Opleiding Informatica

ADTLang:
A declarative language to describe attack defense trees

Stephan Gabriel Meza Orellana

Supervisors:
Nathan Daniel Schiele & Dr. Olga Gadyatskaya

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

07/07/2023

Abstract

Attack defense trees are commonly used in the field to represent attacker paths and defensive measures. However, existing tools like ADTool and Attack Tree have limitations, allowing for example the creation of invalid attack defense trees in the case of Attack Tree. To overcome this, ADTLang is introduced as a domain-specific language that generates valid attack defense tree descriptions. ADTLang provides strong validation and compatibility with ADTool. We also implemented a test suite to showcase the correctness and robustness of ADTLang. Each test case is used to ensure that the requirements of ADTLang are met. ADTLang provides a benefit to cybersecurity experts and students, by providing a language that can clearly and easily define semantically valid attack defense trees.

Contents

1 Introduction **1**
1.1 Thesis overview . . . 1

2 Attack Defense Trees **3**
2.1 ADTool . . . 3
2.2 ADTool and XML file format . . . 6

3 Related Work **8**
3.1 Programming libraries . . . 8
3.2 Visualization . . . 8
3.3 Scripting languages . . . 8

4 ADTLang **13**
4.1 Create statements . . . 13
4.2 Append statements . . . 14
4.3 Syntax and comparison to ADTerm . . . 14
4.4 From ADTLang code to XML . . . 14
4.5 Validation . . . 15
4.6 Robustness and correctness . . . 16

5 Theory **20**
5.1 Formal Languages . . . 20
5.2 Context-Free Grammars . . . 20
5.3 Parse Trees . . . 21
5.4 Parsing Expression Grammars . . . 22
5.5 Parsing . . . 23
5.5.1 Parsing PEGs and Recursive Descent . . . 23
5.6 Parser Generators & Tools . . . 24

6 Implementation **26**

6.1 Grammar and Parsing . . . 26

6.2 Abstract syntax tree & Semantics . . . 28

6.3 Intermediate representation . . . 29

6.4 Generating XML . . . 29

7 Conclusion **33**

References **35**

1 Introduction

According to the national coordinator for security and counterterrorism, in recent years cybercrime has become more scalable and reached industrial proportions; this has led to an increase in victims, damage, and criminal proceeds [ncfsc]. Security experts in both the private and public sectors need ways to model the attack vectors of these cyberattacks. Attack defense trees are a model used within the cybersecurity world to describe the path an attacker might take and the defenses a defender can employ.

ADTool is a domain-specific software used to render attack defense trees. Attack Tree is another tool used to visualize attack defense trees. ADTool and Attack Tree both have limitations. In the case of Attack Tree, it is possible to generate attack defense trees that do not adhere to the definition of an attack defense tree. ADTool on the other hand provides strong validation. The issue with ADTool is that the language it provides (ADTerm) to generate attack defense trees is in our opinion not readable. Due to these limitations, we feel there is a gap for a language that has a simple syntax and strong validation.

ADTLang is a descriptive domain-specific language used to generate a description of an attack defense tree. ADTLang has strong validation ensuring that any tree generated with it is semantically correct and adheres to the definition of an attack defense tree. Furthermore, ADTLang provides a simple syntax based on commands, ensuring readability when creating attack defense trees. ADTLang is compatible with ADTool.

Our aim with ADTLang is to facilitate the process of creating attack defense trees by providing strong semantics, validation, and readability. We think introducing a new language to describe attack defense trees we will contribute to the cybersecurity community. When a cyber-attack occurs, a quick and effective response is important to mitigate potential damage. Attack defense trees created with ADTLang can provide a clear and structured representation of the attack vectors and potential countermeasures. Furthermore, ADTLang has potential in the field of education, ADTLangs readability makes it a valuable educational tool for cybersecurity training and academic purposes.

1.1 Thesis overview

Here we present an overview of the thesis. Chapter 2 describes the definition of attack defense trees and their constraints. We dive into what software is used to visualize attack defense trees, and the file format software uses to store the description of attack defense trees. Chapter 4 we discuss what ADTLang is, how it works, and the internals. In chapter 3 we discuss what other research has been done regarding the visualization of graphs and trees using other programming languages, tools, and libraries. Furthermore, we explore languages used to process XML files. Chapter 5 explores how formal languages are defined, and the process of parsing. We explain context-free grammars and parsing expression grammars, and the recursive descent algorithm which is used to parse parsing expression grammars. We end this chapter with tools that are used to generate parsers. In chapter 6 we describe the process and stages used to implement ADTLang, and we apply everything we talked about in the theory chapter. We go over which structures we used to represent the theoretical notions of the previous chapter. Lastly, in chapter 7 we conclude the thesis with how ADTLang could be expanded to support other features in the definition of attack defense trees.

I would like to thank both of my supervisors Nathan Daniel Schiele and Olga Gadyatskaya. Thank you Nathan for all the ideas and feedback which we discussed when implementing ADTLang and the feedback provided on this bachelor thesis for the Leiden Institute of Advanced Computer Science.

2 Attack Defense Trees

Attack defense trees (ADTrees) are a graphical representation of potential paths an attacker could use to target a system and the countermeasures that a defender can implement to protect the system [Kor13c]. Attack defense trees are built upon the attack tree model defined by B. Schneier in Attack Trees [Sch99]. Attack defense trees consist of the following definitions and constraints [Kor13c]:

- A root node
- A node can either be an attack or defense node
- A node contains a label
- Nodes can have one or more children of the same type. These children nodes represent a sub-goal of the parent node
- A node with children of the same type has a refinement. A refinement is the relationship between the children nodes to the parent node. An 'and' refinement means that all the children node's goals must be true in order for the node's goal to also evaluate to true. An 'or' refinement means only 1 child's goal must evaluate to true so that the node's goal evaluates to true.
- Leaf nodes represent basic actions, leaf nodes have no refinements
- Nodes can at most have one child of the opposite type, this children node represents a countermeasure

The refinement between the children of a node N describes the relationship between each child of N, for all children of N of the same type as N. OR, AND, and SAND, refinements describe if all only one component (in the case of OR) or all components need to be completed, and in which order (In the case of the SAND refinement) [Kor13c]. As an attack defense tree is still a tree an important constraint is that there are no cycles in the tree. Let us look at an example of an attack defense tree.
In figure 1 we can see the root node has the label 'Bank Account'. The goal of the attacker is to gain access to a person's bank account. We can think of each attack node child of the root node as a sub-goal the attacker can take or has (in the case of an 'and' refinement) to take. Leaf nodes can be seen as direct actions the attacker can do. Both attackers and defenders can employ a countermeasure on a node. We can see this in the node with the label 'Online' which has a countermeasure of 'MFA'. The 'MFA' Node itself has a countermeasure of 'Malware'. It is important to remember that a node can have at most one countermeasure (i.e. a child of the opposite type).

2.1 ADTool

ADTool is a program with the aim "to provide security consultants as well as academic researchers with a rigorous but user-friendly application that supports security analysis" [Kor13a]

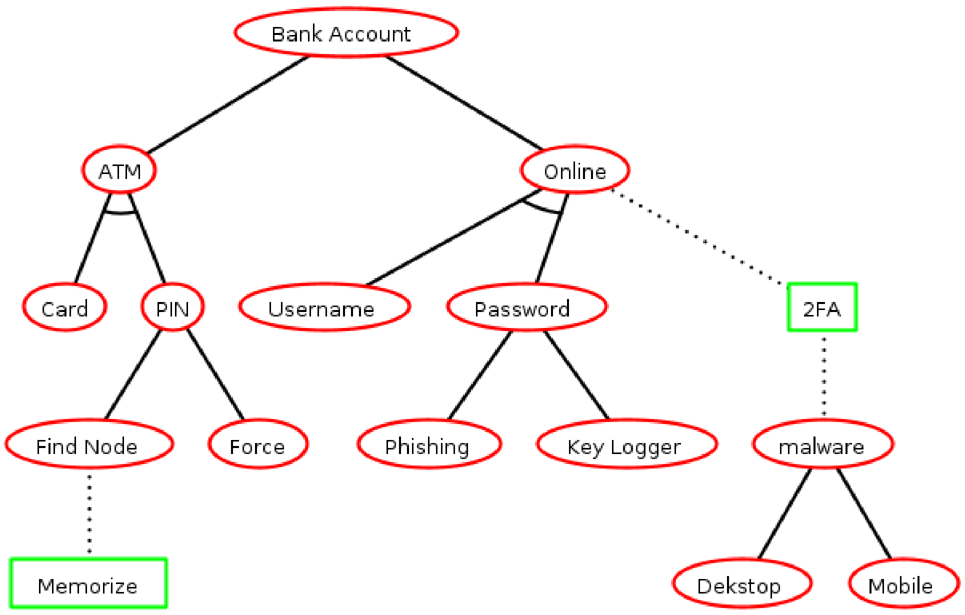


Figure 1: Attack defense tree of gaining access to someone's bank account (Generated with ADTool)

