

A Modular Notation for Monitoring Network Systems

Prashant Raghav and Richard Trefler

David R. Cheriton School of Computer Science, University of Waterloo,
Waterloo, Canada
{praghav, trefler}@uwaterloo.ca

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. MTDs are a *component-oriented* and *compositional* notation. We take advantage of these features of MTDs and point out that, in many cases, a global MTD specification describing behaviors of several system components 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 MTDs in describing several network monitoring conditions related to network security.

Keywords: Distributed Systems, Network Security, Data Mining, Formal Methods, Apache Spark

1 Introduction

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.

Providing a mathematically precise notation for describing essential aspects of distributed systems - in this case, system requirements - may be a complex task. Temporal logic[8] 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.

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.

Our 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. When the mtd's are decomposed into component monitors, all the monitors are described by deterministic automata.

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[25] 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. Section 4 describes how user authentication can be represented and monitored using MTD.

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[20]. Yu Gu et al. [12] proposed a technique to detect anomalies in network traffic using maximum entropy estimation. Brutlag et al [6] used Holt Winter Forecasting[7] model to predict future traffic based on the history of network traffic. Bro et al [19] monitored network system by monitoring network link over intruders traffic transit. Yuh Huang and Thomas Wicks [13] gave a distributed intrusion detection framework using attack strategy analysis. Sekar et al[21] gave a specification language for network intrusion detection system which enables a strict static and dynamic checking. This work introduces the use of compositional nature of MTD's in conjunction with Apache Spark for monitoring network systems.

Our paper is divided into five sections. In Section 2 we describe the syntax and semantics of MTD. Section 3 looks at various security issues in the system and how they are monitored using MTD. Section 4 talks about our experiments where we look at how we used Spark with our framework and we conclude with Section 5.

2 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 constructs to form an MTD. The constructs can denote forking, deterministic choices or iteration. An example MTD with symbols is shown in Figure 1.

2.1 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 1, a precondition is denoted by dashed rectangle that indicates the initiation sequence of an event. A postcondition 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 conjunctive or a disjunctive based on the event. Symbol \vee with guards specifies a disjunctive denoting deterministic choices. Symbol \wedge indicates a conjunctive 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 of precondition nodes are only validated or checked if the guard condition on the edge of precondition node holds. What successors of the current

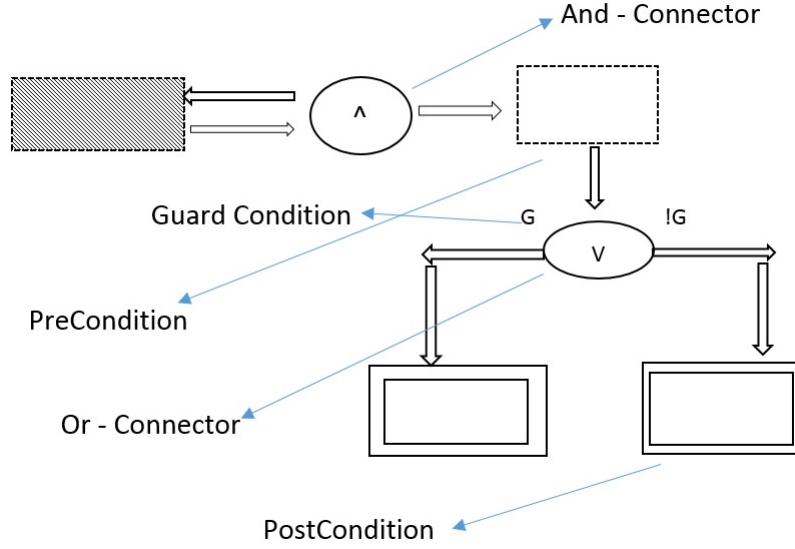


Fig. 1. An example MTD

node are to be checked is determined by the connectors. An \vee connector with guards indicates selection of a unique successor while \wedge requires that all successor nodes are checked. Iterations are allowed in MTD by looping. An example of looping shown in Figure 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.

