A Modular Notation for Monitoring Network Systems

by

Prashant Raghav

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Design of next generation network systems with predictable behavior in all situations poses a significant challenge. Monitoring of events happening at different points in a distributed environment can detect the occurrence of events that indicates significant error conditions. We use Modular Timing Diagrams (MTD) as a specification language to describe these error conditions. MTD's are a *component-oriented* and *compositional* notation. We take advantage of these features of MTD and point out that, in many cases, global MTD specifications describing behaviors of several system component can be efficiently decomposed into a set of sub-specifications. Each of the sub-specifications describes a local monitor that is specific to the component on which the monitor is intended to run. We illustrate the compositional nature of MTD in describing several network monitoring conditions related to network security.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor, Professor Richard Trefler for his continuous support of my masters study and research, his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better supervisor and mentor for my masters study.

Also, I would like to thank my good friends Vijay Subramanya and Hisham El-Zein who supported me through the good times and the challenging times.

Besides my supervisor, I would like to thank the rest of my thesis committee: Prof. Grant Weddell and Prof. Michael Godfrey, for their encouragement, insightful comments, and hard questions.

Last but not the least, I would like to thank my parents, Vijay and Kusum Raghav for their unconditional love, support and encouragement.

Dedication

This is dedicated to my family and my friends.

Table of Contents

List of Figures viii

1 Introduction 1

1.1 Problem Description . . . 1

1.2 Approach . . . 2

1.3 Organization . . . 3

2 Background 4

2.1 History . . . 4

2.2 Monitoring Network Systems . . . 5

2.3 Apache Spark . . . 6

2.3.1 Resilient Distributed Datasets . . . 7

2.3.2 Spark Streaming . . . 7

2.4 Monitoring Network System with MTD . . . 8

3 Modular Timing Diagrams 9

3.1 Informal Description . . . 9

3.2 Modular Timing Diagrams: Syntax & Definition . . . 10

3.3 Semantics . . . 12

3.3.1 ∀-automaton . . . 13

3.3.2 RTD Language . . . 13

3.3.3 MTD Language . . . 14

4 Security Vulnerabilities **15**
4.1 User Authentication 16
4.2 Communication Security 19
4.3 Network Access Control 20
4.4 Email Filtering 20

5 Observations **22**
5.1 KDD dataset 22
5.2 MSF File 24
5.3 Clustering and Live Streaming of Data 25

6 Conclusion and Future Work **28**

References **29**

List of Figures

2.1 Apache Spark Framework . . . 7

3.1 An example MTD . . . 10

3.2 A component RTD . . . 11

4.1 MTD of User Login - Password verification . . . 18

4.2 MTD of User login - IP Verification . . . 19

4.3 MTD of Email Filtering . . . 21

5.1 Data Format . . . 22

5.2 Distributed Monitoring System . . . 23

5.3 MSF file . . . 25

5.4 System Design . . . 26

Chapter 1

Introduction

This thesis provides a framework to monitor network systems that are vulnerable and requires constant monitoring. We accomplish this using Modular Timing Diagrams, a compositional notation, along with Apache Spark [53], a distributed computing framework. The high level specifications written in the MTD notation are converted to a text file which is then provided as an input to Spark. If some suspicious activity is detected, an alarm is raised and the network administrator is notified.

1.1 Problem Description

Multi-component, network services, such as banking and financial services, may be delivered across large disparate networks. These services need to be delivered reliably and robustly. Although simulators and network analyzers can detect some network errors, due to the potentially large state space associated with the network size, analysis may be infeasible. We use network analysis to detect unauthorized access. However network analysis is a difficult, complex and demanding task. Therefore we describe monitors designed to detect and report the occurrence of significant network events.

Formal verification is used to find intricate errors that are hard to detect by testing [35]. One approach to Formal verification is model checking [13]. In the worst case, a model checker may need to analyze all reachable system states, and even modest-sized finite state systems may have state spaces of enormous size[17]. Although in recent years model checking has seen much improvement [11, 6], applying it in real-world scenarios, such as network systems, remains a challenge. Testing [51] is used in practice to detect bugs in the system. However testing in general is, not an exhaustive procedure and it is possible that a significant bug goes unreported and later affects the system. Hence, runtime verification [33] is often used in conjunction with testing to check program correctness during system execution.

Providing a mathematically precise notation for describing essential aspects of distributed systems - in this case, system requirements - may be a complex task. Temporal logic[17] is often used in this regard. Typically, specifications for distributed systems are written from a *global* perspective, while event monitoring at the individual process locations is necessarily a *local* concern. Thus it is important to adopt a notation for writing global specifications that also characterizes the local events of interest.

1.2 Approach

Modular Timing Diagrams (MTD) [3], which is a *component-oriented* and *compositional* notation, is used as a specification language for describing the occurrence of *significant* error events during the system operation. We take advantage of the two aforementioned properties of the notation and point out that, in many cases, global MTD specifications can be efficiently decomposed into a set of sub-specifications in a straightforward manner. Each of the sub-specifications describes a local monitor that is specific to the component on which the monitor runs. A separate, standalone component maybe used to collect the result of several distributed monitoring components. In this regard, we make use of this compositional nature of MTD to translate the single specifications into distributed sub specifications of several parts. Each part in itself is an MTD describing the occurrence of error conditions at a particular location in the network. MTD components are designed to monitor all events and messages local to the network nodes they are running on and raise alarms to notify network administrators of the occurrence of a specified error event.

Modular Timing Diagrams was proposed as a notation that ties together visual specification and modular reasoning of asynchronous system. MTD notation can be used to represent universal properties of asynchronous system. Universal properties are properties that hold for all computation in the system. MTD is an extension to timing diagrams[2] used frequently in the hardware industry to specify timing and ordering properties of hardware protocols. Timing diagrams are simple and intuitive but are unable to express iterations and disjunctions. MTD on the other hand not only provides a way to represent those properties, but is also expressive enough to describe any ω-regular property.

High level specifications of a network system are described using MTD notation. These specifications are error conditions which can result in data loss or unauthorized access. The individual MTDs are then transformed into a *modular specification file*, which is described later. Once the file is generated based on the previous attacks on the system, it is easy to analyze the new requests with Apache Spark[53] and classify them as attacks or normal requests. As part of the framework we describe an approach to monitor network system using MTD. Our experience of using MTD with Spark to monitor network system is very encouraging and we were able to perform the monitoring task with minimal lines of code.

Data transfer in network protocols can lead to security vulnerabilities in the system. Transmitting data over a network allows third parties to access the data resulting in data loss. One approach in mitigating this is user authentication where a user is verified before they are provided access to the resources. Chapter 4 describes how user authentication can be represented and monitored using MTD.

1.3 Organization

The thesis is organized as follows. Chapter 2 describes the background necessary to understand the thesis. In Chapter 3 the syntax and semantics of MTD are described. Chapter 4 looks at several network security concerns and their description using the MTD's. In Chapter 5, an example environment network with MTD monitors is built with Apache Spark. A brief conclusion with suggestions for future work is given in Chapter 6.

Chapter 2

Background

The goal of this thesis is to describe the use of compositional nature of MTD for monitoring network systems. This chapter covers the theoretical background necessary to understand this work. First, the literature and related work around timing diagram's are covered in detail. Then we describe Apache Spark, the framework used in this work. Later, we look at various other ways of monitoring network systems and show how our work differs from them.

2.1 History

Timing Diagrams are often used in hardware industries to specify behaviour for circuit components. Several researchers have investigated the use of timing diagrams for verification. In his paper, Boriello et al[9] provided an approach for symbolic timing verification of timing diagrams. However timing diagrams lacks a precise notation and as such is unsuitable for verifying correctness. Therefore, a representation with more precise semantics that also allows for an efficient, compositional model checking algorithm, such as the Regular Timing Diagrams, is necessary.