ADTool can store the tree in different file formats. One of them is XML. ADTool provides two ways to create an ADTree. The first way is to right-click and use a drop down menu, or use shortcuts to create nodes and add labels [Kor15].The second way is by using a language defined by ADTool called ADTerm. The syntax of ADTerm consists of the following six operators [Kor15]:

- *op*(): OR operator corresponding to the defender.
- *ap*(): AND operator corresponding to the attacker.
- *oo*(): OR operator corresponding to the defender.
- *ao*(): AND operator corresponding to the attacker.
- *cp*(): countermeasure operator corresponding to the defender.
- *co*(): countermeasure operator corresponding to the attacker.

The operators can take in a label (labels cannot contain commas or brackets) or another operator as parameters. The AND and or operators are n-ary and can take in operators of the same type (proponent, opponent), labels, and at most one countermeasure operator. As opposed to the AND and OR operators the countermeasure operators are binary and they take in a label or another operator of the same type and one operator of the opposite type. [Kor15]. In figure 3 we can see an example that corresponds to the attack defense tree visualized in figure 1. As we can see in figure 3 an attack defense tree described with ADTerm quickly becomes a nested structure. It is our opinion that ADTerm is not readable.

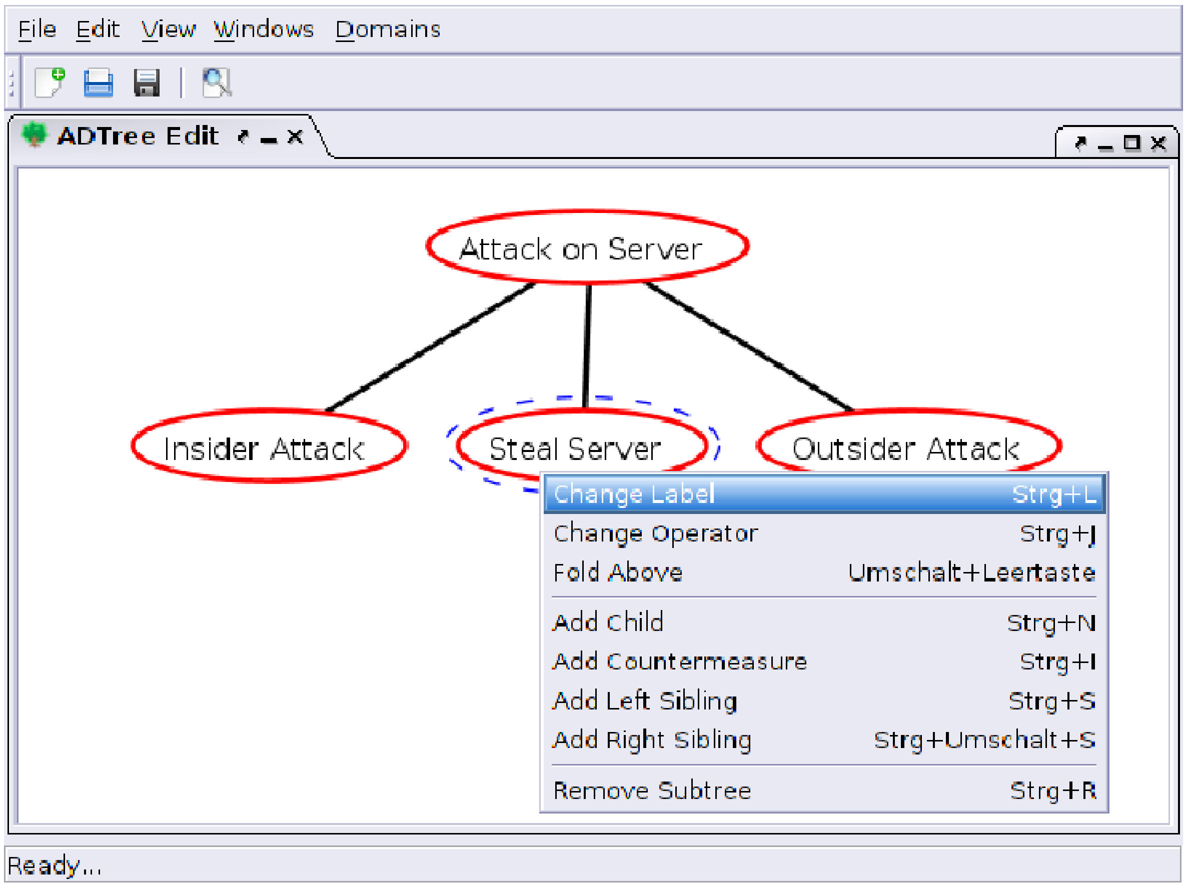
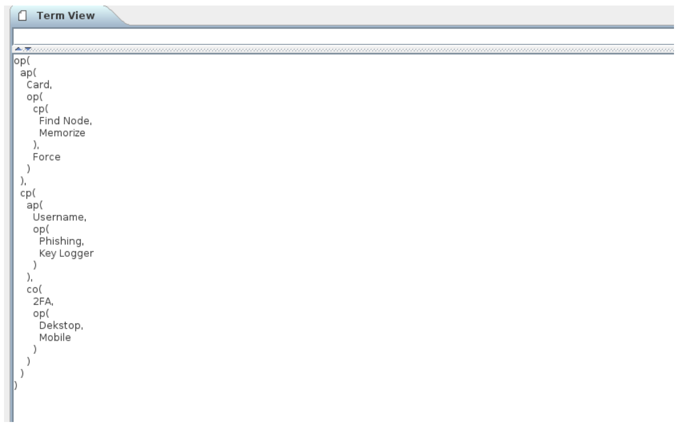


Figure 2: Creating attack defense trees with ADTool Source: [Kor13a]



```
op(
  ap(
    Card,
    op(
      cp(
        Find Node,
        Memorize
      ),
      Force
    )
  ),
  cp(
    ap(
      Username,
      op(
        Phishing,
        Key Logger
      )
    ),
    co(
      2FA,
      op(
        Dekstop,
        Mobile
      )
    )
  )
)
```

Figure 3: Creating attack defense trees with the language provided by ADTool

2.2 ADTool and XML file format

Extensible Markup Language (XML) is a markup language and file format. Its primary purpose is to transform, transmit and reconstruct any structured or unstructured data [W3C08a]. It describes a series of specifications to encode a document in a format that is readable to both machines and humans. An XML file can consist of the following:

- **Character**

  A document in the XML format consists of a set of characters. And a character is defined as any legal unicode character

- **Tag**

  A tag is a set of characters that starts with the `<` character and terminates with the `>` character. For our purposes the following two types of tags are important

  – Begin-tag, for example `<node>`

  – Stop-tag, for example `</node>`

- **Element**

  An element consists of a begin-tag and a corresponding stop-tag. The contents of an element are the set of characters between the begin-tag and stop-tag. this can include more markup and other elements; elements inside other elements are called children elements. Here we can see what an XML element looks like: `<p>This paragraph is short!</p>`. We can already see that XML can support a tree structure by having a root element that contains children elements.

- **Attribute**

  An attribute consists of a key-value part. Attributes are part of a tag specifically a begin-tag. We place them after the name of the tag. Here is an example `<img dest="Fannon.png" width="1080" height="720" />` where the attributes are `"dest"`, `"width"` and `"height"` and the values are `"Fannon.png"`, `"1080"` and, `"720"`. Attributes contain meta-data or additional information about the XML element.

