USA
[3] National Science Foundation, USA
bru@mechyd.ac.in
{indrajit.ray,kushagra.tiwary}@colostate.edu

Abstract. Attack graphs are widely used for modeling attack scenarios that exploit vulnerabilities in computer systems and networked infrastructures. Essentially, an attack graph illustrates a *what-if* analysis, thereby, helping the network administrator to plan for potential security threats. However, current attack graph representations not only suffer from scaling issues, but also are difficult to generate. Despite efforts from the research community there are no automated tools for generating attack graphs from textual descriptions of vulnerabilities such as those from the Common Vulnerabilities and Exposures (CVE) in the National Vulnerability Database (NVD). Additionally, there is little support for incremental updates and refinements to an attack graph model. This is needed to reflect changes to an attack graph that arise because of changes to the vulnerability state of the underlying system being modeled. In this work, we present an artificial intelligence (AI) based planning tool, *AGBuilder* –Attack Graph Builder, for automatically generating, updating and refining attack graphs. A key contribution of AGBuilder is that it uses textual descriptions of vulnerabilities to automatically generate attack graphs. Another significant contribution is that, using AGBuilder, we describe a methodology to incrementally update attack graphs when the system changes. This aspect has not been addressed in prior research and is a crucial step for achieving resiliency in the face of evolving adversarial strategies. Finally, AGBuilder has the ability to reuse smaller attack graphs, e.g., when building a network of networks, and join them together to create larger attack graphs.

Keywords: Attack graphs, Planning Domain Definition Language (PDDL), AI Planning, CVE, NVD

1 Introduction

Cyber-attacks against safety critical and mission critical systems such as nuclear power plants are rising alarmingly. It is no longer a question of "if" but "when" a system will be attacked. Thus, in order to be adequately prepared for such an eventuality, there is a need to better understand how the system can be attacked so that provisions for defense deployment can be made or, perhaps, provisions

for the graceful degradation of mission services can be instantiated when all defenses have failed. Information security planning and management traditionally begins with risk assessment with the help of system mapping and dependency analysis. The outcome of this process is an identification of vulnerabilities in the system, an enumeration of the threats to critical resources arising from these vulnerabilities, and the corresponding loss expectancy. The analysis allows one to determine appropriate security controls to protect resources and minimize their susceptibility to cyber attacks.

Attack trees [17, 10, 2] and attack graphs [1, 20, 13, 16, 22, 28] are two systematic computer security models that represent a networked system's vulnerability to malicious attacks by enumerating known vulnerabilities in the hosts or applications. They capture cause-consequence relationships between system configuration and the vulnerabilities in the form of And-Or tree (attack tree) or a directed graph (attack graph). Nodes in the tree/graph represent system states that may be of interest to the attacker. Edges connecting the nodes denote a cause-consequence relationship among the states. A key weakness of this representation is the explosion of state space. This becomes a critical drawback for analyzing large cyber-physical systems with many resources that need to be protected from a multitude of attacks.

Automated planning holds promise to reduce the number of nodes in the attack graph/tree and produce a scalable solution. Boddy et al. (2005) [3], presented Behavioral Adversary Modeling System (BAMS), a planning system that models attack scenarios and produces countermeasures to subvert the attacks in networks of large organizations. Ghosh and Ghosh [7] proposed a planner based approach for tractable representation of attack graphs and automatic generation of attack paths. However, none of these works discuss how an attack graph can be automatically constructed, refined and updated as needed from textual description of vulnerabilities such as those in the National Vulnerability Database or CVE repositories.

In this work, we model the attack graph generation and analysis problem as a *planning problem* [7] in the artificial intelligence community. When compared with logic programming approaches like [13], the planning approach lends itself to incremental updates and aggregation, which are quite useful to network administrators. We encode the attack graph in the Planner Domain Definition Language (PDDL) representation, referred to as a PDDL *domain*. However, there are some important challenges that arise in this modeling. First challenge is that, translating the attack graph/tree of a large network to the corresponding PDDL domain is an iterative process, which demands a lot of time and effort from engineers. The issue is further exacerbated by the lack of tool support to build, debug and maintain PDDL domains from textual CVE descriptions. The second challenge is that, if the underlying system is changed in any manner, for example, by installing a new application, then updating the corresponding attack graph is a computationally complex and error-prone process. The existing approaches have not addressed this situation and require generating a fresh attack graph for the changed system environment. The third challenge is that,

when a PDDL domain is incrementally built more actions are added into a domain or actions already in the domain are edited. Such incremental development is found in scenarios where network administrators start by analyzing smaller parts of a network and then try to aggregate the smaller network models into a larger network model.

To address these challenges, we present a formal methodology and a corresponding tool-set, *AGBuilder* –Attack Graph Builder, designed to automatically generate PDDL based representation of attacks from textual description of vulnerabilities found in the CVE system or the NVD system. Our tool-set incorporates a natural-language processing based generator to generate a PDDL based model of attacks from vulnerabilities and support for incremental development of the PDDL model to reconcile incremental versions of PDDL domains by generating explanations for changes in plans that result from running the modified domain against a planner. Additionally, the tool-set constructs abstract syntax trees and attack graphs for the PDDL domain representation, which facilitates visualization of the domain and the planning problems.

The rest of the paper is organized as follows. In Section 2, we give an overview of the PDDL language and explain the modeling of attack graphs using PDDL. In Section 3, we give an overview of our approach. In Section 4, we describe the AGBuilder tool set and our methodology in detail. In Section 5, we describe the related work in this domain and conclude in Section 6.

2 Attack Graph Modeling using PDDL

In [7], the authors provided an approach for modeling the attack graphs using AI planners. In this section, we describe this process in detail. A PDDL definition is composed of two key parts: (1) a PDDL domain definition and (2) a PDDL problem description.