Fisler et al[16] showed that we can translate timing diagrams into temporal formulas. In her paper, Amla et al [3] illustrates an efficient model checking algorithm using Regular Timing Diagrams by verifying a master-slave model. Regular Timing Diagrams[2] are an intuitive notation but one cannot express *disjunction* and *iterations* using them. Nina further proposed a class of timing diagrams called Modular Timing Diagrams[3] that not only handles these properties but is also as expressive and intuitive as the omega regular languages. In this work we show to use the Modular Timing Diagram notation to monitor network systems.

There are, however, other notations used to describe high level implementation scenarios. Message Sequence Charts[49](MSC) are the standard way for describing them, particularly in the case of communication protocols. MSC are used to represent properties written in temporal logic and then comparing the execution paths of two different processes. MTD on the other hand specifies universal properties of an asynchronous system and hence is complementary to the MSC notation. Although Timeline Diagrams [44] present universal properties too, they are limited to totally ordered sequence of events. Thus MTD can be viewed as a compact form of set of timelines. Damm et al [19] in their work proposed Live Sequence Charts. Chai and Schlingloff [14] provided an approach to monitor systems with by extending Live Sequence Charts.

2.2 Monitoring Network Systems

Formal Verification is a way of proving that system satisfies correctness properties. There are three parts of formal verification.

- A mathematically precise description of system execution behaviour.
- A logic to represent properties about the system.
- An algorithm to check if the properties are satisfied by the system.

One approach to this is model checking. In model checking systems, properties are represented by temporal logic and checks are provided to see weather the system satisfies the given properties. However model checking is not practical for network systems of even modest size because of the huge state space associated with the system [7].

Runtime Verification deals with the study and application of verification techniques that monitors the system under scrutiny at runtime and checks weather the run of the system satisfies or violates the correctness property. The main tool for runtime verification are monitors whose sole purpose is to check weather the system behaviour satisfies correctness properties. Typically monitors are generated from higher level specifications of system design. These specifications are generally written in Linear Temporal Logic [4]. The goal of thesis is to explore the use of compositional MTD notation to represent high level specifications of network system design.

There are various applications of runtime verification in software systems. Java Pathexplorer [26] is used to monitor java programs. Pike [40] gave an approach to monitor ultra critical systems. Peters and Parnas[39] suggested monitors for runtime verification of software and Hardware systems. Wilcox and Christina [51] provided an approach to effectively monitor Stochastic Systems. Temporal Rover [11] is a runtime verification tool based on future time metric temporal logic.

Networks have become more vulnerable after the shift from location-restrictive access to unrestricted access. Hewlett Packard in its report said there has been a 176% increase in cyber crime since 2010. Some of these attacks follow certain known patterns. For example, accessing a user account from a certain IP address in rapid succession is one of the ways to exploiting network services making the network inaccessible to intended users. This type of attack is classified as denial of service[29]. Other types of attacks could be sending or receiving abnormal bytes of data or accessing multiple ports simultaneously.

Various tools have been developed for monitoring network systems. They detect anomaly in the network by matching the network state to patterns or set of rules describing characteristics of anomalies[41]. Yu Gu et al. [24] proposed a technique to detect anomalies in network traffic using maximum entropy estimation. Brutlag et al [12] used Holt Winter Forecasting[15] model to predict future traffic based on the history of network traffic. Bro et al [38] monitored network system by monitoring network link over intruders traffic transit. Yuh Huang and Thomas Wicks [27] gave a distributed intrusion detection framework using attack strategy analysis. Sekar et al[42] gave a specification language for network intrusion detection system which enables a strict static and dynamic checking. This thesis introduces the use of compositional nature of MTD's in conjunction with Apache Spark for monitoring network systems.

2.3 Apache Spark

Apache Spark is a in-memory cluster computing framework designed for large scale data processing. Spark is 100X times faster than other distributed computing framework called Hadoop[43]. The speed can be attributed to the fact that Spark keeps the intermediate data cached in local JVM. Hadoop on the other hand writes the intermediate data on to the disk, which is expensive. Apache Spark applications can be written in any of Scala, Python and Java. Spark stack shown in Figure 2.1 comes bundled with tools like Spark SQL, MLlib, Spark Streaming and GraphX.

- SparkSQL: Unified access to structured data, provides compatibility with Apace Hive and standard connectivity to tools like JDBC[25] and ODBC[23].

- Spark Streaming: For scalable fault tolerant streaming applications, Spark can run in both batch and interactive mode.
- MLlib:Scalable Machine Learning library.
- GraphX:Large Scale Graph Processing Framework.

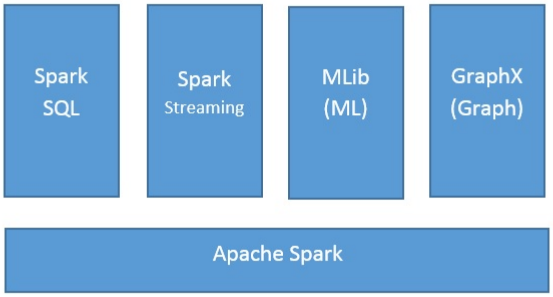


Figure 2.1: Apache Spark Framework

2.3.1 Resilient Distributed Datasets

The main abstraction for computation in Spark is Resilient Distributed Datasets [52]. RDD's are defined as *Read Only fault-tolerant, parallel data structures*[52]. RDD support two kinds of operations *transformations* and *actions*. RDD can store results in memory and can apply simple transformation such as *join* or *map*. Due to the immutable nature of RDD's each transformation results in the creation of new RDD. Spark creates direct acyclic graph of these transformations. Once the RDD's are defined through transformations, actions can be applied on them. Actions are operation that returns a value(count,collect and save). These actions include functions like *count*, *foreach*, *countbykey* and *first* In Spark RDD's can be stored in disk by calling *persist*. Since it is just an RDD it can be queried via SQL Interface or machine learning algorithms etc.

2.3.2 Spark Streaming

Spark Streaming is part of the spark stack and enables processing of live stream of data. After capturing data for a predefined interval, batches are created on which data manipulation operations such as *map*, *reduce* or *join* are performed. Since Spark Streaming is built on top of Spark users can apply machine learning algorithms using *MLLib* and graph processing operations can be performed using *GraphX*.

Once the data is transformed or manipulated it can be stored in database or file system. Spark provides an API for processing live continuous stream of data called *DStream*. DStream is represented internally as Resilient Distributed Datasets and lets user manipulate them through a series of operation. Once the operations are performed the results are returned in chunks. Dstream provides both stateless and stateful operators. Stateless operators such as *map* or *join* are those which act independently on each interval. Operators can be stateful with windowing over several batch intervals.

2.4 Monitoring Network System with MTD

The main goal of this thesis is to monitor network systems for violations described using compositional Modular Timing Diagrams. This covers various security vulnerabilities occurred due to *user authentication* and *dos attacks*. Apache Spark provides the distributed framework necessary to utilise the compositional nature of MTD and it also contains API for live streaming of data which is required in case of network systems.

Data can be ingested from many sources like Kafka, Twitter, HDFS or TCP into Apache Spark. For monitoring network systems using MTD's the main source of live data are the TCP servers. These TCP servers analyzes all the incoming requests and classify them as *normal* or *attack*.

The high level specification provided by the network administrator are represented using our compositional MTD notation. These high level specifications represent the error conditions that may come from attack on the system. MTD notation is manually converted into a text format called MSF file described later. This MSF file is given as input to Apache Spark which parses the file and extract useful pattern/information out of it. The extracted information is then use to verify each incoming connection request and an alarm is raised if some suspicious activity is detected.