- **XML declaration**

  An XML document can begin with an XML declaration. A declaration contains metadata of the XML. An example is `<?xml version="1.0" encoding="UTF-8"?>`. This serves the purpose of informing the XML parser which encoding and XML version to use. [W3C08a]

From version 1.2 ADTool supports exporting and importing XML documents. We can represent and attack defense tree in the XML format by defining an XML schema called an XSD(XML Schema Definition). ADTool implemented the ADTree language schema, adtree.xsd [Kor15]. An XML file conforms with the adtree.xsd schema if the document contains exactly one `adtree` element that represents the attack defense tree, each node of the attack defense tree is represented by a `node` element. The `node` element has the following properties [Kor15]:

- `label`: Corresponds to the label of the node.

- `node:` Represents a node's children, including countering nodes.
- `refinement attribute:` This is how we specify the refinement of a node. The refinement attribute consists of two distinct values: `disjunctive` (or) and `conjunctive` (and).
- `switchRole attribute:` Used to specify the type of a node. It specifies the type relative to its parent node. It is an attribute of type bool the possible values are: `yes` and `no`.

The values of the `label` element are strings that must adhere to the following regular expression.

```
^([0-9]|[a-z]|[A-Z]|\s|\t|(\?|!|-|_|\.))+$
```

This means the string is only allowed to consist of digits between 0-9, upper and lower-case letters from the English alphabet, spaces or tabs, and the following symbols `?, !, -, _, .`. The schema is actually broader and more attributes are supported however for our purposes, the described properties are enough.

3 Related Work

At the core of ADTLang we have a scripting language that can generate a description of an attack defense tree in an XML-formatted document in the adtree.xsd schema.

3.1 Programming libraries

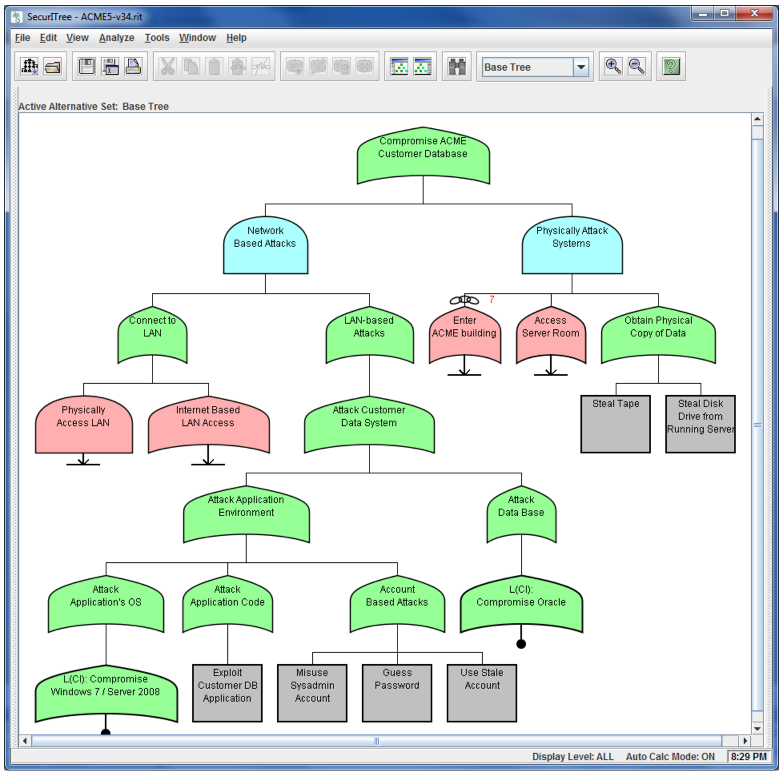
Many programming languages have libraries that allow the creation of XML files. Here are some examples. Javascript has XMLBuilder2 [ooz], Python has xml.etree.ElementTree [Lun], and C++, RapidXML [Kal]. All of these libraries are based on the object-oriented paradigm. In all three there is an XML class that contains a root node. You can create a variable with a node and append it to other nodes using member methods. [rap] [ooz] [Lun] Using any of those libraries, however, has to prerequisite of knowing those programming languages.

3.2 Visualization

The survey by Barbara Kordy. [Kor13b] provides an overview of different software which is used to visualize attack (defense) trees. Commercial software includes SecurlTree and AttackTree+. Academic tools include SeaMonsters, SQUARE Tool, and Attack Dog. ADTool also falls under this category however, due to how this thesis is structured we have decided to discuss this in chapter 2. In figure 4, figure 6, and figure 1 we see how attack trees are visualized by ADTool, SecurlTree, and SeaMonsters respectively. As we can see there is no standard way to visualize attack defense trees. If we compare figure 4 which is an attack defense tree generated with SecurlTree with figure 1 which was generated with ADTool we can see that SecurlTree uses logical gates to visualize refinements. The 'or' gate for an 'or' refinement and an 'and' gate for the 'and' refinement. ADTool on the other hand visualises refinements with an arched line between nodes to visualise an 'and' refinement, and no line between nodes for an 'or' refinement. Another important distinction is that SecurlTree generates attack trees not attack defense trees.

3.3 Scripting languages

ADTool also falls under this category however, due to how this thesis is structured we have decided to discuss this in chapter 2. There are also formats and scripting languages which are used to generate graphs or trees in general. The most famous one being the DOT language. The DOT language can describe undirected graphs, directed graphs, labels and attributes [GKN15]. It also has a lot of flexibility in how to display the shape of the nodes and color outline. The DOT language itself only provides a way to describe a graph. In order to visualize or render the graph a separate program or library, like GraphVIZ [GKN15] or a programming language library like Viz.js is needed. Figure 7 showcases a simple DOT program corresponding to a very simple attack defense tree. In figure 8 we see how GraphViz renders the program. The DOT format can be used to describe and visualize attack defense trees. At first, it seemed like a possibility to let ADTLang compile to a DOT format program as the rendering part would be trivial. One thing we quickly encountered is that the support for an ‘and‘ refinement using the DOT Format might not be possible. We found a way to generate an ‘and‘ refinement, however, the rendering did not look good so we dropped this idea.


SecurITree - ACME5-v34.rit
File Edit View Analyze Tools Window Help
Base Tree
Compromise ACME Customer Database
Network Based Attacks
Physically Attack Systems
Connect to LAN
LAN-based Attacks
Enter ACME building
Access Server Room
Obtain Physical Copy of Data
Physically Access LAN
Internet Based LAN Access
Attack Customer Data System
Steal Tape
Steal Disk Drive from Running Server
Attack Application Environment
Attack Data Base
Attack Application's OS
Attack Application Code
Account Based Attacks
L(CI): Compromise Oracle
L(CI): Compromise Windows 7 / Server 2008
Exploit Customer DB Application
Misuse Sysadmin Account
Guess Password
Use Stale Account
8:29 PM

Active Alternative Set: Base Tree

Display Level: ALL Auto Calc Mode: ON

Figure 4: ACME Attack Tree generated with SecurITree, Source: [acm]

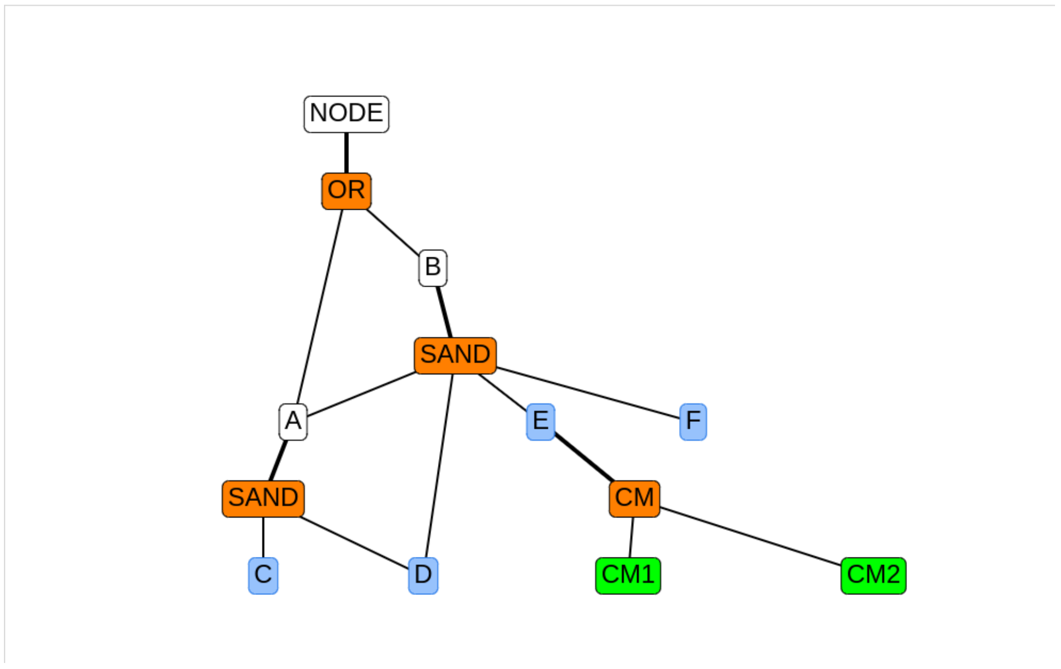


Figure 5: Attack tree generated with Attack Tree, containing DAG and two countermeasures