PDDL Domain: A PDDL domain is a high-level description of a set of problems and the corresponding actions and constraints involved. A PDDL domain specifies the requirements it supports, the available actions, the pre-conditions and post-conditions of actions. The pre-conditions and post-conditions are expressed as first-order logic predicates. The requirements of the PDDL domain specify which features it expects the planner to support. A planner will only accept a domain if it supports all the requirements mentioned in the domain. A single PDDL domain can be used to represent multiple attacks from vulnerability databases. The PDDL domain stores pre-conditions, post-conditions and cause-effect relationships in courses of actions that represent attacks. Consider the following PDDL domain that models an attack:

(*define* (domain PAG) (:requirements :equality :disjunctive-preconditions)
(:*functions* (version ?software))
(:*predicates* (user ?User) (email-msg ?User ?Msg) ... (browser-ssl-compromised ?Browser) (certificate-authorized ?Certificate))
(:*action* attacker-sends-email-with-keylogger :*parameters* (?User ?File ?Keylogger) :*precondition* (and (user ?User) (file ?File) (has-trojan ?File) (keylogger-trojan ?File ?Keylogger)) :*effect* (and (email-msg ?User bad-email) (mail-attachment bad-email ?File))))
(:*action* user-visits-site :*parameters* (?User ?Browser ?Site) :*precondition* (and (user ?User) (software ?Browser) (browser ?Browser) (site ?Site)) :*effect* (and (use-software ?Browser) (user-visits-site ?User ?Site)))
(:*action* user-starts-email :parameters (?User ?Mailer) :*precondition* (and (user ?User) (mailer ?Mailer)) :*effect* (and (use-software ?Mailer) (running ?Mailer)))
(:*action* user-reads-email :*parameters* (?User ?Mailer ?Msg) :*precondition* (and (user ?User) (mailer ?Mailer) (use-software ?Mailer) (email-msg ?User ?Msg)) :*effect* (and (msg-opened ?Msg))
(:*action* user-presses-F1-at-vbscript-site :*parameters* (?User ?Browser ?Site) :*precondition* (and (user ?User) (use-software ?Browser) (browser ?Browser) (= ?Browser browser-IE) (= ?Site vbscript-link) (user-visits-site ?User vbscript-link)) :*effect* (user-types F1))
(:*action* user-opens-attachment :*parameters* (?User ?Msg ?File ?Mailer) :*precondition* (and (user ?User) (use-software ?Mailer) (mailer ?Mailer) (msg-opened ?Msg) (mail-attachment ?Msg ?File) (file ?File)) :*effect* (opened ?File))
(:*action* key-logger-installed :*parameters* (?User ?File ?KeyLogger) :*precondition* (and (user ?User) (opened ?File) (file ?File) (has-trojan ?File) (keylogger-trojan ?File ?KeyLogger)) :*effect* (and (key-logger ?KeyLogger) (installed ?KeyLogger)))
::
::
(:*action* user-login-with-keylogger-activated :*parameters* (?User ?Account ?Keylogger) :*precondition* (and (user ?User) (account ?Account) (key-logger ?Keylogger) (running ?Keylogger)) :*effect* (and (logged-in ?User ?Account) (information-available ?User ?Account) (records ?Keylogger ?Account))))

The *predicates* (`user ?User`) and (`file ?file`) among others describe the state of the world. They can either be true or false. `?User` and `?file` are formal parameters to the predicates, user and file. They describe who the user is and which file is being used.

Actions allow a planner to move from one state to another. An action can have pre-conditions and post-conditions, both of which are expressed as predicates. An action can only occur in the current state if the current state supports its preconditions, i.e., the predicates already in the current state do not negate the predicates in the action's preconditions. The effects of an action result in the next state.

To elaborate, the shown PDDL domain Personalized Attack Graph (PAG), contains ten actions. Each action is defined with pre-conditions and post-conditions (effects). For example, action `user-reads-email` with parameters (`?User`, `?Mailer` and `?Msg`) has preconditions(`user ?User`), (`mailer ?Mailer`), (`use-software ?Mailer`) and (`email-msg ?User ?Msg`)). These preconditions verify if the `mailer` supplied as an argument actually exists, if the user is indeed the user supplied in the argument and whether the email message has the user supplied in the argument and the message supplied in the argument. In other words, the pre-conditions test to see if those predicates are true in the current state. If the pre-conditions are true in the current state, then the post-conditions or effects are applied. The effects are (`use-software ?Mailer`) and (`running ?Mailer`). When the effects are applied, the predicates (`use-software ?Mailer`) and (`running ?Mailer`) are set to true. Each action can be a unit step towards exploiting a vulnerability.

PDDL Problem: A PDDL problem is a concrete instance of a specific PDDL domain where the general variables in a PDDL domain are replaced with concrete values and a sub-set of actions defined in the PDDL domain. A PDDL problem contains an initial state: a set of predicates that are set to true initially, and a goal state: a set of predicates that may or may not be true with the actions defined in the domain. A PDDL problem can be used to perform what-if analysis by simulating conditions of the system and testing various attack paths originating from the current simulated state of the system. Consider the following PDDL problem that corresponds to the domain "PAG".

(*define* (problem PAG-problem1) (:*domain* PAG) (:*objects* user1 ... browser-seamonkey browser-mozilla)
(:`init` (user user1) (mailer gmail) (exploit-vulnerability vulnerability-key-logger) (file file-with-trojan) (has-trojan file-with-trojan) (key-logger key-logger1) (key-logger-trojan file-with-trojan key-logger1) (site vbscript-link) (has-crafted-dialog-box vbscript-link) (vb-script-version VB-5-1) (software browser-IE) (browser browser-IE) (software browser-firefox) (= (version browser-firefox) 2) (browser browser-firefox) (information-available user1 account-bank) (account account-bank))
(:`goal` (and (information-leakage account-bank))))

In the above problem, the initial state consists of predicates, (`user user1`) and (`mailer gmail`) etc. The initial state describes what is true about the system when the planner starts out. The goal state is defined as (`information-leakage account-bank`). This indicates that when (`information-leakage account-bank`) becomes true, the goal state is said to have been reached.

PDDL Planner: A PDDL planner tries to solve a PDDL problem by finding a plan that satisfies it. A successful plan is a sequence of actions from those specified in the PDDL problem for a given initial state.

PDDL Plan: A plan is a sequence of actions from the initial state in the PDDL problem to the final state in the PDDL problem. A plan can also be thought of as a sequence of transitions from the initial state to the goal state.