Chapter 3

Modular Timing Diagrams

In this section, the syntax and semantics of *modular timing diagrams* is discussed. Modular timings diagrams (MTD) are an extension of the regular timing diagram[1] notation. A regular timing diagram (RTD) module is defined over a finite time period and represents timing dependencies between events that occur in the time period. These RTD modules are linked together by construct to form an MTD. The constructs can denote forking, deterministic choices or iteration. An example MTD with symbols is shown in Figure 3.1.

3.1 Informal Description

MTD semantics are represented by sequences of change events. Each event sequence is described by a precondition and postcondition. As shown in Figure 3.1, a precondition is denoted by dashed rectangle that indicates the initiation sequence of an event. A post-condition is represented by solid rectangle indicating the outcome of an MTD condition. Filled node at the beginning indicates an initial precondition node represented by an empty RTD. Precondition and postcondition are linked together by a connector. Terminal nodes are subset of postnodes and are not associated with an outgoing connector. A connector can be a conjuctor or a disjunctor based on the event. Symbol $\vee$ with guards specifies a disjunctor denoting deterministic choices. Symbol $\wedge$ indicates a conjunctor and represents branching.

MTD checks begin at the initial node which can be an empty node, a precondition or a postcondition. Edges between nodes are labelled by *Guard condition.* The successors

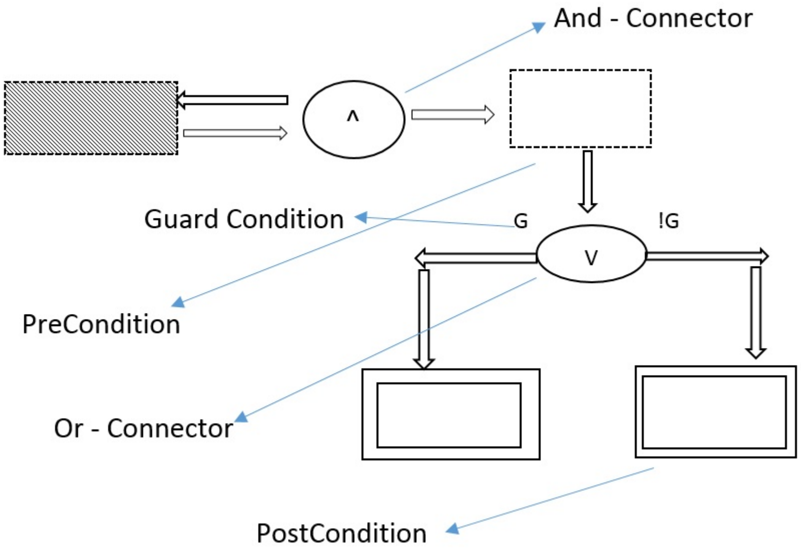


Figure 3.1: An example MTD

of precondition nodes are only validated or checked if the guard condition on the edge of precondition node holds. What successors of the current node are to be checked is determined by the connectors. An ∨ connector with guards indicates selection of a unique successor while ∧ requires that all successor nodes are checked. Iterations are allowed in MTD by looping. An example of looping shown in Figure 3.1 can be represented by an arrow from initial node back to the empty node. If a precondition is satisfied its associated postcondition node must be satisfied. All postcondition nodes used in current work are represented by RTD's with no looping conditions. Therefore each of the post condition can be represented by a deterministic finite automaton.

3.2 Modular Timing Diagrams: Syntax & Definition

An MTD is specified by a number of variables, each taking a finite set of values. An MTD models sequences of change events over time. Event MTD specifies the ordering and dependencies between different events with respect to time. An event can be denoted by a pair (n, i) where n is the variable linked to a domain D_n, and i denotes the position of an event.

If one event is sequentially dependent on another event, such dependencies are called

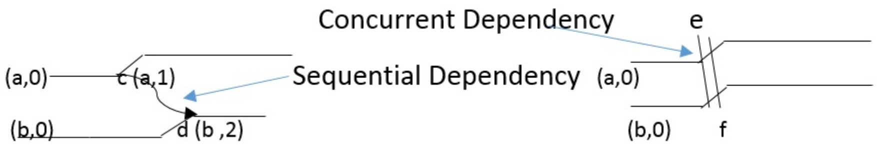


Figure 3.2: A component RTD

sequential dependencies and are represented by a curved line as shown in Figure 3.2. The transition or dependency of one event to another consumes clock cycles, denoted by a *pair(a,b)* where a occurs before b. Vertical lines in the diagram denote disjoint sets of concurrent dependencies.

A Waveform denotes changes in value of an event variable(see Figure 3.2). Each component RTD is specified by waveforms and timed dependency between points on the waveform. The values could be 0 (Low), 1 (High) and X. X denotes unspecified values. The (a, b) waveform is shown in Figure 3.2.

Modular timing diagrams are formed by composing together RTDs. The definition of point, event, RTD and MTD are discussed in the next few paragraphs.

Definition 1: Point A point (p, i) associates the variable p with a particular time instance in the computation, i and a value to the variable. For the RTD in Figure 3.2: $(a, 0)$, $(b, 0)$, $(a, 1)$ and $(b, 2)$ denote points in the waveform (a, b).

Definition 2: Regular Timing Diagrams A Regular Timing Diagram is defined by a tuple of the form (p, S, E, SD, CD) where

- p is a set of point defined over the waveform, WF.
- S represents a finite set of variable names associated with the events over a period of time. A single variable name is denoted by s.
- For every s there is a finite set of events $E(s)$ represented as $[s, 0]$, $[s, 1]$,..., $[s,n_s]$.
- SD denotes a set of sequential dependency on points in the waveform, WF. Each SD is denoted by $(p, i) \xrightarrow{[a,b)} (q, j)$, where (p, i) and (q, j) denotes two points in WF. The value a, b represents a timing dependency. Here $a \in N$ is a natural number and $b \in$ N $\cup\ \infty$ and 1<a<b. Figure 3.2 gives a simple example of sequential dependency, at points c and d.
- CD is a set of disjoint points called concurrent dependency. Figure 3.2 shows concurrent dependencies at point e and point f.

Another import term for understanding MTD is an event. An event is a change in value of a variable wrt. time. Example of events can be the change in state of a button, a user logging activity or a click on the webpage.

Definition 3: Event An event in an RTD (p, S, E, SD, CD) is defined as follows.

- E denotes an event.
- If s denotes a variable, linked to an event then *(s,0)* is an event.
- For an event (s, i), if there is a change in value from $s(i)$ to $s(j)$ then (s, j) as event.
- If (s, i) is in concurrent dependency with an event, then *(s,i)* is an event.
- For a sequential dependency $(p, i) \xrightarrow{c,c} (q, j)$ if (p, i) is an event then (q, j) is an event.

Definition 4: Modular timing Diagram A modular timing diagram is specified by a tuple (N, C, I, F) where,

- N denotes the finite set of nodes. The set consist of two types of nodes N_{pre} and N_{post} where N_{pre} denotes a set of pre nodes and N_{post} denotes a set of post nodes.
- C represents a set of connectors, $C \in (\vee, \wedge)$. $\vee$ connector $\in N \times 2^{G \times N}$. Where G is set of guards or boolean expressions. These guard are set of intersection constraints. Further, $\wedge$ connector is an element of $N \times 2^{N}$. Figure 3.1 shows an example MTD with all the components labelled.
- $I \subseteq \mathrm{N}_{pre}$ is a set of initial nodes.
- F is a set of fair nodes that defines co-Buchi acceptance condition.

3.3 Semantics

This section describes the semantics of MTD's as mentioned in the original paper [3]. The semantics of an MTD is a set of infinite sequences over a vector of variable values declared in the component RTD's. Each of the vector values represent a state. The semantics is specified using a $\forall$-automaton. The language of the automaton is the semantics for an MTD.