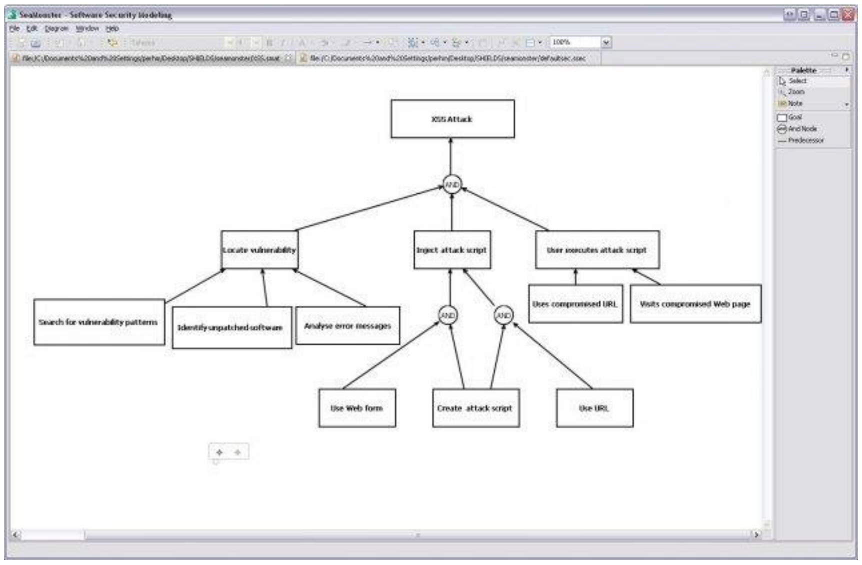


Figure 6: Attack Tree generated with SeaMonsters, Source: [mon]

```
graph StealMoneyBankAccount {
    node [shape=circle style=filled fillcolor=white \
    penwidth=2 color=red fixedsize=true width=1.5];

    root [label="Steal Money\nBank Account" labeljust="l"];
    pin [label="PIN Code" labeljust="l"];
    mobile [label="Mobile App" labeljust="l"];
    third_child [label="MFA" color=green labeljust="l"];

    root -- pin;
    root -- mobile;
    root -- third_child [style=dotted];
}
```

Figure 7: DOT Format program to generate a simple ADTree

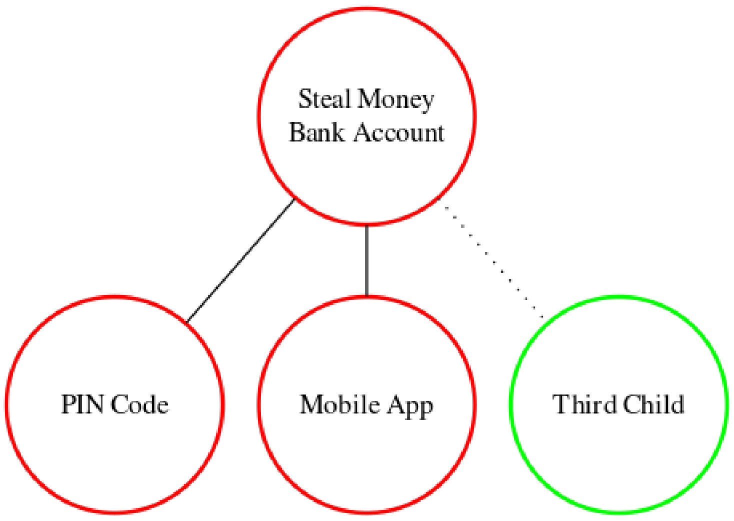


Figure 8: ADTree corresponding to DOT program in figure 7 (Generated with GraphViz)

Attack Trees is an open-source Attack Tree Modelling and Analysis Graphical Interface tool used to visualize attack trees [Wei22]. The tool was implemented in Python. What is notable to us is that, unlike ADTool, Attack Trees allows a user to input a program that describes an attack defense tree and is then rendered onto the screen. It provides a grammar to describe attack trees. The provided source input is then transpiled into a corresponding JSON file which contains the attack tree. In figure 9 we can see a program that generates a simple attack defense tree. The program must follow the grammar defined in figure 10. It must start with the 'RELATIONS' keyword in this section we can declare nodes, children, and refinements. Line 2 declares a root node called 'NODE' that has the disjunctive refinement and has two children 'A' and 'B'. Line 3 expands node 'A' by assigning two children 'C', and 'D', and giving it a sand refinement. The next section must start with the 'COUNTERMEASURE' keyword. In this section, we can create any countermeasure nodes and append them to another node. This is done in line 6. The last section starts with the 'PROPERTIES' keyword here we can add cost and property attributes to our nodes.

There has also been research done into XML querying languages. These languages are typically used to search within large XML files (Databases). XQuery [Mic04] is the Microsoft implementation of an XML querying language. In contrast with ADTLang which is used to generate XML files. In figure 11 we see a query using XQuery to search for all XML elements in a document where the price is bigger than 30, and all titles are sorted.

```
RELATIONS
NODE -OR- A,B;
A -SAND- C,D;
B -SAND- D,E,F;
COUNTERMEASURES
CM1 (E)
PROPERTIES
C:cost=1, prob=1.0;
D:cost=1, prob= 1.0;
E:cost=1, prob=1.0;
F:cost=1, prob= 1.0;
```

Figure 9: Attack Trees, attack tree program

```
RELATIONS
node -logic_gate- list_of_children
COUNTERMEASURES
cm_name(list_of_nodes)
PROPERTIES
leaf_name:cost= COST, prob = PROB
```

Figure 10: Attack Trees, language grammar

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

Figure 11: XQuery Query example, Source: [W3C08b]

4 ADTLang

In this chapter, we will give an overview of ADTLang and the internals of the language. Furthermore, we will describe how attack defense trees XML documents are generated with ADTLang. ADTLang offers an alternative to methods provided by ADTool and Attack Tree. ADTLang enforces strong validation ensuring that the tree adheres to the definition of an attack defense tree. It is our opinion that providing a programmatic way of generating attack defense trees without the need to write boiler-plate code of another programming language is a useful addition for any cyber-security researcher or academic teacher, or student who wants to use attack defense trees as a model. ADTLang was implemented with javascript. The idea is that it is a library that can be used either in a front-end or back-end javascript web application. ADTLang has the following requirements:

- Attack and Defense node types
- Counter-Measure nodes, at most one counter-measure child per node
- 'And' and 'Or' refinements
- Output is XML in xsd.schema, by definition supported by ADTool
- Exceptions are thrown to ensure no tree can be generated which does not adhere to the definition of an attack defense tree

4.1 Create statements

In the grammar of ADTLang we can create nodes with the following production rule

```
    CreateStatement
        = create NodeType (label is)? string Children? Relation?
  CreateAppend? endstatement
```

lowercase symbols are terminals with the exception of 'string' which matches any strings. The ? symbol means that the symbol before it must appear either zero or one time. This makes everything between parenthesis and the 'Children', 'Relation', and 'CreateAppend' non-terminals optional. The most simple createstatement can be the following:

```
create attack node "Bank Account";
```

we can append this node to another node as follows.

```
create attack node "Bank Account" append to "Steal Money";
```

we can optionally create children and append them to the node we are creating. That statement would look as follows:

```
create attack node "Bank Account" with children "PIN", "Credit Card", "
  MFA" append to "Steal Money";
```

Lastly, we can optionally add a type of refinement to the node we are creating.

```
create attack node "Bank Account" with children "PIN", "Credit Card", "
  MFA" has and relation append to "Steal Money";
```

4.2 Append statements

In ADTLang we can append nodes with the following production rule:

```
    AppendStatement
        = append StringList to string endstatement
```

The most simple AppendStatement can be the following:

```
append "Bank Account" to "Steal Money";
```

The statement also supports appending multiple nodes to one node. We can do this by providing a list of nodes separated by a comma. For example:

```
append "Bank Account", "Crypto Account", "Mattress" to "Steal Money";
```

It is important to note that the first node which is created will be the root node of the attack defense tree. This root node is what will appear in the generated XML. Another thing is that if certain optional non-terminals are omitted the ADTLang program will default to certain values. For example, if the 'Relation' non-terminal is omitted the node that will be created will have an 'or' refinement.