Here is the plan produced for the domain "PAG" and problem "PAG-problem1" by using the Metric-FF planner [8]:

```
0: ATTACKER-SENDS-EMAIL-WITH-KEYLOGGER USER1 FILE-WITH-TROJAN KEY-LOGGER1
1: USER-VISITS-SITE USER1 BROWSER-IE VBSCRIPT-LINK
2: USER-STARTS-EMAIL USER1 GMAIL
3: USER-READS-EMAIL USER1 GMAIL BAD-EMAIL
4: USER-OPENS-ATTACHMENT USER1 BAD-EMAIL FILE-WITH-TROJAN GMAIL
5: KEY-LOGGER-INSTALLED USER1 FILE-WITH-TROJAN KEY-LOGGER1
6: USER-PRESSES-F1-AT-VBSCRIPT-SITE USER1 BROWSER-IE VBSCRIPT-LINK
7: KEY-LOGGER-ACTIVATED KEY-LOGGER1 BROWSER-IE
8: USER-LOGIN-WITH-KEYLOGGER-ACTIVATED USER1 ACCOUNT-BANK KEY-LOGGER1
9: ATTACKER-INTERCEPTS KEY-LOGGER1 ACCOUNT-BANK
```

The sequence of actions is "ATTACKER-SENDS-EMAIL-WITH-KEYLOGGER USER1 FILE-WITH-TROJAN KEY-LOGGER1" followed by "USER-VISITS-SITE USER1 BROWSER-IE VBSCRIPT-LINK" and so on until the last step, "ATTACKER-INTERCEPTS KEY-LOGGER1 ACCOUNT-BANK"

In essence, these steps describe that an attacker sends an email with a keylogger to a user. Once the user opens his/her browser and goes to their Gmail account, reads their email, and opens the attachment from the email sent by the attacker, the keylogger is installed on their system. When the user presses `F1` on a website with VB Script, the keylogger is activated. This allows the attacker to remotely track all of the user's keystrokes and allows the attacker to intercept the user's bank account credentials when the user visits their bank's website and tries to sign in. The PDDL domain above only has one vulnerability, but in a more realistic scenario, we can expect thousands of such vulnerabilities in a domain and at least one plan for every vulnerability, which allows us to derive attack paths.

Planning Graph: A planning graph is a layered directed graph, consisting of alternating layers of states and actions. The state layers contain predicates that are for that state. The action layers consist of actions that map pre-conditions and post-conditions. An edge from a predicate to an action indicates that the predicate is the pre-condition of that action. An edge from an action to a predicate implies that the predicate is an effect or post-condition of that action. The planning graph, therefore, represents the transitions from the initial state to the goal state by using the actions and predicates defined in the PDDL domain. A planning graph is, simply put, a more granular way to look at a PDDL plan and can help us visualize the attack path.

3 Our Approach for Automated Attack Graph Generation and Refinement

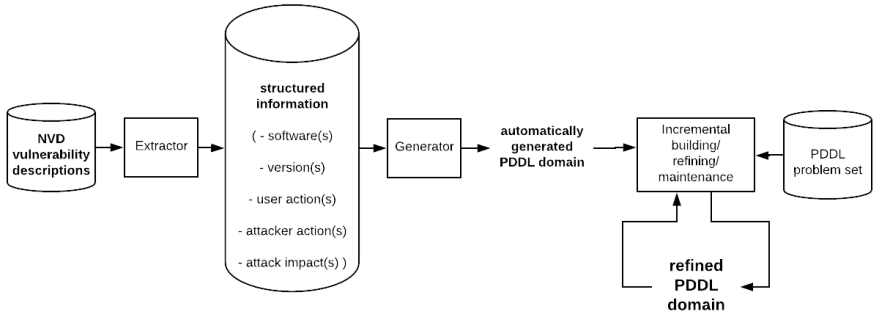

Fig. 1. Overview of the process of generating and maintaining attack graphs

The AGBuilder tool set's workflow shown in Figure 1 has the following key steps each of which has been developed as an independent software module:

- *Step 1.* The Extractor is used to extract structured information from the vulnerability descriptions. Such information is typically obtained from vulnerability databases such as NVD (https://nvd.nist.gov) or ICS-CERT (https://ics-cert.us-cert.gov).
- The structured information is used to generate a PDDL domain by the Generator module. The PDDL domain represents cause-effect relationships in the attack.
- *Step 2.* The PDDL domain is tested using different PDDL problems, which are extracted from the information in event logs of the system. [4] Each PDDL problem is meant to test at least one attack/vulnerability. These PDDL problems are stored in a database internally maintained by the AGBuilder tool.
- *Step 3.* An AI planner is used to generate one PDDL plan for each PDDL problem along with the PDDL domain. The set of actions in the PDDL planner describe an attack path in the attack graph.
- *Step 4.* Whenever the domain is updated/modified, the tool ensures that the latest version of the PDDL domain is consistent with respect to the last known stable version. The tool checks for the consistency of PDDL plans against the PDDL domain and PDDL problems, and provides feedback on plans which failed, thus helping the developer maintain the cause-effect relationships.

[4] An event log is record of actions taken by users and may represent the exploitation of a vulnerability.

3.1 Automatically Generating PDDL Domain from Natural Language Textual Descriptions

The automatic generation phase takes vulnerability descriptions (CVE or NVD) from a vulnerability database as input and renders an automatically generated PDDL domain as output. Our tool extends the NLP-based (natural language processing) software previously developed by us as part of a prior project [29] to extract structured information from unstructured text. The NLP algorithm implementation is based on parts-of-speech tagging model wherein lexical patterns are identified from the text and the relationships of the subjects and rules are identified from these patterns. The Stanford coreNLP POS Tagger [23] was used for tagging. A corpus of 30 descriptions were used for the manual rule generation. Based on these rules, the parts-of-speech tagging algorithm works by tagging labels of a word in the text based on its role in the sentence, like verb, noun, adjective etc, and the context of the word's usage. For instance, software names are typically tagged as proper nouns. Another rule is that software names are followed by a preposition or subordinating conjunction. A pre-stored gazette of 48709 entries, consisting of software and operating systems, is used to match a proper noun to a software name. File names can be matched using regular expressions containing period "." and so on. To identify attacker and user actions, we partition the description and based on the relative positioning of the subject, verbs and modifiers, e.g., like "through", to extract the subjects [11] and the respective actions. As a subject can be either attacker or user, sentiment analysis [6] is used to label the subject as positive or negative sentiment by considering the sentiment labels of the respective verbs and modifiers, and finally, labeling the respective subject as attacker or user. The information extracted comprises of the following: software name(s), software version(s), user action(s), attacker action(s) and attacker impact(s) Here is an example of a vulnerability description from NVD:

> CVE-2010-0483: vbscript.dll in VBScript 5.1, 5.6, 5.7, and 5.8 in Microsoft Windows 2000 SP4, XP SP2 and SP3, and Server 2003 SP2, when Internet Explorer is used, allows user-assisted remote attackers to execute arbitrary code by referencing a (1) local pathname, (2) UNC share pathname, or (3) WebDAV server with a crafted .hlp file in the fourth argument (aka helpfile argument) to the MsgBox function, leading to code execution involving winhlp32.exe when the F1 key is pressed, aka "VBScript Help Keypress Vulnerability."