3.3.1 $\forall$-automaton

The deterministic $\forall$-automaton, A, can be denoted by a tuple (Σ, Q, Q_0, δ , F) where Σ is a finite set of symbols called an alphabet; Q is a finite set of states; $Q_0 \subseteq Q$, is a nonempty set of start state of the automaton; δ is a transition function specified by δ: $Q \times \Sigma \rightarrow 2^{Q^+}$, where $Q^+ = Q \cup \{\epsilon(q) : q \in Q\}$; $F \subseteq Q^\omega$ denotes a set of accepting sequences. A word is a finite sequence of letters in Σ. The set of all possible finite words over Σ is denoted by Σ^*.

A run of the $\forall$-automaton, A, on input string $\sigma \in \Sigma^\omega$ is an infinite sequence ρ from (Q$\times$ $N)^\omega$ starting with an initial state Q_0. A representation of the form (q, i) indicates the current state of the automaton is q and is reading symbol σ_i. A run is valid if for initial state, $\rho_0 = (q, 0)$, $q \in Q_0$ and, $\forall i \in N$, $\rho_i = (q_i, a_i)$ and $\rho_{i+1} = (q', a')$ where $a' = a$ and $\epsilon(q') \in \delta(q, \sigma_a)$ or $a' = a + 1$ and $q' \in \delta(q, \sigma_a)$. The input is accepting if the projection of ρ on Q is in F.

3.3.2 RTD Language

The language for, r, for a non empty RTD, is specified by a DFA, (S, ζ, SD, CD). The language for the automaton is a set of finite strings z in, Σ^* that satisfy the following condition. For each string there is a *locator* function, which determines the position of the events in the string. The *locator* function for z is specified by $\lambda_z : \zeta \rightarrow [0..|z| - 1]$ such that

- The value of each event in ζ can be located in z and has a value consistent with that in r. If $\lambda_z(s, i)$ has a value p then value of s at the p position on z, $z_p(s) = v(s, i)$.
- Let value of $\lambda_z(s, i) = k$ and $\lambda_z(s, i + 1) = l$, then for every j that lies in $[k, l)$, the value of s at jth position of z, $z_j(s) = v(s, i)$.
- For each sequential dependency specified as, $(s, i) \xrightarrow{c} (t, j)$, where c denotes the timing constraint of the form $(clock, [a, b))$. The number of events between $\lambda_z(s, i)$ and $\lambda_z(t, j)$ is in $[a, b)$.
- For each pair of events, (s, i) and (t, j) in concurrent dependency $cd \in CD$, $\lambda_z(s, i)$ $= \lambda_z(t, j)$.

3.3.3 MTD Language

An MTD, T composed of RTD's $\{r_i\}$ is denoted by a tuple (N, C, I, F). The $\forall$-automaton, B_T for T is defined as follows: For RTD r_i, let B_i =(Σ, Q_i, $\{q_0\}$, δ_i, F_i) where Q_i finite set of states; $\{q_0\}$ is a nonempty set of initial state; F_i is the set of final states. The set of states for $\forall$-automaton, B_T, is $(\cup_i Q_i) \cup \{t_i \mid i \in \mathrm{N}_{term}\}$. A transition of B_T includes the transitions of each B_i along with new transitions in the given order.

- If r_i denotes a terminal post node, then for each $q \in F_i$ and $a \in \Sigma$, add $\delta(q, a) = \{\epsilon(t_i)\}$, add $\delta(t_i, a) = \{(t_i)\}$. This represents a transition from final state of terminal nodes to the state that accepts any subsequent set of values.
- If r_i $\wedge$-connected to $r_j, ..., r_k$ then for each $q \in F_i$ and $a \in \Sigma$, add $\delta(q, a) = \{\epsilon(q_i^0), ..., \epsilon(q_k^0)\}$. This represents forking, and every node is accepting.
- If r_i is $\vee$-connected to guards $g_j, ..., g_k$ to RTD's $r_j, ..., r_k$ then for each $q \in F_i$ and $a \in \Sigma$, add $\delta(q, a) = \{\epsilon(q_l^0)\}$, where $g_l(a)$ is the unique guard that holds for a. This denotes deterministic choices.

The acceptance condition ensures that any infinite path should get stuck in prenodes or should be in infinitely often accepting states for post nodes. Thus whenever a run enters nodes of MTD, it must either satisfy all the pattern in the post nodes, or it should exit at the prenode.

Chapter 4

Security Vulnerabilities

Security vulnerabilities are network communication flaws that may result in sharing secure information or in denying users access to information which they must be able to access. Our testing dataset contains attacks that can be classified into four broad categories wide enough to encompass the major attacks. We describe them below.

- Denial of Service[29] (DoS) is an attack where the attacker floods the server with requests affecting its performance. Examples include Syn flood[37] and teardrop attacks [30]. A DoS attack on a system, called **land**[21], is when the attacker sends spoofed SYN packets with source and destination having the same address.
- Remote to Local (R2L) attacks concerns unauthorized access from a remote machine. Eg. password guessing[20].
- User to Local (U2L) attacks involve unauthorized access to local superuser privileges. The attacker exploits the vulnerability to gain access to the root account despite having only normal user privileges. Eg. buffer overflow attacks[18].
- Probing[54] is gathering information about a network system to breach its security. Eg. portscanning.

Some of the security vulnerabilities that cause modern data breaches and their specification are highlighted below.

4.1 User Authentication

Organisations need to know the identity of a user before allowing them access to the system. This prevents misuse of data, forging emails and keeps the system secure. Before logging into the system the user is requested to enter his credentials. This process is termed authentication.

An attack on this kind of system generally falls under *R2L* where a remote machine tries to gain access to a forbidden resource.

To verify that the system's authentication is not violated we represent the user authentication process using our MTD notation shown in Figure 4.1. Authentication process on the server side can be represented as a sequence of events in MTD and the outcome of logging after verification is a post condition in MTD. MTD can also represent two-way verification process by synchronizing the verification process at each level and linking it through an MTD node. We present the steps of authentication/verification process but not the details of cryptographic calculations since their verification is beyond the scope of this work. To illustrate our work, a detailed MTD for user authentication is given in Figure 4.1.

There are different ways to identify each kind of attack. *DoS* attack can be identified using the number of requests from a given host. Some probe attacks can be identified based on number of port accessed in a particular time period. However attacks on user authentication, such as *R2L*, are mostly content based [45]. To detect these attacks, content features such as the number of failed login attacks are used to look for any suspicious behaviour in the data field of the packets. For experiments, our dataset contains attacks marked by the type of *R2L* attack. We used those attacks as error conditions and generated MTD's from them for user authentication. Our experiments are further discussed in the next chapter.

A specification for user authentication can be described as:

- Webpage displays a form for user login, in Figure 4.1 denoted by *page*.
- User enters the login id and password.
- Database *loginDb* contains a database of all registered users, all incoming login request with valid id and password is matched with .
- If the password is valid, user is given access to the system.

- If invalid password, user is asked thrice to enter the password, failure to enter correct password after three trials results in an alarm and admin is notified.

The main entities involved in the user authentication process is the user account, a webpage to display the login, a database of registered users and a server that directs the request to the database. The user account represented by *user* are identified by two variables, *id* and *pwd*. The webpage provides a view to the system where the user enters details. The webpage is handled by two variables *cmd* and *detail*. Server consists of two variables *request* and *db*. These variables ensures that each login request is directed to the database for verification. In the current case the database that maintains the list of users is represented by *loginDb*.

We have specified the following features for verification: the basic user login with valid username and password; the retry password request; specific account access for a particular user. The retry password feature enables the user to reenter password if the password in the database does not match with the valid user account. The user can retry entering the password three times. Failure to enter correct password after the three trials results in blocking the requesting IP to further access the network. In the current example the blocking period is 20 sec represented by *server.block*. The administrator is notified of the suspicious activity by the variable *admin.notify*.