2.2 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 *sequential dependencies* and are represented by a curved line as shown in Figure 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 2). Each component RTD is specified by waveforms and timed dependency between

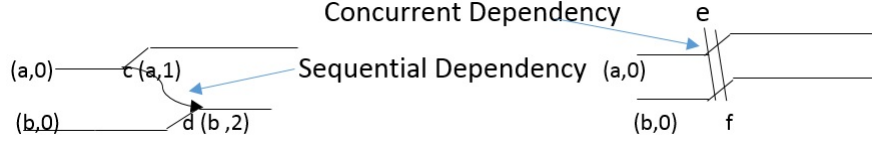


Fig. 2. A component RTD

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 2.

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

Definition 1: 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 $[s, 0], [s, 1], \dots, [s, n_s]$.
- 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 $a \in \mathbb{N}$ is a natural number and $b \in \mathbb{N} \cup \infty$ and $1 < a < b$. Figure 2 gives a simple example of sequential dependency, at points c and d .
- CD is a set of disjoint points called concurrent dependency. Figure 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 2: 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, θ) 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 3: 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 1 shows an example MTD with all the components labelled.
- $I \subseteq N_{pre}$ is a set of initial nodes.
- F is a set of fair nodes that defines co-Buchi acceptance condition.

2.3 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.

2.4 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 j th 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 $cd \in CD$, $\lambda_z(s, i) = \lambda_z(t, j)$.

2.5 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 $B_i = (\Sigma, Q_i, \{q_0\}, \delta_i, F_i)$ 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 $(\cup_i Q_i) \cup \{t_i \mid i \in N_{term}\}$. 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, \dots, r_k then for each $q \in F_i$ and $a \in \Sigma$, add $\delta(q, a) = \{\epsilon(q_i^0), \dots, \epsilon(q_k^0)\}$. This represents forking, and every node is accepting.
- If r_i is \vee -connected to guards g_j, \dots, g_k to RTD's r_j, \dots, r_k then for each $q \in F_i$ and $a \in \Sigma$, add $\delta(q, a) = \{\epsilon(q_i^0)\}$, 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.

3 Security Vulnerability

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[14] (DoS) is an attack where the attacker floods the server with requests affecting its performance. Examples include Syn flood[18] and teardrop attacks [15]. A DoS attack on a system, called **land**[11], 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[10].
- 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[9].
- Probing[26] 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.

3.1 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 3. 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 3.

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 [22]. 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 3 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 3 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 3 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.

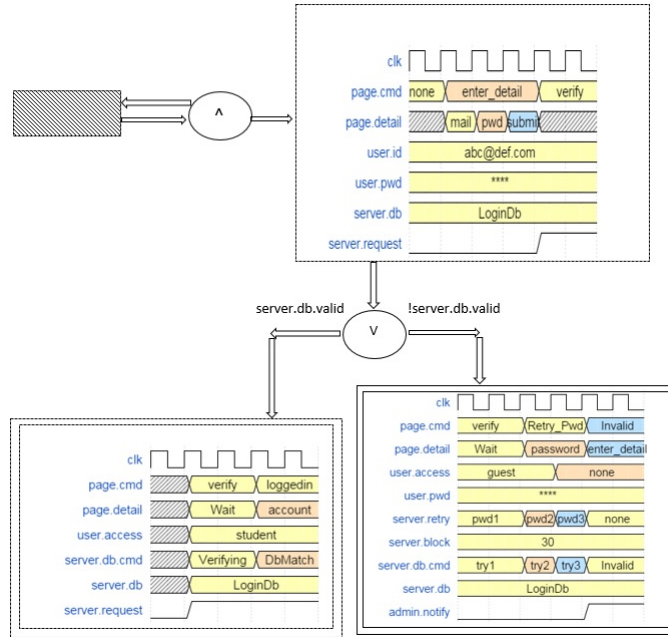


Fig. 3. MTD of User Login - Password verification

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. 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 L_1 , 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.