For the above vulnerability description, our extractor gives the following output:

Software : VBScript
Versions : [51, 56, 57, 5.8, 2000, 2003, SP2, SP2, winhlp32.exe]
Modifiers : [and 5.8]
User Action: Internet Explorer is used
User Action: the F1 key is pressed
Attack Vector: referencing a -LRB- 1 -RRB- local pathname , -LRB- 2 -RRB- UNC share pathname , or -LRB- 3 -RRB- WebDAV server with a crafted hlp file in the fourth argument -LRB- aka helpfile argument -RRB- to the MsgBox function , leading to code execution involving winhlp32exe
Attack Impact: execute arbitrary code

The structured information extracted from vulnerability descriptions is then used to automatically generate the corresponding PDDL domain. Figure 2 is

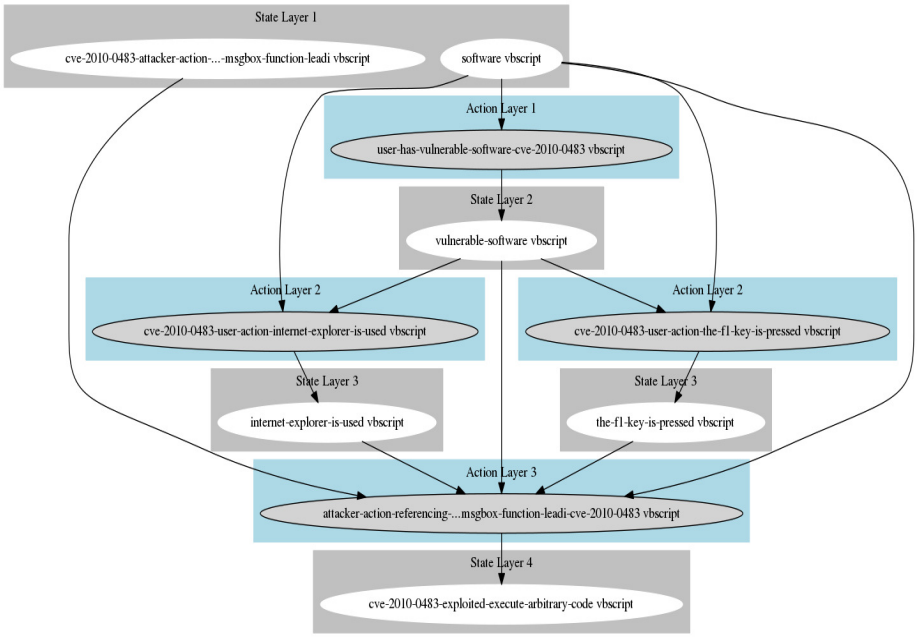

Fig. 2. Attack Path generated for NVD 2010-0483

an example of an attack graph that is automatically generated by the Generator module of our AGBuilder toolset using the above domain and the relevant problem file, and illustrates the attack path taken to exploit the vulnerability.

3.2 Incremental Building and Refinement of the Attack Graph

AGBuilder assumes that the PDDL domain, PDDL problem and the plan that represent the cyber threat situational awareness model are syntactically correct

and valid. A system administrator focuses only on the undesired plans, i.e., plans that have changed since the last update of the domain. Figure 3 illustrates the process flow for the domain reconciliation algorithm.

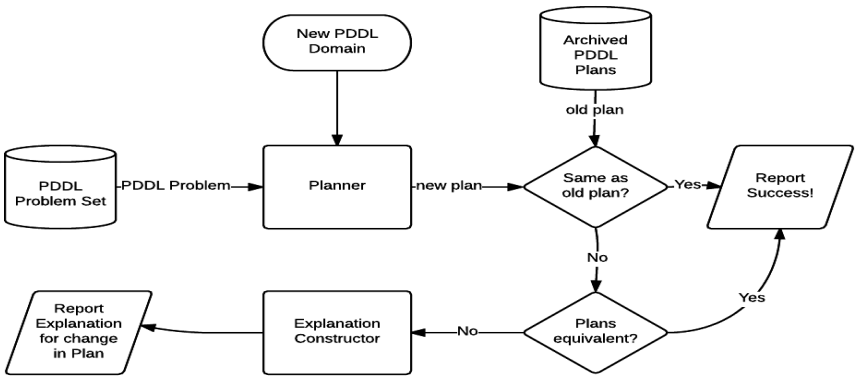

Fig. 3. Overview of incremental support for maintaining/refining attack domains

Given that the system administrator has the last known stable version of the PDDL domain, the PDDL problem set, archived plans from the last known stable version, and the new PDDL domain with bugs in it, a PDDL planner is run on each problem from the PDDL problem set and the new PDDL domain to generate one plan per problem. The tool then matches each pair of a newly generated plan and its corresponding archived plan. For each pair of plans that do not match, the tool constructs an explanation for the observed difference in plans. Sometimes plans may have a different sub-sequence of actions such as $A \rightarrow B \rightarrow C \rightarrow D$ as opposed to $A \rightarrow C \rightarrow B \rightarrow D$. If actions B and C can be executed in parallel, the tool will consider the two plans to be equivalent.