MTD diagrams are read from left to right. The $\wedge$ connector at the start indicates that the event has to be validated at any point along the computation provided by a left arrow pointing to the initial node. The precondition node of MTD presents a webpage to the user represented by *page* and *user* enters the required details to access the account. A clock represented by *clk* represents the time frame at which the specific event occurs. It can be seen from Figure 4.1 that verification requests starts just after the *server.request* variable is set. The database consist of all users registered with their encrypted passwords and user name.

The server verifies the details with the database. If the details entered by the user matches the one in the system the user is given student access. The access level varies according to the user id - a professor access or a student access or an administrative access. Although the login page is the same for all users, what portion of the system is accessible to the user is determined based on credentials. In Figure 4.1 the node after the execution is a postcondition node. The exclusive-or connector above the condition ensures only one active state at a given time. Also it can be seen in the postcondition that *server.db.cmd* variables ensures that user can enter the password at most thrice but after that the *server.request* variable is set to false.

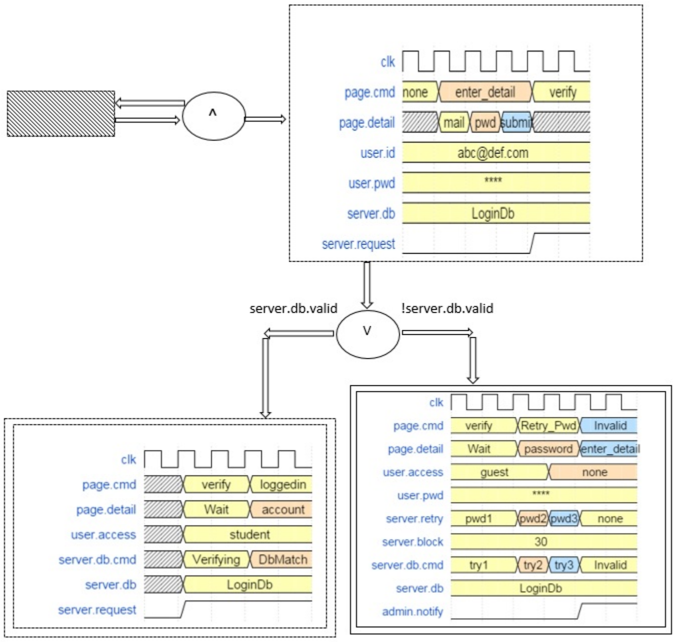


Figure 4.1: MTD of User Login - Password verification

If the username and password do not match then the user is asked to re-enter the password in the error message displayed. The postcondition verifies the username and password and passes to the next step of IP verification. A $\vee$ connector indicates the selection of a unique successor while in case of $\wedge$ all successor nodes are checked. The user is given a limited number of trials set by *server.retry*. If these trials end in failure then the user is asked to register and the account is temporarily blocked for a short period of time specified by *server.db.cmd*.

Once the user details are verified the next task is to verify the IP address as shown in Figure 4.2. Such location specific monitoring may be required in financial service companies, for instance. The main task of IP verification is to prevent suspicious account activity. This usually occurs when some malware is installed in the system resulting in remote access from a false location. If the user is logged in from a location, say *L1*, at a point of time and from a far-away location L_2 after a short period, then the activity is considered suspicious. In such a case, the user is not given access to the system despite providing correct login details. This verification is done by perusing through the last few login activities of the user to find a match with the user's current location. On failure, the user is blocked with a message saying suspicious activity detected.

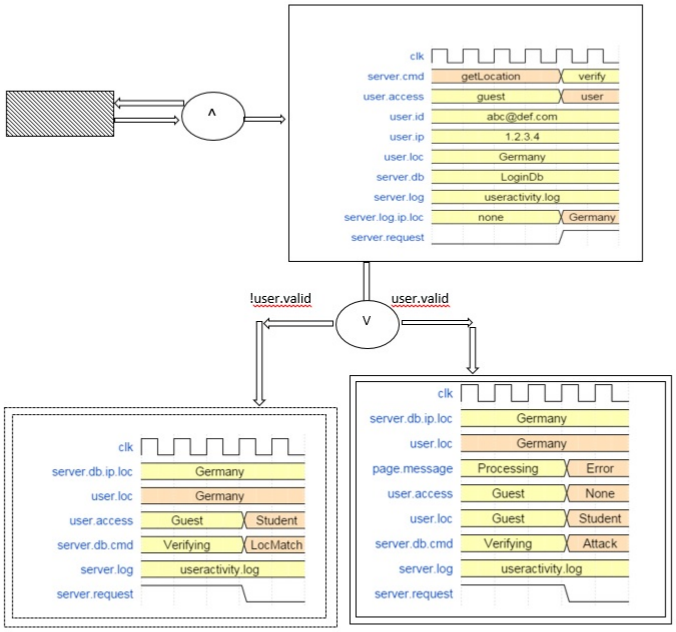


Figure 4.2: MTD of User login - IP Verification

In the example shown in Figure 4.2 we assume that the user is located in Germany denoted by variable *user.loc* and all of their requests are coming from Germany. In the preprocessing phase, the *loc* variable provides the location of the request. Initially the server retrieves the *location* of the user and then compares it with their account's previous locations obtained from the log file, *useractivity.log*. The MTD connector verifies the condition that the variable *user.valid* holds. *user.valid* compares the locations to provide access to the user. In our example, the postcondition indicates that a suspicious activity is detected since the account login is requested from *Germany* while the last few accesses were from *India*.

4.2 Communication Security

Communication security is ensuring secure communication by preventing a third party from listening to the communication. Hiding the parties in communication is one of the ways to achieve this. Attacks on the communication link usually belong to the class of attacks called *probing*. The network administrator provides error conditions that might lead to attack on the system. These network conditions are represented using MTD. User can make use of the distributed nature of MTD to represent them separately for each machine. MTD can communicate with the server to check each incoming connection. During data transfer between two parties if the data is passing through an unregistered node, this might be a attack on system and the framework notifies the network administrator about it.

4.3 Network Access Control

Network Access Control (NAC) manages the access to a resource on a sever. Access control restricts the data that each registered user can access. NAC also involves the use of various software to restrict access such as firewall and spyware detection tools.

Attacks on a system with NAC can be classified as above into R2L or L2R. MTD checks can be enforced at the initial verification step to limit the access of resources to a particular person. This will ensure a constant monitoring of not only the network systems but also the software on it.

4.4 Email Filtering

Email is critical to any organisation. Emails can serve as delivery system for spyware, worms and viruses. Email is sometimes used as a tool for DoS attack.

Email bomb[8] is a form of DoS attack where attacker sends large number of emails in order to overload the server where the email is hosted. We show that our MTD framework is capable of detecting email bombs. MTD constantly monitors emails coming from different sources by distributing the specifications across different machines. Specifications contain sender's address, destination address along with the mail domains. Any form of email bombing will result in overloading of server by mails.

Email monitoring happens at two levels. The lowest level is the connection level where the senders Ip address and domain are checked. Checks are provided to make sure that filtered mails adhere to RFC standard. When an unwanted probe is detected system raises an alarm. In the content level verification the content of the mail is verified. Attachments are examined to check if they are virus free and the mail doesn't contain suspicious content.

In this work, we are mostly concerned with the connection level verification of emails. Our dataset contains connection string marked with attack and normal. Error conditions

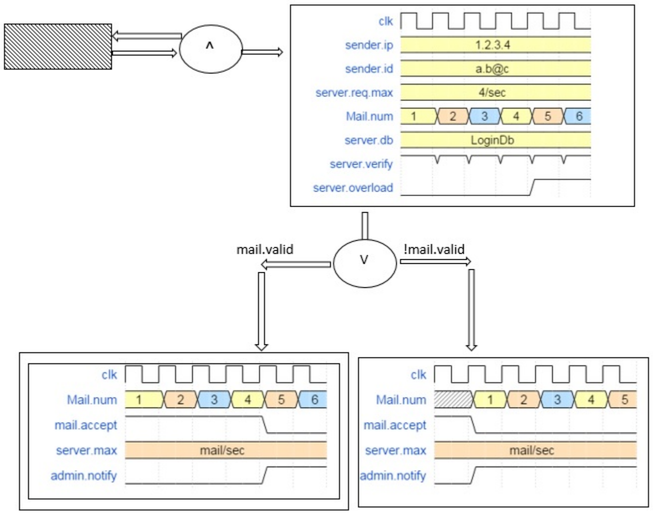


Figure 4.3: MTD of Email Filtering

are generated from our testing dataset, discussed later with connection strings marked with *DoS*. These error conditions are further enhanced considering security for financial institutions, and an MTD shown in Figure 4.3 is generated from it.