4.3 Syntax and comparison to ADTerm

As we can see in the previous subsection ADTLang is based on two commands, create and append. We implemented ADTLang around these commands because we believe explicitly declaring and appending the nodes provides a clear and readable structure for the user. Looking at the example provided in figure 14 and comparing it to figure 3 we can quickly see that ADTLang does not have a nested structure. All nodes are explicitly created. And we have an append command that explicitly appends nodes to each other. Suppose we wanted to change the tree by deleting a subtree. In ADTLang we only need to remove the command that appends the root of the subtree to our main tree. With ADTerm we would need to look inside the nested structure and find the node and remove the whole subtree. We think that the syntax of ADTLang is more readable compared to the syntax of ADTerm. The same would apply to creating a subtree. In ADTLang we can generate a subtree completely separately from the 'main'. And append it to any node on the 'main' tree. In ADTerm we would need to first figure out where in the nested structure we want to append the tree and then insert the whole subtree inside that position.

4.4 From ADTLang code to XML

We will now show what the output of the program in figure 30 is. We will use some data structures that we explain in more detail in chapter 6. We start by doing syntax analysis; we check whether the program is in the language of ADTLang. If the syntax analysis is correct we continue to the semantic analysis. During this stage, we traverse the concrete syntax tree and generate an abstract syntax tree. We perform semantic validation of the program. Next is the intermediate representation. During this stage we we traverse the abstract syntax tree and generate the statements queue. Now we will execute these statements. We will execute either a create or an append statement. During this part, we perform validation to ensure the program adheres to the definition of attack defense trees. Lastly, we traverse this XML Tree we have created and apply the algorithm in figure 34. In figure 20 we see the generated XML

4.5 Validation

In the case of a semantic error during the generation of the XML, an exception will be thrown. ADTLang throws three types of exceptions. Syntactic exceptions, semantic exceptions, runtime exceptions. A syntactic exception occurs when the provided program is not in the grammar of ADTLang. A semantic exception is thrown when the program we try to compile has errors in its logic; for example, we for example try to append to a node that has not been declared yet. Another case where a semantic exception is thrown is when we try to redeclare the same node. Lastly, runtime exceptions are thrown when the program we try to compile does not adhere to the definition of an attack defense tree, for example, we try to append a node to itself, to an ancestor, or when we try to append multiple countermeasure nodes. In each case where an exception is thrown a corresponding message explaining why the exception was thrown is given. Let us look at some examples.

```
create attack node "Bank Account";
create attack node "Bank Account";
```

The above ADTLang code will throw an exception with the following message:

```
[Semantic Error in Create Statement] node with label: "Bank Account" is
    already defined
```

Redeclaring a node is not allowed, in ADTLang.

```
create attack node "Bank Account";
append "Bank Account" to "Steal Money";
```

The above ADTLang code will throw an exception with the following message:

```
[Semantic Error in Create Statement] node with label: "Steal Money"
   cannot be found
```

The reason it generates an error is that we are trying to append to a node that does not exist. Let us look at an example that would generate a tree which does not adhere to the definition of an attack defense tree

```
create attack node "Bank Account";
create defense node "MFA";
create defense node "IP range allowlisting";
append "MFA", "IP range allowlisting" to "Bank Account";
```

The above code tries to create an attack defense tree where a node contains multiple countermeasures. ADTLang returns the following error message:

```
[Runtime error in XML generation] node with label: "Bank Account"
   already contains a countermeasure node
```

The following program tries to generate a cycle

```
create attack node "Bank Account";
create defense node "MFA" append to "Bank Account";
append "Bank Account" to "MFA";
```

We get the following error message

```
[Runtime Error in Append Statement] appending node with label: "Bank
   Account" to node with label: "MFA" creates a cycle
```

As we can see by using exceptions we ensure constraints in the definition of attack defense trees are kept. We ensure we cannot have a graph (keeping the tree structure), we ensure a node can only have one countermeasure at most.

4.6 Robustness and correctness

In order to test the correctness of ADTLang we will use construct a small test suite to check that the trees we generate are correct. These tests check that all the requirements we specified are met. In test 1 figure 12 we construct the most basic attack defense tree, a tree with a root node of attack type. Test 2 figure 13 showcases the support for 'and' refinements. Test 2 figure14 shows us that ADTool can generate both attack and defense nodes, and 'or' refinements. We see in test 14 that ADTLang supports countermeasures. Lastly, in test 4 figure 15 we can see that ADTool does not generate a diagonally acyclic graph. Instead, multiple nodes with the same label are appended to their corresponding nodes. With these tests coupled with the previous section on validation, we can see that ADTLang conforms to the requirements specified at the beginning of the chapter.

Test Name	**Description**	**Source Code**	**Output**
Test 1	ADTree 1 root	figure 12	figure 16
Test 2	ADTree 1 root node 2 children and refinement	figure 13	figure 17
Test 3	ADTreee with countermeasure node	figure 14	figure 18
Test 4	ADTreee without DAG	figure 15	figure 19

Table 1: Test Results

```
create attack node "Root";
```

Figure 12: Test 1, source code

```
create attack node "Root" with children "1","2" has and relation;
```

Figure 13: Test 2, source code

```
create attack node "Root";
create defense node "Countermeasure" append to "Root";
create attack node "1";
create attack node "2";
create attack node "3";
append "1","2","3" to "Root";
create attack node "4" append to "Countermeasure";
```

Figure 14: Test 3, source code

```
create attack node "Bank Account" with children "Web App", "Mobile App
   ";
create defense node "MFA";
append "MFA" to "Mobile App";
append "MFA" to "Web App";
```

Figure 15: Test 4, source code

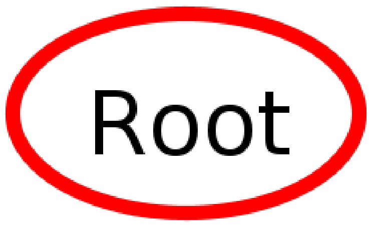


Figure 16: ADTree corresponding to test 1 in figure 12 (Generated with ADTool)

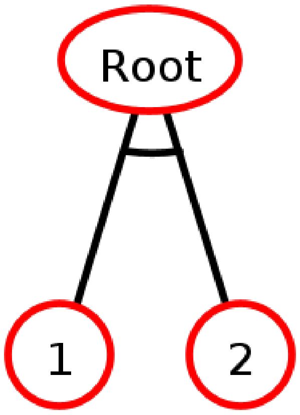


Figure 17: ADTree corresponding to test 2 in figure 13 (Generated with ADTool)

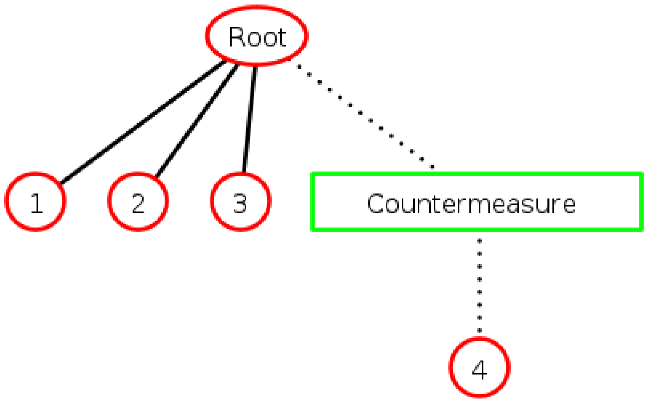


Figure 18: ADTree corresponding to test 3 in figure 14 (Generated with ADTool)