As a motivating example, for incrementally refining the model, let us assume that a planner runs on a stable PDDL domain and a PDDL problem and creates a plan ($user - opens - email - client \rightarrow user - opens - email \rightarrow user - downloads - keylogger - in - attachment \rightarrow user - enters - username - and - password - on - site1 \rightarrow user - enters - username - and - password - on - site2$) where the actions in the plan are defined in the domain. If some modification to the domain causes the plan to be generated as ($user - opens - email \rightarrow user - opens - email - client \rightarrow user - downloads - keylogger - in - attachment \rightarrow user - enters - username - and - password - on - site1 \rightarrow user - enters - username - and - password - on - site2$), this new plan may be considered an undesirable plan by the developer since the user now opens the email before even opening the email client. Now, to debug the domain definition and fix the plan, the developer needs to track the state of the system starting with the initial state defined in the PDDL problem and trace all the actions in the plan and

their effects on the state of the system, to figure out what caused the observed difference in the plans.

Further, it is possible that a new plan ($user - opens - email - client \rightarrow user - opens - email \rightarrow user - downloads - keylogger - in - attachment \rightarrow user - enters - username - and - password - on - site2 \rightarrow user - enters - username - and - password - on - site1$) is observably different, but it might still be equivalent to the old plan ($user - opens - email - client \rightarrow user - opens - email \rightarrow user - downloads - keylogger - in - attachment \rightarrow user - enters - username - and - password - on - site1 \rightarrow user - enters - username - and - password - on - site2$. This is possible if $user - enters - username - and - password - on - site1$ and $user - enters - username - and - password - on - site2$, are independent actions, i.e., actions that do not affect each other in anyway and can be executed in parallel. If the goal of the PDDL plan is for the user's credentials from site1 and site2 to be compromised in no particular order, it should not make a difference if the user's credentials for site1 are compromised before his/her credentials for site2 and vice-versa. If the actions $user - enters - username - and - password - on - site1$ and $user - enters - username - and - password - on - site2$ do not have any preconditions or effects in common, then they can be considered independent. Therefore, even though the two plans are observably different, they are equivalent. This would imply that neither of the two is an undesired plan. The developer would still need to manually look at the code to make this deduction.

With incremental development, some inadvertent changes to the old actions in the PDDL domain can also lead to unexpected outcomes in terms of the plans produced. At some point, the interaction of the actions in the PDDL domain can get very challenging to visualize for the developer. If multiple developers collaborate on a PDDL domain, and some unintended changes are made to the domain (for instance, adding a new pre-condition to an action or deleting an action), this change in the domain can cause a different plan to be generated by the planner. It can get very hard to manually inspect this change in plan and deduce what caused the change.

Our tool assists the PDDL domain developer ensure that subsequent versions of the PDDL domain are consistent by constructing explanations for the observed changes in the old and new plan by creating two planning graphs: (1) a planning graph using the last stable version of the domain, the PDDL problem and the last stable version of the plan, and (2) planning graph using the new domain, the PDDL problem and the new plan. It then traverses the two planning graphs layer by layer and determines what caused the change in the plan. It infers whether the cause for the change is the addition/removal/replacement of a predicate in the set of pre-conditions or post-conditions, or the addition/removal/replacement of an action. This is then reported as an explanation for the observed change in plans. Next, we describe the details of the modules in AGBuilder and their functionality.

4 AGBuilder Modules

We categorize AGBuilder into its *knowledge base components* and *processing modules*.

4.1 Knowledge Base Components

Input data: Input to AGBuilder includes the presumably faulty domain which needs to be fixed, the most recent known stable version of the domain, the set of PDDL problems pertaining to the faulty domain, and the archived PDDL plans.

Most recent stable domain: This is the last known stable version of the domain that produces all the plans as expected by the user.

Faulty domain: A domain file is considered faulty because of changes made to the most recently stable version of the domain. The most recently stable version of the domain produces all the plans as expected but the faulty domain produces unexpected/undesired plans. Unexpected/undesired plans are plans with unexpected sequences of actions in them, or sequences of actions that are different from those produced in plans from the last known stable version of the domain.

PDDL problem set: The PDDL domain is accompanied by a set of problems. The PDDL problems are used for generating plans. The tool uses the planner, the faulty domain, and PDDL problem set to generated plans for the faulty domain.

Archived PDDL plans: The archived PDDL plans can be thought of as the reference output. There is one archived plan per problem in the problem set. The archived PDDL plans were generated using the most recently stable version of the PDDL domain. If the plans generated using a PDDL domain and the problem set match those in the archived set, then the plans are treated as correct and the PDDL domain used to generate those plans is considered "stable" and not "faulty".

4.2 AGBuilder Processing Modules

AGBuilder has a parsing module, a planning graph construction module, and a module for generating explanations.

Parsing module: The parsing module has a built-in lexer and parser that reads a PDDL domain or problem into an abstract syntax tree (AST). The AST structure is useful as it allows the tool to run queries on it to find actions that contain a predicate for determining if two or more actions are independent of each other. AGBuilder uses a parser and a lexer. The parser and lexer were generated using a BNF representation of PDDL 3.1 and ANTLR [14], a popular parser generator for Java.

Planning graph construction module: We use the Metric-FF the PDDL Planner to generate plans [8]. However, any other new planner can also be used, which gives the tool a distinct advantage of keeping up to date with minimal

effort. The planning graph construction module takes as input, a PDDL domain, a PDDL problem and a plan generated using the PDDL domain and the PDDL problem. It converts the domain and the problem into ASTs and then uses the ASTs and the plan to create a planning graph.

1. Initialize the first layer as the set of predicates in the initial conditions from the PDDL Problem's AST.
2. For every action a_i in the sequence of actions in the plan:
   (a) Initialize the next layer as an action layer (if it is not initialized already), and set it as the current layer.
   (b) If the current action layer is not the first action layer and the current action a_i is "independent" with respect to every single action in the previous action layer:
      i. Add current action a_i to previous action layer and set previous action layer as current layer.
   (c) Initialize the next layer as state layer (if it is not initialized already) and set it as the current layer.
   (d) For every precondition of action a_i in the previous state layer, set a directed edge from the pre-condition in the previous state layer to the action a_i in the previous action layer.
   (e) Apply the postconditions of action a_i to the current state layer.
   (f) For every postcondition of the action a_i in the current state layer, set a directed edge from the action a_i in the previous action layer to the postcondition in the current state layer.