As soon as the mail enters the system it is verified to see if it is coming from a registered domain. Since the mail is not stored unless verified another variable called *server* starts the verification process. Server checks the database to verify if the mail adheres to RFC standards, and is coming from registered user. Server replies back with a command to indicate weather the mail is valid or not. If the guard condition denoted by *mail.valid* is true the mail is accepted in the system. If the domain is not valid then the mail is discarded and network administrator is notified, denoted by *network.notfiy*. Variable *server.request.max* denotes the maximum number of request a server can handle at a particular instance of time. This is just a error condition and can very according to the specification. To keep the count of number of request the MTD maintains a variable *mail.num*. If the number of mails from a particular user exceeds the *server.req.max* limit, this might be a case of *email bombing*, so the administrator is notified about it. In Figure 4.3 as soon the user receives more then four mails from the sender per second labeled by *mail.num* the variable *mail.accept*becomes $false$ and the mail is discarded.

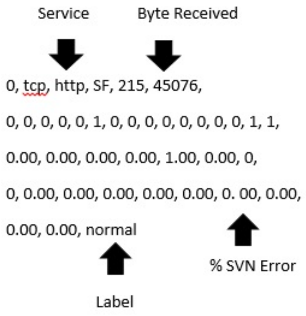
Chapter 5

Observations

To demonstrate the compositional nature of MTD, we performed various experiments on a cluster computing framework called Apache Spark[53]. One of the main reasons for choosing Apache Spark was its performance in handling large amount of real time data.

5.1 KDD dataset

For testing purposes, we used a popular intrusion detection dataset by KDD [46]. The dataset is captured from DARPA[32] 98 intrusion detection system evaluation program. DARPA is about 4 gigabytes of compressed raw TCP dump data from seven weeks of network traffic. The data consists of around 5 million connections, each of around 100 bytes. A connection represents a sequence of TCP packets from a source IP to target IP



```
0, tcp, http, SF, 215, 45076,
0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
0.00, 0.00, 0.00, 0.00, 1.00, 0.00, 0,
0, 0.00, 0.00, 0.00, 0.00, 0.00, 0. 00, 0.00,
0.00, 0.00, normal
```

Figure 5.1: Data Format

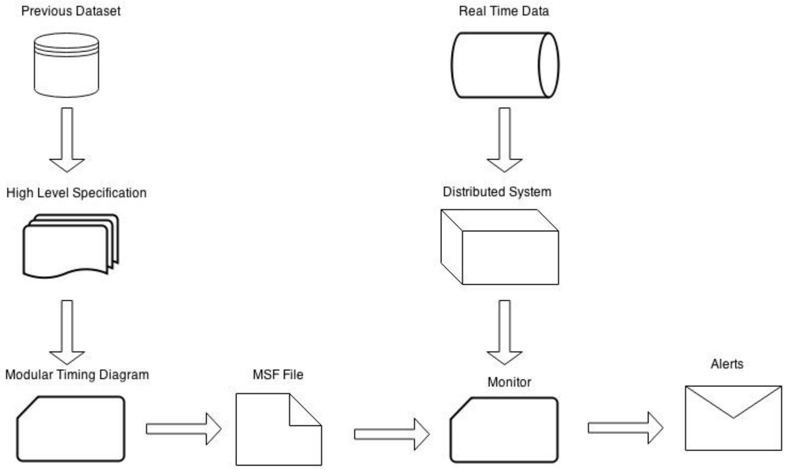


Figure 5.2: Distributed Monitoring System

address, number of bytes sent, types of connection etc. Each row in the dataset is marked with either normal or attack.

A row from the dataset is shown in Figure 5.1. The main fields are labeled. The connection shown in Figure 5.1 denotes a TCP connection with 215 bytes sent and 45076 bytes received. We see that the last field specifies the type of request - *type of attack* or *normal*. This field is used to separate normal connections from suspicious ones marked by a type of attack. During the monitoring process, a request to the system or network is observed by monitors immediately.

The KDD dataset is analyzed and all requests marked by a type of *attack* are extracted. The attacks belong to the four categories of attacks mentioned above. For our system, we extracted all the attacks marked *DoS* and *R2L*. The IP addresses associated with these attacks are termed blocked. The connection strings associated with the attacks are then explored. Error conditions are generated from these strings which in turn generate the specifications and hence the MTDs. This dataset is only used to test our approach and should not be used for building real time systems as the dataset reflects traffic pattern more then a decade ago and quite a few newer types of attack have come up since then.

However, in practice, the network administrators provide their own specifications. Figure 5.2 above summarizes the key step in the distributed monitoring process. Initially the system administrator generates high level specification from previous data. This can be log files or datasets of previous user activity. These high level specification contains error conditions associated with the network. Examples of error conditions include blocked IP addresses, the maximum number of requests per time unit a user is allowed, the number of incorrect password attempts etc.

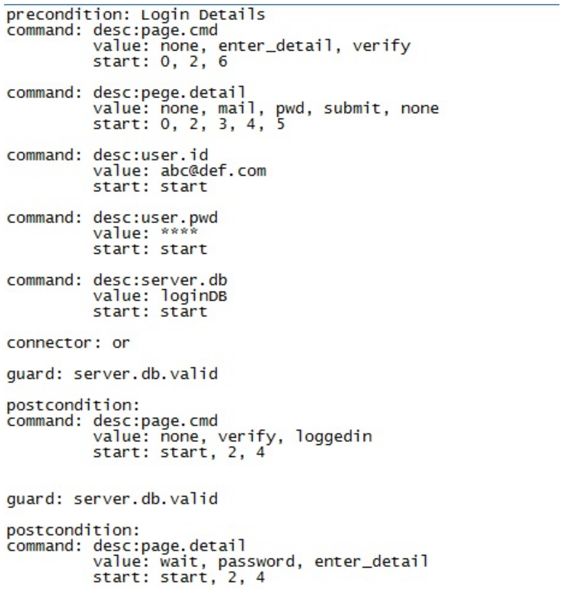
These high level specification are expressed with our MTD notation. We make use of the compositional nature of MTDs to translate single monolithic specifications into distributed specifications of several parts. Each part in itself is an MTD describing the occurrence of these error conditions at a particular location in the network.

5.2 MSF File

To make the system compatible with Spark and utilize MTD's compositional nature, the MTD specifications are translated into a text file. The text file is referred as MTD specification file (MSF). Currently, this file is generated manually from the MTDs but a tool for converting diagram to text, written specifically for MTD could be useful for this task. A sample format of MSF is shown in Figure 5.3. The MSF file corresponds to one of the attack marked *R2L* in the KDD dataset, which is represented by the MTD in Figure 4.1.

The generated MSF file is used as a specification input to Apache Spark on the given dataset. Data is ingested in Apche Spark using the streaming API's. Spark receives stream of TCP communication data which is then processed by the spark engine. Each connection is then classified as attack or normal based on the specifications provided. Apache Spark's streaming APIs make it easy to process the live stream of data. The tool analyses the real-time data looking for irregular patterns or fluctuation that might suggest a security breach. As soon as an attack is inferred, the administrator is notified of the security breaches.