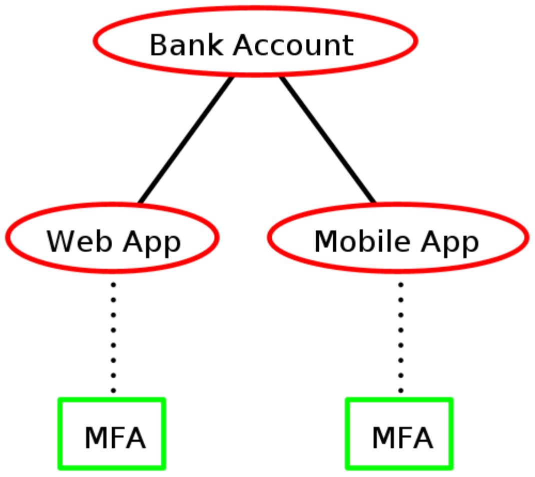


Figure 19: ADTree corresponding to test 4 in figure 15 (Generated with ADTool)

```xml
<?xml version='1.0'?>
<adtree>
  <node refinement="disjunctive">
    <label>Bank Account</label>
      <node refinement="conjuctive">
        <label>ATM</label>
          <node refinement="disjunctive">
            <label>Card</label>
          </node>
      </node>
      <node refinement="conjuctive">
        <label>Online</label>
          <node refinement="disjunctive">
            <label>Username</label>
          </node>
      </node>
  </node>
</adtree>
```

Figure 20: Generated XML corresponding to program in figure 30

5 Theory

The purpose of this chapter is to describe the theory behind formal languages and parsing. Parsing is the process of determining if a stream of characters is in a formal language, this means that the stream of characters is syntactically correct. Furthermore, in order to implement ADTLang we need to define a grammar that will generate a parser and be used to parse any ADTLang program.

5.1 Formal Languages

A formal language can be defined as a set of characters that are part of a finite alphabet. We can specify a formal language in two ways. Either with a set of rules that can generate a string in the language (regular expression, and context-free grammars). The second way is by using a machine that can accept the language, by machine we mean a computational model like a Turing-machine or a pushdown automata [Sco09]. A grammar can be written down as a set of production rules in the following way $\alpha \rightarrow \beta$. This means substitute α with β. Furthermore, a grammar always contains a starting symbol from which all substitutions must start. A symbol is either a terminal or non-terminal symbol. A terminal symbol is a symbol that is never on the left-hand side of a production rule. If a string can be generated from a given grammar using different production rules we say that the grammar is ambiguous.
Not all grammars are created equally. Noam Chomsky defined a hierarchy for different types of grammars [NC63].

- type-0, Recursively Enumerable Languages.
- type-1, Context-Senstive Languages.
- type-2, Context-Free Languages.
- type-3, Regular Languages

Type-0 grammars can express the most formal languages. The difference between each grammar is that the production rules are more restrictive the lower in the hierarchy you go. For our purpose, we will partly focus on Context-Free grammars, as some programming can be written down in using a CFG.

5.2 Context-Free Grammars

Context-free grammars are grammars where the production rules are defined as followed: $A \rightarrow \alpha$ where A is a single non-terminal symbol, α can be either a terminal or non-terminal symbol(s). We define a context-free grammar as a quadtuple $G = (V, \Sigma, P, S)$, where [Mar91]

- The set V is finite; all elements in V are non-terminals.
- The set Σ is finite; all elements in Σ are terminals. The terminals are the tokens that compose the content of a sentence in the grammar. Σ is the alphabet of the language that the grammar G defines. Lastly, $\Sigma \cap V = \{\}$.

- The P set is finite and contains all of the productions in the grammar in the form $A \rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.
- S is the start symbol, all productions start from S, $S \in V$.

The following grammar describes a trivial context-free language, where the number of a's is always double the number of b's and all b' precede the a's As we can see context-free grammars support

```
S → bSaa | ϵ
```

Figure 21: CFG Grammar for the trivial language

the notion of recursion. In figure 21, production rule 1, we see that the starting symbol S is present in both the left and right-hand sides of the production rule. The following grammars accept the same language and showcase left and right recursion

```
S → Saa | ϵ
S → aaS | ϵ
```

Figure 22: CFG Grammars showcasing left and right recursion

5.3 Parse Trees

When we generate a string to from a grammar following a production rule. We can generate a parse tree. Where every non-leaf node corresponds to a non-terminal symbol. A leaf node corresponds to a terminal symbol. If a string can be generated with different parse trees we say that the grammar is ambiguous. In figure 24 we see a parse tree that corresponds to the derivation of the string "aab" with the grammar described in figure 23

```
S → AB
A → aA'
A' → A | ϵ
B → b
```

Figure 23: CFG Grammar corresponding to the language a+b

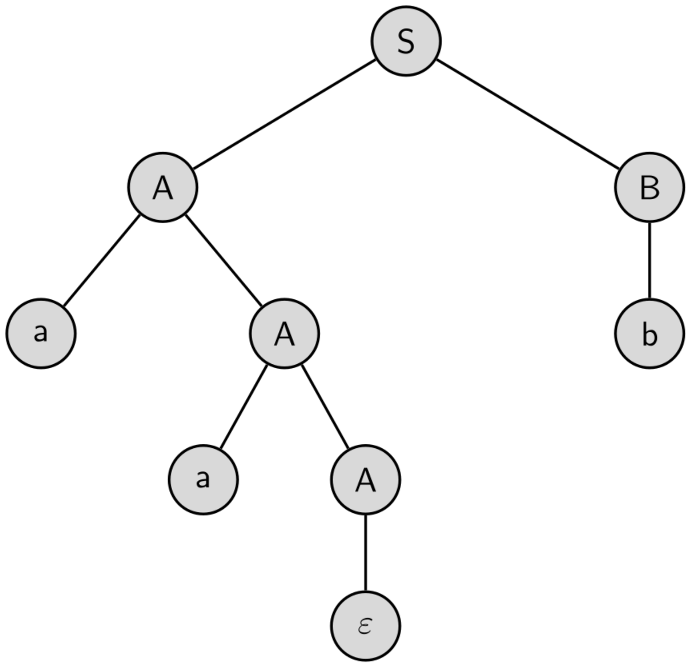


Figure 24: Parse tree for grammar in figure 23, string "aab". Source: [par]

5.4 Parsing Expression Grammars

PEGs (Parsing Expression Grammars) are a different kind of formal grammar used to describe formal languages. Every parsing expression grammar is consists of [For04]:

- The set N which is finite; all elements in N are non-terminals.
- The set Σ which is finite; all elements in Σ are terminals. It is important to note that $N \cap \Sigma = \{\}$
- The set P which is finite; which contains the production rules.
- The parsing expression e_S. The starting expression

In a parsing expression grammar rules have the following format $A \leftarrow e$, where $A \in N$ and e is a parsing expression. Similarly to regular expressions, parsing expressions have a sense of hierarchy(some operations take precedence over others). We construct parsing expression grammars in the following way [For04]:

- a parsing expression consists of one of the following options:
  - A terminal symbol
  - A nonterminal symbol
  - ε, the empty string.

- Suppose we have the parsing expressions e_1 and e_2. We can use the subsequent operators to generate a new expression:
  - Sequence: $e_1 e_2$
  - Ordered choice: e_1/e_2
  - Zero or more: e_1^*
  - One or more: e_1^+
  - Zero or one: $e_1?$

PEGs are similar to CFGs, however, an important difference is that PEGs are always unambiguous, as opposed to CFGs were only a subset of grammars are unambiguous. Unlike CFGs, PEGs cannot have left recursion [For04]. The ordered choice differs from a CFG choice '|' operator in that a PEG will match the first choice, if it succeeds it will not attempt to match the next choice. There is conjecture that not all CFGs can be written as a PEG however, there is no proof yet. As we can see PEGs give us a similar notation to CFGs with the added feature of regex-like operators. The following PEG defines a language that accepts an arithmetic expression.

```
Expr    ← Sum
Sum     ← Product (('+' / '-') Product)*
Product ← Power (('*' / '/') Power)*
Power   ← Value ('^' Power)?
Value   ← [0-9]+ / '(' Expr ')'
```

Figure 25: PEG Grammar Arithemthic Expression. [Lau21]

5.5 Parsing

Parsing is the process of tokenizing a stream of characters and determining if the stream of characters is part of a defined formal language. Tokenizing refers to breaking up the stream of characters into specific tokens, where each token is a terminal in the defined grammar.