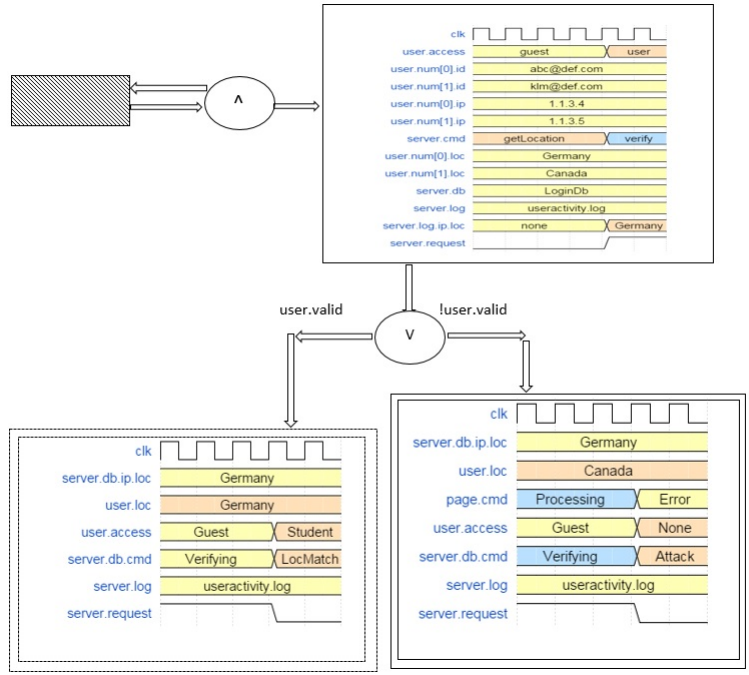


Fig. 4. MTD of User login - IP Verification

In the example shown in Figure 4 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 *Canada* while the last few accesses were from *Germany*.

3.2 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[4] 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.

Our dataset contains connection string marked with attack and normal. Error conditions are generated from our the 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 5 is generated from it.

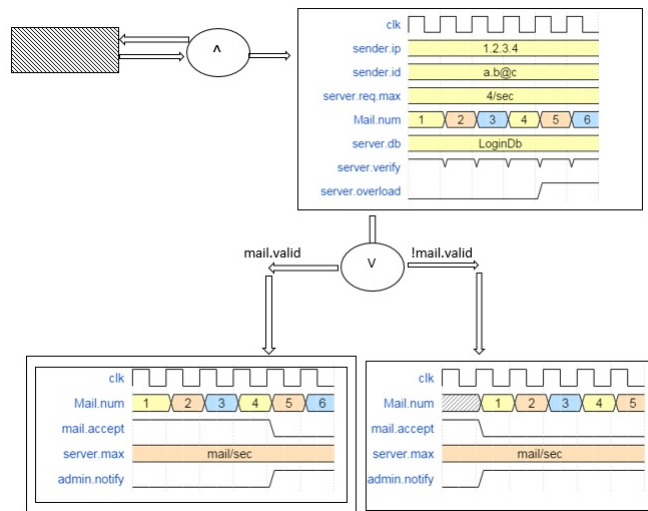


Fig. 5. MTD of Email Filtering

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.notify*. 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 vary 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 5 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.

4 Observations

To demonstrate the compositional nature of MTD, we performed various experiments on a cluster computing framework called Apache Spark[25]. One of the main reasons for choosing Apache Spark was its performance in handling large amount of real time data.