MTD specification files contain several keywords.*Pre* represents a precondition in MTD and *post* represents a postcondition. The connector to be used is indicated in the MSF by the keyword *Connector*. The *command* tag indicates the messages exchanged in the system. To handle the events that occur at a certain point in time, we associate a time variable with each command. This variable is assigned either discrete values or indicators such as 'mid', 'end', 'start' etc. In the given MSF, the messages exchanged are *verfiylogin*, *errormessage* and *verifylocation*. The postcondition, which is determined by the guard conditions, is the final state of the system. In our case, the guard conditions are denoted by a variable called *server.db.valid*. When a new user enters the system, the generated MSF is parsed to extract essential information including password, mailid and IPaddress along with the messages that are associated with each user. Here, user denotes not only a person entering the system but also various entities such as a database, a system and a server.



```
precondition: Login Details
command: desc:page.cmd
         value: none, enter_detail, verify
         start: 0, 2, 6

command: desc:pege.detail
         value: none, mail, pwd, submit, none
         start: 0, 2, 3, 4, 5

command: desc:user.id
         value: abc@def.com
         start: start

command: desc:user.pwd
         value: ****
         start: start

command: desc:server.db
         value: loginDB
         start: start

connector: or

guard: server.db.valid

postcondition:
command: desc:page.cmd
         value: none, verify, loggedin
         start: start, 2, 4


guard: server.db.valid

postcondition:
command: desc:page.detail
         value: wait, password, enter_detail
         start: start, 2, 4
```

Figure 5.3: MSF file

5.3 Clustering and Live Streaming of Data

All the experiments were performed on Amazon Elastic Compute Cloud(Ec2). Five machines were used to work as a cluster. The cluster contains all five *m*1.*small* instances. One of the nodes is the master node responsible for scheduling tasks to the rest four slave nodes. The operating system was Ubuntu Server 14.04 LTS.

Spark provides API to access data from different data sources such as HDFS [10], Cassandra [31] and Hive [48]. In our current system we store the data in Hadoop Distributed File System(HDFS).

In the Hadoop Distributed File System, the master node is called *namenode*. All other nodes are called *datanodes* and all data processing are performed on them. The MSF files generated from KDD dataset are copied to HDFS. These MSF files are then loaded to Spark as an RDD. The file is then parsed using the Spark API's available. Due to the compositional nature of MTD, the generated MSF files can be distributed across different slave machines for faster processing. To avoid loading data from disk every time, Spark caches the data in memory.

A graphical view of our cluster is shown in Figure 5.4. To monitor network on a cluster of machines we followed the algorithm described below:

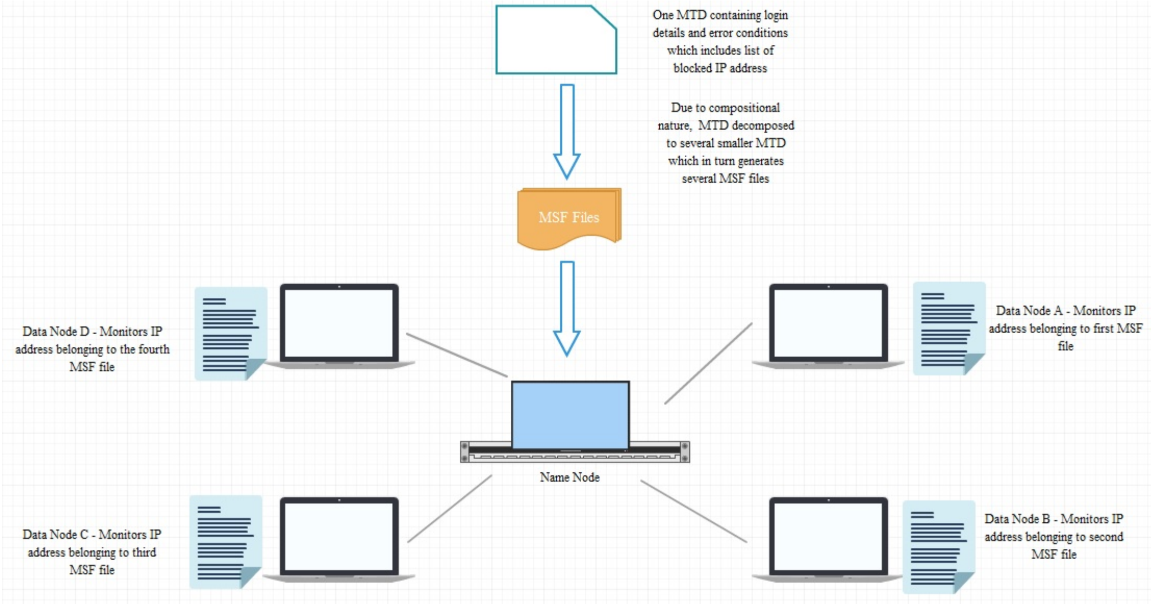

One MTD containing login details and error conditions which includes list of blocked IP address
Due to compositional nature, MTD decomposed to several smaller MTD which in turn generates several MSF files
MSF Files
Data Node D - Monitors IP address belonging to the fourth MSF file
Data Node A - Monitors IP address belonging to first MSF file
Name Node
Data Node C - Monitors IP address belonging to third MSF file
Data Node B - Monitors IP address belonging to second MSF file

Figure 5.4: System Design

- Generate a large MTD with blocked IP addresses and specifications for monitoring the network system.
- Generate smaller sub-specifications from the MTD. This is possible owing to the compositional nature of MTD.
- Decide the range of IP addresses for each data node based on the previous datasets.
- Each smaller MTD is then converted to an MSF file: in our case, four MSF files containing different ranges of IP addresses are generated.
- Create a Spark cluster, or a cluster of four data nodes and one name node in our case.
- Copy MSF file to the corresponding datanode.
- Channel the new requests to the concerned data node for verification based on the IP address of the incoming request.

The range of IP addresses at each data node is decided based on the test dataset. Each incoming request is directed to the data node associated with its IP address. Once the request is directed, the MSF file is parsed to match the incoming IP address with the list of blocked IP addresses at that node. In the case of a match, an alarm is raised to notify the administrator of the possible attack.

Since network systems are real-time, Spark streaming API's can be used to process the real time data. The real-time data is ingested using the TCP connection. In our test case, the KDD dataset is used and passed to the Spark cluster. The incoming traffic is monitored based on the RDD generated from MSF file.

Our framework is able to detect unregistered IP addresses and blocked IP address from the requesting connections. We are also able to monitor cases where the number of requests to a particular server from single machine exceeds a certain threshold and thus we could prevent any basic *dos* attack. We also found that the MSF files after parsing, contains the error conditions associated with the four types of attacks and, hence, we conclude that the method of generating MTDs from specifications is effective. Furthermore, our experiments were conducted on a small cluster; we plan to perform them on multi-user network systems.

Chapter 6

Conclusion and Future Work

In this work we show how to monitor network systems using compositional MTD notation. The main advantage of MTD is its clear graphical interface and its compositional nature making it easier to represent system specifications of network systems. More specifically, we show how to use modular reasoning with MTD properties along with asynchronous compositional reasoning rule to monitor the network. From the error conditions, we generate specifications that describe the attack. We show that converting these diagrams to text, allows us to effectively check the user information for forbidden conditions and detect attacks. To evaluate our approach, we performed our experiments using a distributed framework called Apache Spark. Our MTD components are designed to monitor events and messages across the network nodes and to notify network administrator of the occurrence of a specific error event.

For future work, we plan to test our monitoring approach on an IP based network system. We aim to extend the RTDT [1] tools to MTD to design a graphical editor for them. A tool to generate MSF file from MTD is the next step in our MTD notation. We also want to consider how to combine specifications into an optimal representation. Another important future work could be to use machine learning algorithms to predict abnormal behaviour.