This module determines whether two actions are independent. The criteria for two actions to be independent is that the negative effects (negated predicates) of either action should not have any intersection with the preconditions or the positive effects (predicates that aren't negated) of the other action. This is formally expressed in the following relationship as: actions a1 and a2 are independent if and only if:

$effects^-$ (a1) $\cap$ ($preconditions$(a2) $\cup$ $effects^+$(a2)) = $\{\phi\}$

and

$effects^-$ (a2) $\cap$ ($preconditions$(a1) $\cup$ $effects^+$(a1)) = $\{\phi\}$

The relationship implies that two independent actions don't interact with each other in any way, so they can be executed simultaneously, or can be in the same layer of a planning graph. For a set of actions to be independent, every possible pair of actions in that set needs to satisfy the aforementioned relationship for independence. Thus, for a plan $A \rightarrow B \rightarrow C \rightarrow D$ where B and C are independent according to the aforementioned criteria, the planning graph would have A in the first action layer, B and C in the second action layer and D in the third action layer. Another plan $A \rightarrow C \rightarrow B \rightarrow D$ would also have A in the first action layer, B and C in the second action layer, and D in the third action layer. This is how it is determined that two plans with differing sub-sequences of actions are equivalent.

Module for constructing explanations: The module for constructing explanations is the main module of AGBuilder. It takes one problem at a time from the PDDL problem set, generates two planning graphs, one for the last known stable version of the domain and the archive plan, and another for the faulty domain and the new plan. It then compares these two planning graphs to deduce if two plans are actually different or if they are equivalent but have different sub-sequences of actions. If the plans are different, it uses the planning graphs to deduce an explanation for causes of the change in the new plan. The main algorithm for generating explanations in AGBuilder is as follows: For each PDDL problem x, in the problem set $\{X\}$:

1. Run planner on PDDL problem and the new (faulty) domain to generate a plan.
2. If the plan produced has a different sequence of actions than the archived plan:
   (a) Run the faulty domain and the stable version of the domain through the parsing module to get two ASTs.
   (b) Run the PDDL problem x_i through the parsing module to get an AST for the problem.
   (c) Construct one planning graph using the stable domain AST, problem x_i AST and the archived plan of the table domain on the problem x_i.
   (d) Traverse both the graphs simultaneously, one layer at a time, and find differences in actions or states at each layer. If a difference is observed in the corresponding layers of the two planning graphs, report the differences observed and halt any further traversal of planning graphs. The process of constructing explanations is discussed in more detail next.

Now, the explanation construction module takes as input the planning graph from the most recent stable domain, the planning graph from the new (presumably faulty) domain, the old domain AST and the new domain AST. The algorithm for generating explanations is as follows:

1. Skip the first state layer in the two planning graphs. This is because the first state layer has the initial preconditions from the PDDL problem (which both planning graphs share), therefore they must be the same.
2. Compare the two sets of actions after the first state layer of the two planning graphs.
3. If the sets of actions in the current action layer of the old graph and the new graph are different:
   (a) Print the actions in the current layer of the new graph which are not in the corresponding layer of the old graph. These actions are extra actions in the new plan.
   (b) Print the actions in the current layer of the old graph which are not in the corresponding layer of the new graph. These actions are missing actions in the new plan.
4. If the sets of predicates in the current state layer are different:

(a) Print the predicates in the new graph which are a negation of the predicates in the old graph.
(b) For each predicate P_i in the new graph that is $\neg P_i$ in the old graph:
    i. Print the action(s) from the previous action layer which has this predicate $\neg P_i$ in its post conditions. The action(s) that had this predicate as its post condition is what contributed to the change in the observed plan.
    ii. Print the action(s) from the previous action layer of the old graph that has the predicate P_i in its post conditions for reference.
(c) Print the predicates in the current layer of the old graph which are not in the corresponding layer of the new graph. For each of these predicates:
    i. Print the action(s) from the previous action layer in the old planning graph that has this predicate in its post condition.

5. If the sets of predicates in the current state layer of the two graphs are the same and there are more action layers:
   (a) Repeat step 2) with the next action layer of the two graphs as the current layers.

Finally, we present a worked out example for the domain PAG.

4.3 Working Example of Explanation Constructor

For the exercise, we consider the PDDL problem, PAG-problem1 from Section 2. We edited the action `user-visits-site` and included all the parameters, pre-conditions and effects from `user-starts-email` in `user-visits-site`. In essence, the two actions were combined and put in `user-visits-site`. This was to simulate an inadvertent change to the domain.

The planner runs on the last known stable version of the domain and the problem, and then on the new version of the domain and the problem. By constructing planning graphs for both instances and crawling the planning graphs, the tool displays the change in domain that caused the new plan to be different from the previous version. The original actions in the PDDL definition of PAG-problem1 are:

(1) `user-visits-site` and (2) `user-starts-email`. A new updated PDDL Domain needs is generated with the following new actions: (1) `user-visits-site` and (2) `user-starts-email`:

(:*action* user-visits-site :*parameters* (?User ?Browser ?Site ?Mailer) :*precondition* (and (user ?User) (software ?Browser) (browser ?Browser) (site ?Site) (user ?User) (mailer ?Mailer)) :*effect* (and (use-software ?Browser) (user-visits-site ?User ?Site) (use-software ?Mailer) (running ?Mailer)))
(:*action* user-starts-email :*parameters* (?User ?Mailer) :*precondition* (and (user ?User) (mailer ?Mailer)) :*effect* (and (use-software ?Mailer) (running ?Mailer)))

Finally, the updated plan from new PDDL domain for PAG-problem1 is given by:

```
0: ATTACKER-SENDS-EMAIL-WITH-KEYLOGGER USER1 FILE-WITH-TROJAN KEY-LOGGER1
1: USER-VISITS-SITE USER1 BROWSER-IE VBSCRIPT-LINK GMAIL
2: USER-READS-EMAIL USER1 GMAIL BAD-EMAIL
3: USER-OPENS-ATTACHMENT USER1 BAD-EMAIL FILE-WITH-TROJAN GMAIL
4: KEY-LOGGER-INSTALLED USER1 FILE-WITH-TROJAN KEY-LOGGER1
5: USER-PRESSES-F1-AT-VBSCRIPT-SITE USER1 BROWSER-IE VBSCRIPT-LINK
6: KEY-LOGGER-ACTIVATED KEY-LOGGER1 BROWSER-IE
7: USER-LOGIN-WITH-KEYLOGGER-ACTIVATED USER1 ACCOUNT-BANK KEY-LOGGER1
8: ATTACKER-INTERCEPTS KEY-LOGGER1 ACCOUNT-BANK
```