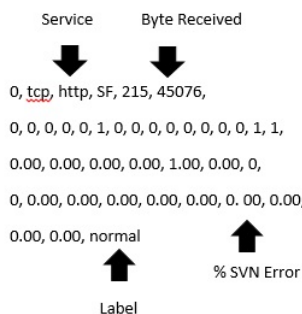


Fig. 6. Data Format

4.1 KDD dataset

For testing purposes, we used a popular intrusion detection dataset by KDD [23]. The dataset is captured from DARPA[17] 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 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 6. The main fields are labeled. The connection shown in Figure 6 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 than a decade ago and quite a few newer types of attack have come up since then.

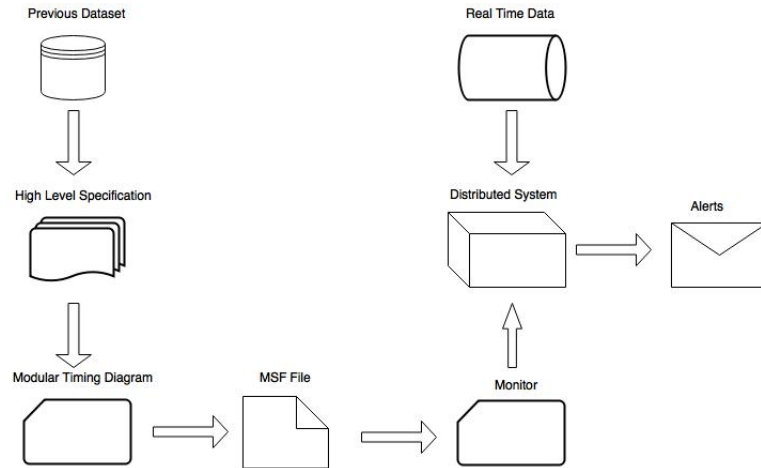


Fig. 7. Distributed Monitoring System

However, in practice, the network administrators provide their own specifications. Figure 7 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.

4.2 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 8. The MSF file corresponds to one of the attack marked *R2L* in the KDD dataset, which is represented by the MTD in Figure 3.

The generated MSF file is used as a specification input to Apache Spark on the given dataset. Data is ingested in Apache 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.

```

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

command: desc:page.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

```

Fig. 8. MSF file

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 *verfylogin*, *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 IP address 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.

4.3 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 *m1.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 [5], Cassandra [16] and Hive [24]. 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. To monitor network on a cluster of machines we followed the algorithm described below:

- 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.

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.

5 Conclusion

In this paper 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 generating MSF file from MTD is the next step in our MTD notation. We also want to consider how to combine specification 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 Treffer. Visual specifications for modular reasoning about asynchronous systems. In *Formal Techniques for Networked and Distributed Systems FORTE 2002*, pages 226–242. Springer, 2002.
4. 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.
5. Dhruba Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT* http://hadoop.apache.org/common/docs/current/hdfs_design.pdf, 2008.
6. Jake D Brutlag. Aberrant behavior detection in time series for network service monitoring. In *InProc. of the 14th Systems Administration Conference*, page 13.
7. Chris Chatfield. The holt-winters forecasting procedure. *Applied Statistics*, pages 264–279, 1978.
8. 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.
9. Crispian 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.
10. Yun Ding and Patrick Horster. Undetectable on-line password guessing attacks. *ACM SIGOPS Operating Systems Review*, 29(4):77–86, 1995.
11. Scott Dubal, Douglas Boom, Patrick Connor, and Mark Montecalvo. Detecting a network attack, December 18 2002. US Patent App. 10/323,985.
12. 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.
13. 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.
14. Peyman Kabiri and Ali A Ghorbani. Research on intrusion detection and response: A survey. *IJ Network Security*, 1(2):84–102, 2005.
15. 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.
16. Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

17. Richard P Lippmann, David J Fried, Isaac Graf, Joshua W Haines, Kristopher R Kendall, David McClung, Dan Weber, Seth E Webster, Dan Wyszogrod, 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.
18. Ross Oliver and Tech Mavens. Countering syn flood denial-of-service attacks. In *Invited Talks of USENIX Security Symposium*, 2001.
19. Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.
20. Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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.
26. 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.