References

[1] Nina Amla, E Allen Emerson, Robert P Kurshan, and Kedar Namjoshi. Rtdt: A front-end for efficient model checking of synchronous timing diagrams. In *Computer Aided Verification*, pages 387–390. Springer, 2001.

[2] Nina Amla, E Allen Emerson, and Kedar S Namjoshi. Efficient decompositional model checking for regular timing diagrams. In *Correct Hardware Design and Verification Methods*, pages 67–81. Springer, 1999.

[3] Nina Amla, E Allen Emerson, Kedar S Namjoshi, and Richard J Trefler. Visual specifications for modular reasoning about asynchronous systems. In *Formal Techniques for Networked and Distributed SytemsFORTE 2002*, pages 226–242. Springer, 2002.

[4] Martin Leucker Andreas Baeur and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4), 2011.

[5] Ion Androutsopoulos, John Koutsias, Konstantinos V Chandrinos, and Constantine D Spyropoulos. An experimental comparison of naive bayesian and keyword-based anti-spam filtering with personal e-mail messages. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 160–167. ACM, 2000.

[6] Christel Baier and Joost-Pieter Katoen. Principles of model checking. *The MIT Press, Cambridge, MA*, 2008.

[7] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[8] Tim Bass, Alfredo Freyre, David Gruber, and Glenn Watt. E-mail bombs and countermeasures: cyber attacks on availability and brand integrity. *Network, IEEE*, 12(2):10–17, 1998.

[9] Gaetano Borriello. Formalized timing diagrams. In *Design Automation, 1992. Proceedings.,[3rd] European Conference on*, pages 372–377. IEEE, 1992.

[10] Dhruba Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[11] Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, and Rich Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25(2-3):167–198, 2004.

[12] Jake D Brutlag. Aberrant behavior detection in time series for network service monitoring. In *InProc. of the 14th Systems Administration Conference*, page 13.

[13] R. Jhala C. Killian, J. W. Anderson. Life, death, and the critical transition: Finding liveness bugs in systems code. *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation(NSDI)*, 2007.

[14] Ming Chai and Bernd-Holger Schlingloff. Monitoring systems with extended live sequence charts. In *Runtime Verification*, pages 48–63. Springer, 2014.

[15] Chris Chatfield. The holt-winters forecasting procedure. *Applied Statistics*, pages 264–279, 1978.

[16] Hana Chockler and Kathi Fisler. Temporal modalities for concisely capturing timing diagrams. In *Correct Hardware Design and Verification Methods*, pages 176–190. Springer, 2005.

[17] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[18] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security*, volume 98, pages 63–78, 1998.

[19] Werner Damm and David Harel. *LSCs: Breathing life into message sequence charts.* Springer, 1999.

[20] Yun Ding and Patrick Horster. Undetectable on-line password guessing attacks. *ACM SIGOPS Operating Systems Review*, 29(4):77–86, 1995.

[21] Scott Dubal, Douglas Boom, Patrick Connor, and Mark Montecalvo. Detecting a network attack, December 18 2002. US Patent App. 10/323,985.

[22] Ylies Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In *Runtime Verification*, pages 40–59. Springer, 2009.

[23] Kyle Geiger. *inside ODBC*. Microsoft Press, 1995.

[24] Yu Gu, Andrew McCallum, and Don Towsley. Detecting anomalies in network traffic using maximum entropy estimation. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 32–32. USENIX Association, 2005.

[25] Graham Hamilton, Rick Cattell, Maydene Fisher, et al. *JDBC Database Access with Java*, volume 7. Addison Wesley, 1997.

[26] Klaus Havelund and Grigore Roşu. Monitoring java programs with java pathexplorer. *Electronic Notes in Theoretical Computer Science*, 55(2):200–217, 2001.

[27] Ming-Yuh Huang, Robert J Jasper, and Thomas M Wicks. A large scale distributed intrusion detection framework based on attack strategy analysis. *Computer Networks*, 31(23):2465–2475, 1999.

[28] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P Berman, and Phil Maechling. Scientific workflow applications on amazon ec2. In *E-Science Workshops, 2009 5th IEEE International Conference on*, pages 59–66. IEEE, 2009.

[29] Peyman Kabiri and Ali A Ghorbani. Research on intrusion detection and response: A survey. *IJ Network Security*, 1(2):84–102, 2005.

[30] Byoung-Doo Kang, Jae-Won Lee, Jong-Ho Kim, O-Hwa Kwon, Chi-Young Seong, and Sang-Kyoon Kim. An intrusion detection system using principal component analysis and time delay neural network. In *Enterprise networking and Computing in Healthcare Industry, 2005. HEALTHCOM 2005. Proceedings of 7th International Workshop on*, pages 442–445. IEEE, 2005.

[31] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[32] Richard P Lippmann, David J Fried, Isaac Graf, Joshua W Haines, Kristopher R Kendall, David McClung, Dan Weber, Seth E Webster, Dan Wyschogrod, Robert K

Cunningham, et al. Evaluating intrusion detection systems: The 1998 darpa off-line intrusion detection evaluation. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 12–26. IEEE, 2000.

[33] Leonardo Mariani and Mauro Pezze. A technique for verifying component-based software. *Electronic Notes in Theoretical Computer Science*, 116:17–30, 2005.

[34] Tony A Meyer and Brendon Whateley. Spambayes: Effective open-source, bayesian based, email classification system. In *CEAS*. Citeseer, 2004.

[35] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. *NSDI*, 4, 2004.

[36] Kedar S Namjoshi and Richard J Trefler. On the completeness of compositional reasoning. In *Computer Aided Verification*, pages 139–153. Springer, 2000.

[37] Ross Oliver and Tech Mavens. Countering syn flood denial-of-service attacks. In *Invited Talks of USENIX Security Symposium*, 2001.

[38] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[39] Dennis K Peters and David Lorge Parnas. Requirements-based monitors for real-time systems. *Software Engineering, IEEE Transactions on*, 28(2):146–158, 2002.

[40] Lee Pike, Sebastian Niller, and Nis Wegmann. Runtime verification for ultra-critical systems. In *Runtime Verification*, pages 310–324. Springer, 2012.

[41] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

[42] R Sekar, Ajay Gupta, James Frullo, Tushar Shanbhag, Abhishek Tiwari, Henglin Yang, and Sheng Zhou. Specification-based anomaly detection: a new approach for detecting network intrusions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 265–274. ACM, 2002.

[43] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[44] Margaret H Smith, Gerard J Holzmann, and Kousha Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 14–22. IEEE, 2001.

[45] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali-A Ghorbani. A detailed analysis of the kdd cup 99 data set. In *Proceedings of the Second IEEE Symposium on Computational Intelligence for Security and Defence Applications 2009*, 2009.

[46] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali-A Ghorbani. A detailed analysis of the kdd cup 99 data set. In *Proceedings of the Second IEEE Symposium on Computational Intelligence for Security and Defence Applications 2009*, 2009.

[47] Wolfgang Thomas. Automata on infinite objects. *Handbook of theoretical computer science*, 2, 1990.

[48] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[49] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Detecting implied scenarios in message sequence chart specifications. *ACM SIGSOFT Software Engineering Notes*, 26(5):74–82, 2001.

[50] National Threat Assessment Ctr US Secret Service, United States of America, CERT® Division of the Software Engineering Institute, United States of America, CSO Magazine, and United States of America. Us cybercrime: Rising risks, reduced readiness key findings from the 2014 us state of cybercrime survey. 2014.

[51] Cristina M Wilcox and Brian C Williams. Runtime verification of stochastic, faulty systems. In *Runtime Verification*, pages 452–459. Springer, 2010.

[52] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[53] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[54] GR Zargar and P Kabiri. Identification of effective network features for probing attack detection. In *Networked Digital Technologies, 2009. NDT'09. First International Conference on*, pages 392–397. IEEE, 2009.