Below is an example output of AGBuilder on all the problems for the last known stable version of the domain and the new version of the domain and reporting changes:

```
Problems found: 1) problems/PAG-problem1.pddl: PAG-problem1
Archived plans found: 1) archivedPlans/PAG-problem1.plan

Creating planning graph for domainPAG.pddl, PAG-problem1.pddl and PAG-problem1.plan ...done
Creating planning graph for domainPAGV2.pddl, PAG-problem1.pddl and new plan ...done

plans different? True
Difference observed:
State Layer 1: same
Action Layer 1: same
State Layer 2: same
Action Layer 2: same
State Layer 3: different!
extra predicates found in action: USER-VISITS-SITE
extra effects:
1) (use-software ?Mailer)
2) (running ?Mailer)
```

As shown above, the tool runs on the two versions of the PAG domain and PAG-problem1. It crawls the two planning graphs generated using both versions of the domain, the problem and the two versions of the plans. The plan generated from the new domain does not include "USER-STARTS-EMAIL USER1 GMAIL" because "USER-VISITS-SITE USER1 BROWSER-IE VBSCRIPT-LINK GMAIL" has all of its pre-conditions and post-conditions. AGBuilder finds a disparity at state layer 3, which is after USER-VISITS-SITE was run for both domains and complaints that the predicates in the state layers were

different and leads to the action which caused this change. This is how the tool generates an explanation for the change observed in plans.

5 Related Work

There is a wealth of knowledge [13, 19, 12, 28, 16, 10, 21, 22, 4, 2] on attack graphs and attack trees. We discuss a few works in brief here.

In [20], demonstrated the first technique for automatic generation of attack graphs using symbolic model checking techniques. However, this approach suffers from scalability issues. In [13], the authors model the attack graph generation as a logic programming problem in Prolog. Their tool MulVAL takes in information from vulnerability databases, configuration information from each machine and the network. Once the entire information is available and encoded in the MulVAL framework, for a given host and policy configurations, an attack simulation is performed by the MulVAL scanner for policy violations or presence of exploit paths. However, even small changes in the configuration will require the simulations to be rerun again. In AGBuilder, the configuration information can be updated incrementally and generate new attack paths. Furthermore, we can combine two smaller PDDL domains of different networks and check for attack paths. While the underlying logical framework remains same, the planning domain formulation allows for simpler way to perform incremental updates and analysis. In [16], the authors describe Bayesian Attack Graphs that not only encodes cause-consequence relationship between network states but also consider the likelihoods of exploiting these relationships. However, none of the existing works did considered building attack graphs from natural language descriptions of the attack/vulnerabilities.

Prior work in the area of applying AI planning in cyber-security applications, provide evidence for the ability of PDDL being used to model attack-graphs. Mark Roberts et al. have demonstrated that PDDL can be used for modeling an attack graph for use in personalized security agents. [18].

Existing tools for PDDL allow for automatically generating PDDL code from formal models, developing PDDL code in an IDE with an integrated planner, and determining whether plans generated by a planner are correct [25]. AGBuilder seeks to compliment the existing research on knowledge engineering tools by assisting in the generation and incremental development of very large and complex domains.

ItSimple [26, 27] is a knowledge engineering tool that uses simple UML models and converts them into PDDL representations. Invariants (constraints) on UML classes are represented using OCL (Object Constraint Language). According to the authors, ItSimple was designed to give support to users during the construction of a planning domain application mainly in the initial stages of the design life cycle" ItSimple 2.0 [24, 27] assists users to resolve issues during requirements specification, analysis and modeling phases. ItSimple 2.0 allows users to build use case diagrams from UML, thus allowing the requirements to be represented at a high level of abstraction. ItSimple 4.0 also features a text editor for further editing after the PDDL domain has already been created, uses

modeling patterns (essentially a set of common planning models in UML), time based models that describe how properties of objects change during the execution of an action, and a wizard that allows users to quickly select the initial preconditions and goal states for actions [27].PDDLStudio is an IDE for PDDL that features document management, syntax highlighting, code completion with hints, and planner integration [15].

ModPlan is an integrated environment that allows for knowledge acquisition and domain analysis in planning applications. ModPlan helps in knowledge engineering by examining pairwise dependencies in actions and letting the user examine these dependencies [5]. It allows for plans to be visualized using Vega and validated using VAL.

VAL is used to determine if the plans generated by a planner are correct. It first examines if the domain and the problem are syntactically correct and if the objects, preconditions and goal state(s) in the problem match the domain. It then validates the plan generated by a planner for a given domain and a problem, and reports if the plan is flawed, and how it might be fixed [9].

6 Conclusion and Future Work

In this paper, we presented a tool, AGBuilder, that generates attack graphs in PDDL from textual descriptions of CVEs. The tool assists developers with incrementally developing PDDL domains for modeling attack graphs by generating explanations for why undesired plans are produced when PDDL domains are modified, and also allows for what-if analysis of attacks with the PDDL representation. The tool also allows for generating abstract syntax trees of PDDL domains and problems, and generating attack graphs as DOT files, which can then be rendered into image files, in order to visualize attack paths. We demonstrated our tool on an NVD description of a key-logger malware propagation and showed the corresponding attack graph/paths in PDDL.

Future work will look into extending the tool to perceive the current state of the system. This will allow us to estimate courses of actions that could lead to an attack in real time. Secondly, we will obtain quantifiable quality measurements of AGBuilder's scalability. Thirdly, we will investigate the feasibility of stitching together attack paths from the various attack graphs modeled by the attack graph domain to build a single consolidated attack graph. This task will require using the domain and all problem files to generate a DOT file that can then be rendered into an image file. Since the AGBuilder already has the capability of building abstract syntax trees and attack graphs for individual graphs, we anticipate this feature enhancement to be an immediate attainable goal. Finally, we will seek to integrate AGBuilder with an online IDE that we are in the process of developing. This will allow for multiple users to collaborate and build a PDDL domain.