5.5.1 Parsing PEGs and Recursive Descent

This subsection will explain how PEGs are parsed. We will focus on the recursive descent parsing algorithm as PEGs can directly be implemented with recursive descent.
Recursive descent works by defining each non-terminal symbol in a grammar as a procedure. Let us look at an example of a grammar and its associated recursive descent parser. Arithmetic expressions can be defined with the PEG grammar in figure 25, figure 27 shows us a parser implementation using recursive descent. As we can see every non-terminal corresponds to a procedure in python. Recursive descent uses backtracking to try and match every non-terminal in a production rule, in the case of ordered choice. To illustrate this let's look at the grammar in figure 26
Suppose we try to match the following string:

```
1024
```

```
S       ← P '+' S / P '+' S / P
P       ← N '*' S / N '/' S / N
N       ← [0-9]+
```

Figure 26: Simplified PEG Grammar Arithemthic Expression. [Lau21]

It is trivial for us the see that we have to pick the production rule $S \leftarrow P$, next we need to pick $P \leftarrow N$. However, recursive descent will try all possible production rules. Backtracking to a position before it tried a production rule in case there was no match. Then try the next production rule. In total, a naive implementation of recursive descent would have to call the N non-terminal production a total of 10 times while parsing the string 1024. Because of backtracking a PEG parser using recursive descent could experience a run-time of exponential complexity. Using a packrat parser it is possible to implement PEG parser that runs in linear time. This is accomplished by using memoization [For04]. This is done by caching every (sub)expression called; remembering if it was a match or not. This way if the same (sub)expression is invoked we only explore it, if it is not in the cache. This way it is ensured that each (sub)expression is called at most once for a given position in the input. In our example, we would call the sub-expression N only 1 time. This comes at a cost as we will need to use memory to store the intermediate results. PEG grammars are of great interest to us as the grammar of ADTLang is defined with a PEG grammar. It is important to understand both the pros but also the limitations of a PEG grammar. As ADTLang is a small language a lot of the features of a PEG grammar were not used, as they were not needed. For example, the 'and' and 'not' predicates were not used while designing a grammar for ADTLang.

5.6 Parser Generators & Tools

As we can see the simple grammar described in figure 25 containing 5 production rules requires around 40 lines of (unoptimized) Python code we can see this in figure 27. Parser generators or, compiler compilers are programs that take in a formal grammar and generate the corresponding parsing code. There are many different types of compiler-compilers, based on different formal grammars and parsing algorithms. We use these parser generators to generate a parser that not only checks if a given input is in the formal grammar but also defines semantics. As will be explained in further detail in chapter 6 ADTLang uses the Ohm framework as a parser generator.

```
def nextSymbol()
def Expr():
    return Sum()
def Sum():
    L = Product()
    while symbol == '+' or symbol == '-':
        op = symbol
        nextSymbol()
        R = Product()
        if op == '+':
            L +=  R
        elif op == '-':
            L -= R
        else:
            error("Incorrect")
    return L
def Product():
    result = Power()
    while symbol == '*' or symbol == '/':
        op = symbol
        nextSymbol()
        R = Power()
        if op == '*':
            L *=  R
        elif op == '/':
            L = L / R
        else:
            error("Incorrect")
    return L
def Power():
    result = Value()
    if symbol == '^':
        nextSymbol()
        Power()
    return result
def Value():
    if symbol == '(':
        nextSymbol()
        Expr()
        assert symbol == ')'
    else:
        assert (symbol >= '0' and symbol <='9')
        res =  int(symbol)
        if i < len(stream)-1:
            nextSymbol()
        return res
```

Figure 27: Recursive descent algorithm for PEG Arithmetic expression grammar

6 Implementation

In this section, we will explain how ADTLang was implemented. ADTLang is declarative language with a simple syntax that generates an XML file in the adtree.xsd schema. A declarative programming language is a programming language where the programmer specifies the goal that should be achieved, rather than how a goal should be achieved [CS13]. What makes ADTLang a declarative language is the idea that the language is built around commands which describe which action has to be done. How this action is done does not matter to the programmer as this is all handled by the ADTLang program.
The implementation of ADTLang works with stages and passes. Each stage transforms the input into another data structure and this output is passed as the input of the next stage. ADTLang has three stages. The first stage is the parsing stage. In this stage, the input is a string that represents an ADTLang program. Here we verify that the input is indeed a part of the grammar describing ADTLang. Furthermore, we ensure that the program is semantically valid. And output an abstract syntax tree. The next stage is the intermediate representation. We take the abstract syntax tree as input, walk through the tree in the pre-order and generate a queue of statements. Lastly, we execute these statements and output an XML document. In the last stage we do further checks to ensure the tree we generate adheres to the definition of an attack defense tree.

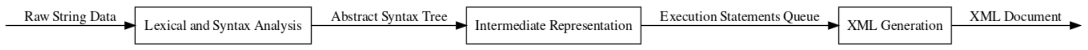

Raw String Data
Lexical and Syntax Analysis
Abstract Syntax Tree
Intermediate Representation
Execution Statements Queue
XML Generation
XML Document

Figure 28: Implementation stages of ADTLang

6.1 Grammar and Parsing

The parsing stage and the grammar of ADTLang were developed with the Ohm parsing toolkit. Ohm is a compiler-compiler framework written in javascript [War16]. The Ohm language is based on parsing expression grammars (PEGs). It reads in a formal grammar and generates corresponding javascript parsing code. The grammar format which Ohm expects; represents a PEG grammar, however, the meta-syntax is slightly different compared to the original meta-syntax described in the Bryan paper. The parsing code which the Ohm toolkit generates is a concrete syntax tree object, we can think of this as the parse tree described in chapter 5. The grammar described in figure 29 contains the core features supported by ADTLang.

```
ADTLangGrammar{
    Start
        = Statements

    Statements
        = Statement+

    Statement
        = CreateStatement
        | AppendStatement

    CreateStatement
        = create NodeType (label is)? string Children? Relation?
   CreateAppend? endstatement

    AppendStatement
        = append StringList to string endstatement

    CreateAppend
        = append to string

    Relation
        = has booleanoperator relation

    Children
        = with children StringList

    StringList
        = NonemptyListOf<string, ",">

    NodeType
        = type node

    node
        = "node"

    type
        = "attack"
        | "defense"

    string
        = quotes words quotes

    endstatement
        = ";"
    ...
  }
```

Figure 29: ADTLang PEG Grammar definition

An ADTLang program consists of statements. A statement is either a create or append statement. This way we can create and append nodes to an attack defense tree. Inside a create statement we can create children, give it a refinement, and append it to another node. The append statement lets us append a list of nodes to another node. A simple ADTLang script which is in the grammar can be seen in figure 30

```
create attack node "Bank Account";
create attack node "ATM" with children "Card" has and relation;
create attack node "Online" with children "Username" has and relation;
append "ATM", "Online" to "Bank Account";
```

Figure 30: ADTLang program to generate an attack defense tree

6.2 Abstract syntax tree & Semantics

With the grammar specified in the previous subsection, we can check if an input is part of ADTLang. This means that we can detect syntax errors. From this point, we need to determine if the input is semantically valid, and we generate an abstract syntax tree.
An abstract syntax tree is a tree data structure used to represent the structure of source code at a higher level of abstraction. The AST is useful in transforming source code from one representation to another. The AST serves as a bridge between the textual representation of source code and the execution of code. In ADTLang the abstract syntax tree consists of a tree class that has a root node and a symbol table see figure 31.
In the case of ADTLang, we define semantic analysis as checking the program for adherence to the rules and constraints of attack defense trees, as well as verifying the logical consistency and correctness of the code. Take the following ADTLang program in figure 32 as an example.

The program is syntactically valid however we try to append the 'Bank Account' node to a node that has not been declared. In the semantic analysis part of ADTLang, we check if we try to append to a node that is not present in the symbol table. The symbol table is a list that contains the labels of all nodes that we create. Whenever a node is created we check if it already exists in the symbol table, if it does exist it means we are trying to redeclare a node. Ohm provides an easy way to add semantics. When we parse an input using ohm-generated parsing code, the parser generates a concrete syntax tree. This allows us to add a function whenever a node in the concrete syntax tree is visited.

```
...
Node: class Node {
    constructor(type, values)
    {
            this.type  = type;
            this.values = values;
            this.children = [];
    }
}
AbstractSyntaxTree: class AbstractSyntaxTree {
    constructor(root)
    {
        this.root = root;
        this.symbolTable = new module.exports.SymbolTable();
    }
    getRoot()
    {
        return this.root;
    }
}
...
```

Figure 31: Javascript Objects that represent the AbstractSyntaxTree

```
create attack node "Bank Account";
append "Bank Account" to "Stealing money";
```

Figure 32: ADTLang program with semantic errors.