Acknowledgment

This work is partially supported by the U.S. Department of Energy under award number DE-NE0008571 subcontracted through the Ohio State University. This

material is also based upon work performed by Indrajit Ray while serving at the National Science Foundation. Research findings presented here and opinions expressed are solely those of the authors and in no way reflect the opinions the DOE, the NSF or any other federal agencies.

References

1. Ammann, P., Wijesekera, D., Kaushik, S.: Scalable, graph-based network vulnerability analysis. In: Proceedings of the 9th ACM Conference on Computer and Communications Security. pp. 217–224. CCS '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/586110.586140
2. Audinot, M., Pinchinat, S., Kordy, B.: Guided design of attack trees: a system-based approach. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). pp. 61–75. IEEE (2018)
3. Boddy, M.S., Gohde, J., Haigh, T., Harp, S.A.: Course of action generation for cyber security using classical planning. In: ICAPS. pp. 12–21 (2005)
4. Cao, C., Yuan, L.P., Singhal, A., Liu, P., Sun, X., Zhu, S.: Assessing attack impact on business processes by interconnecting attack graphs and entity dependency graphs. In: IFIP Annual Conference on Data and Applications Security and Privacy. pp. 330–348. Springer (2018)
5. Edelkamp, S., Mehler, T.: Knowledge acquisition and knowledge engineering in the modplan workbench. Proceedings of the First International Competition on Knowledge Engineering for AI Planning pp. 26–33 (2005)
6. Esuli, A., Sebastiani, F.: Sentiwordnet: A publicly available lexical resource for opinion mining. In: LREC. vol. 6, pp. 417–422. Citeseer (2006)
7. Ghosh, N., Ghosh, S.K.: A planner-based approach to generate and analyze minimal attack graph. Applied Intelligence 36(2), 369–390 (2012)
8. Hoffmann, J.: The metric-ff planning system: Translating"ignoring delete lists"to numeric state variables. Journal of Artificial Intelligence Research 20, 291–341 (2003)
9. Howey, R., Long, D., Fox, M.: Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In: Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on. pp. 294–301. IEEE (2004)
10. Jhawar, R., Kordy, B., Mauw, S., Radomirović, S., Trujillo-Rasua, R.: Attack trees with sequential conjunction. In: IFIP International Information Security and Privacy Conference. pp. 339–353. Springer (2015)
11. Klein, D., Manning, C.D.: Accurate unlexicalized parsing. In: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1. pp. 423–430. Association for Computational Linguistics (2003)
12. Lippmann, R.P., Ingols, K.W.: An annotated review of past papers on attack graphs. Tech. rep., MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB (2005)
13. Ou, X., Boyer, W.F., McQueen, M.A.: A scalable approach to attack graph generation. In: Proceedings of the 13th ACM conference on Computer and communications security. pp. 336–345. ACM (2006)
14. Parr, T.J., Quong, R.W.: Antlr: A predicated-ll (k) parser generator. Software: Practice and Experience 25(7), 789–810 (1995)

15. Plch, T., Chomut, M., Brom, C., Barták, R.: Inspect, edit and debug pddl documents: Simply and efficiently with pddl studio. System Demonstrations and Exhibits at ICAPS pp. 15–18 (2012)
16. Poolsappasit, N., Dewri, R., Ray, I.: Dynamic security risk management using bayesian attack graphs. IEEE Transactions on Dependable and Secure Computing 9(1), 61–74 (2012)
17. Ray, I., Poolsapassit, N.: Using attack trees to identify malicious attacks from authorized insiders. In: ESORICS. vol. 3679, pp. 231–246. Springer (2005)
18. Roberts, M., Howe, A.E., Ray, I., Urbanska, M.: Using planning for a personalized security agent. In: Workshop on Problem Solving using Classical Planners at 26th AAAI Conf. on Artificial Intelligence (2012)
19. Sawilla, R.E., Ou, X.: Identifying critical attack assets in dependency attack graphs. In: European Symposium on Research in Computer Security. pp. 18–34. Springer (2008)
20. Sheyner, O., Haines, J., Jha, S., Lippmann, R., Wing, J.M.: Automated generation and analysis of attack graphs. In: Proceedings 2002 IEEE Symposium on Security and Privacy. pp. 273–284. IEEE (2002)
21. Singhal, A., Ou, X.: Security risk analysis of enterprise networks using probabilistic attack graphs. In: Network Security Metrics, pp. 53–73. Springer (2017)
22. Sun, X., Dai, J., Liu, P., Singhal, A., Yen, J.: Using bayesian networks for probabilistic identification of zero-day attack paths. IEEE Transactions on Information Forensics and Security 13(10), 2506–2521 (2018)
23. Toutanova, K., Manning, C.D.: Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In: Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics-Volume 13. pp. 63–70. Association for Computational Linguistics (2000)
24. Vaquero, T.S., Romero, V., Tonidandel, F., Silva, J.R.: itsimple 2.0: An integrated tool for designing planning domains. In: ICAPS. pp. 336–343 (2007)
25. Vaquero, T.S., Silva, J.R., Beck, J.C.: A brief review of tools and methods for knowledge engineering for planning & scheduling. KEPS 2011 p. 7 (2011)
26. Vaquero, T.S., Tonidandel, F., Silva, J.R.: The itsimple tool for modeling planning domains. Proceedings of the First International Competition on Knowledge Engineering for AI Planning, Monterey, California, USA (2005)
27. Vaquero, T., Tonaco, R., Costa, G., Tonidandel, F., Silva, J.R., Beck, J.C.: itsimple4. 0: Enhancing the modeling experience of planning problems. In: System Demonstration–Proceedings of the 22nd International Conference on Automated Planning & Scheduling (ICAPS-12). pp. 11–14 (2012)
28. Wang, L., Singhal, A., Jajodia, S.: Measuring the overall security of network configurations using attack graphs. In: IFIP Annual Conference on Data and Applications Security and Privacy. pp. 98–112. Springer (2007)
29. Weerawardhana, S., Mukherjee, S., Ray, I., Howe, A.: Automated extraction of vulnerability information for home computer security. In: International Symposium on Foundations and Practice of Security. pp. 356–366. Springer (2014)