6.3 Intermediate representation

The next stage in the ADTLang implementation is to traverse the AST and generate a queue containing every instruction the program has to execute. This means that we turn nodes into statements and push them unto the queue. This means taking a create node and turning it into a create statement. We do the same thing for append nodes.

6.4 Generating XML

Now that we have a queue containing the instructions of a program we run through each statement sequentially and execute it. As described in Chapter 2 we described the XML format, The XML file format can be thought of as a Tree. Each tag is a node and all tags inside a tag are the children. This means that we can represent an XML object as a tree using javascript. We use the following classes defined in figure 33 to represent an XML tree and XML node. In ADTLang the first node that is created is also the root. This is how ADTLang can create different nodes without directly declaring a root. During this stage constraints are checked, ensuring that the generated ADTree adheres to the definition of an Attack Defense Tree. Here we make sure that we cannot append

```
...
XMLNode: class XMLNode
{
    constructor(label, refinement,type, parent = null)
    {
        this.label = label;
        this.refinement = refinement;
        this.children = [];
        this.type = type;
        this.parent = parent;
    }
}
...
XMLTree: class XMLTree
{
    constructor(root)
    {
        this.root = root;
        this.spacing = '';
    }
}
...
```

Figure 33: Javascript Objects used to represent an XML structure

a node to itself, and that we do not create a cycle. Also that we cannot append a node where the node to be appended is an ancestor of the appendee node. Lastly, we ensure that we cannot have more than one countermeasure per node. Once we have a tree representing the XML object. All that is left is to generate a string representation of the XML object. We can do this with the following algorithm. It is a simple pre-order traversal of the XML-Tree object. When we first visit a node we create the opening tag. Then we visit all the children recursively. When we return from the recursion we create the corresponding closing tag.

```
    ...
    XMLNode: class XMLNode {
        ...
        toString(indentation = '    ')
        {
            this.xmlstr = ''
            appendToString("<?xml version='1.0'?>")
            appendToString("<adtree>");
            this.spacing = indentation;
            this.visit(this.root, indentation);
            appendToString("</adtree>");
        }
        visit(node, indentation)
        {
            let switchRole = ''

            if(node.parent && (node.type !== node.parent.type))
                switchRole = ' switchRole=\"yes\"';

            let nodetag = indentation + '<node refinement=\"' +
                          node.refinement + "\"" + switchRole + '>';

            appendToString(nodetag);
            appendToString(indentation
                        + this.spacing
                        + '<label>'
                        + node.label +'</label>');

            node.children.forEach( c => {
                this.visit(c, indentation + this.spacing + this.spacing
  );
            });

            let endtag = indentation + '</node>';
            appendToString(endtag);
        }
    }
    ...
```

Figure 34: Generate XML format algorithm

7 Conclusion

In this thesis, we have introduced ADTLang a declarative language with simple syntax and strong validation that is capable of taking in commands and outputting an XML file in the adtree.xsd schema. We defined a set of requirements for ADTLang and implemented a small test suite to check for correctness and robustness. The test suite checked that we can create attack defense trees with attack nodes, defense nodes, countermeasures, 'and' and 'or' refinements. Also that when we generate an attack defense tree that does not adhere to the definition we receive informative and useful error messages. ADTLang provides an alternative to the current way provided by Attack Tree which can generate trees that are not in the definition of an attack defence tree, while having strong validation like ADTool. We achieve this because of ADTLang's strong validation and semantic checks in the different stages of the program. Lastly, by not implementing a nested syntax structure we believe ADTLang has more readability compared to ADTerm. We think our contribution fills in the gap we describe in chapter 1 and will allow cybersecurity experts and incident responders to quickly and correctly map out attack vectors during incident response cases. While also providing a learning tool for cybersecurity students who want to learn about attack defense trees.

ADTLang currently supports creating trees with either and/or refinements and counter-measures nodes. The adtree.xsd schema supports additional features which could be added to ADTLang. These features include the SAND refinement and adding attributes to an XML tag like a probability attribute. It could be extended to take into account any attribute supported by adtree.xsd. We can define a key followed by a value. This could further be expanded to support a list of keys and values separated by a comma. Because of how modular ADTLang was implemented adding new features means adding the required changes for every stage. The grammar could be restructured to accept to the following input.

```
create attack node "Bank Account" probability 0.6 with children "PIN",
    "Credit Card", "MFA" append to "Steal Money";
```

Right now we can create a node with multiple children in one command. A future enhancement that could be beneficial is to create multiple nodes and then append them to another node something like this.

```
create attack node "Bank Account", "Credit Card", "MFA" append to "
    Steal Money";
```

Furthermore, another feature that could be implemented in the future is modifying the create statement to allow the possibility of adding a countermeasure child node as right now you need two statements to do this. It could look like this:

```
create attack node "Bank Account", "Credit Card", countermeasure "MFA"
    append to "Steal Money";
```

Lastly, because of the modularity of the ADTLang implementation, we could even output something else besides XML. Looking at the DOT format we described in chapter 3, it might be possible to use the DOT format or another type of format to describe an attack defense tree. This could make the rendering easier if the provided format already has a rendering library.

References

[acm] Acme attack tree. "https://www.amenaza.com/s1.php".

[CS13] Amanda Clare and Martin Swain. *Declarative Language*. Springer, New York, NY., 2013.

[For04] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. *Symposium on Principles of Programming Languages*, 2004.

[GKN15] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot. 2015.

[Kal] Marcin Kalicinski. Rapidxml. "https://rapidxml.sourceforge.net/".

[Kor13a] Barbara Kordy. Adtool: Security analysis with attack–defense trees. *International Journal of Computer Science*, 42:123–456, 2013.

[Kor13b] Barbara Kordy. Dag-based attack and defense modeling: Don't miss the forest for the attack trees. 2013.

[Kor13c] Barbara Kordy. Foundations of attack–defense trees. *Formal Aspects in Security and Trust*, pages 80–95, 2013.

[Kor15] Piotr Kordy. The adtool manual. 2015.

[Lau21] Nicolas Laurent. Peg cfg semantics (performance). https://www.youtube.com/watch?v=lJcK9-0daBY&t=1116s&ab_channel=NicolasLaurent", 2021.

[Lun] Fredrik Lundh. xml.etree.elementtree. "https://docs.python.org/3/library/xml.etree.elementtree.html".

[Mar91] John C. Martin. *Introduction to Languages and The Theory of Computation*. McGraw Hill, 1991.

[Mic04] Microsoft. Xquery language reference (sql server). https://learn.microsoft.com/en-us/sql/xquery/xquery-language-reference-sql-server?view=sql-server-ver16", 2004.

[mon] Seamonster - security modeling software. "https://sourceforge.net/projects/seamonster/".

[NC63] M.P Schutzenberger Noam Chomsky. The algebraic theory of context-free languages. *Studies in Logic and the foundations of mathematics*, page 118–161, 1963.

[ncfsc] national coordinator for security and counterterrorism. Cyber security assessment netherlands. "https://english.ncsc.nl/topics/cybersecurity-assessment-netherlands".

[ooz] oozcitak. Xml builder 2. "https://oozcitak.github.io/xmlbuilder2/".

[par] Parse tree to saab. "https://en.wikipedia.org/wiki/Parse_tree#/media/File:Parse-tree.svg".

[rap] Rapidxml documentation. "https://rapidxml.sourceforge.net/manual.html".

[Sch99] Bruce Schneier. Attack trees. https://www.schneier.com/academic/archives/1999/12/attack_trees.html, 1999.

[Sco09] Michael L. Scott. *Programming Language Pragmatics (Third Edition),*. International Journal of Computer Science, 2009.

[W3C08a] W3C. Extensible markup language (xml) 1.0 (fifth edition)). "https://www.w3.org/TR/REC-xml/", 2008.

[W3C08b] W3C. Xquery tutorial. "https://www.w3schools.com/xml/xquery_intro.asp", 2008.

[War16] Alessandro Warth. Modular semantic actions. *Proceedings of the 12th Symposium on Dynamic Languages*, 2016.

[Wei22] Florentin Weiser, Thomas ; Delcourt. Graphical user interface for dag-based attack trees. 2022.