RUNTIME MONITORS AS SENSORS OF SECURITY SYSTEMS

Sajjan Shiva, Ramya Dharam and Vivek Shandilya
Department of Computer Science
University of Memphis
Memphis, TN-38152, USA
email: { sshiva, rdharam, v.shandilya@memphis.edu}

ABSTRACT
Formal methods have been used to establish the idea of safety and monitorable properties. Drawing from such work, we provide here two examples of monitorable properties, affecting the security of the systems. In this work we have constructed two sensors using run-time monitors. One of them detect the anomalies in the system operations due to an internal error, while the other due to an disruption by an external agency. A formal representation of the target program is presented establishing the monitorable property used to detect the anomalies. Being able to sense the anomalies during execution is the first step in ensuring the security of a system. Constructing such sensors is the first step in constructing security systems. Such sensors have to detect malfunctions caused both by internal errors and external intrusions. In the examples discussed, the executions are monitored for violations of the identified property by monitors constructed using JavaMop tool. The programs monitored are simple versions of Java programs used in common web applications.

KEY WORDS
Safety property, Monitorable property, Monitor, Monitor Oriented Programming, Turing Recognizable, Sensors.

1 Introduction

There is an increase in the usage of on-line services for social and commercial activities. This makes the reliability of the software systems providing these on-line services increasingly important. Though careful design and testing can reduce errors in these systems, they cannot ensure complete reliability because the systems can behave unexpectedly during run time. The deviation in the observed behavior from the expected behavior of the software systems is termed as software anomaly. To detect such anomalies there is a need for run-time monitoring. Monitors facilitating it are to be constructed by identifying the monitorable properties of the software systems. [1] explored the motivations and foundations of run time monitoring. Similar works in [2, 3] explore the formal frame works for establishing the safety and monitorable properties. These monitors form the sensors for the security systems which ensure the security of the target systems.

The anomalies may be caused due to internal errors in the software system or interference by external agencies with malicious intentions. Such anomalies must be detected by the run time monitors. To construct the monitors we need to identify the monitorable property in the software system. To identify the monitorable properties we need both full access1 to its code and the specified requirement2. By accessing the code the nodes are identified following the work in [4]. Using these nodes we construct a control flow graph and identify all the execution paths in it. Using these execution paths and requirements we identify the monitorable properties. We express the identified properties using logical formalisms that can be used in the run time monitoring tools to build the monitor. There are many such tools available as discussed in [5]. In our experiment, we used a freely available tool called JavaMop, whose details are discussed in [6].

In this work we present two examples which are simplified versions of real life scenarios. The monitors we construct in this work are designed to act as the sensors in a security system implementing the defense architecture called GIDA, Game theory Inspired Defense Architecture as described in [7].

The main contributions of this work are follows.

1. We combine two prior works to construct a procedure to identify the monitorable property starting from the program code and the specified requirement.
2. We provide two examples with run time anomalies; one caused by an internal error and another by an external agency with malicious intentions. We construct monitors for both of them which act as sensors to the security system.

The section 2 describes the formal procedure constructed by adapting from the prior works [4] & [8], to identify the monitorable property in the target program. Sections 3 and 4 present examples with a run time anomaly caused by an internal error and by an external user with malicious intentions respectively. We apply the formal procedure described in section 2, to these examples to identify the moni-

---

1If we don’t have full access to the system’s code we should have our own program which concurrently run sniffing some system parameters, whose values indicating the performance of the applications we want to monitor. Then designing some control flow delineation in the program depending on the values of the observed parameter, we can construct a monitor for this sniffing program.

2Some researchers refer to this as requirement property.
torable properties and construct monitors accordingly. The section 5 describes the role of the monitors as sensors in a security system, which provide security for a target system. The section 6 discusses the related works. The section 7 presents the conclusion. 3

2 Monitorable property Identification Procedure (MIP)

In this section we describe a procedure to identify monitorable properties based on [4] and [8].

Given the target program we identify the following six statements in it: an assignment statement, an input or output statement, a procedure call statement, an procedure entry statement, an goto, break, continue or label statement, a predicate of conditional or loop statement.

These statements in the given program are represented as nodes, denoted by the symbol $s_i$. The subscript $i$ in the symbol $s_i$, represents where the statement appears in the program from the beginning. The potential control flow between these nodes corresponds to arcs. With the nodes as vertices and arcs as the edges, the program is represented as a control flow graph. This graph is a directed graph and every path (which is a sequence of nodes) in it, starting with the vertex (representing the invocation of program execution) is identified.

A program in execution is a process. When the process is observed at successively different instants of time, a node in execution could be seen. Thus, a node in control flow graph corresponds to a state in the process. The set of all valid states is denoted by $S$. The transition between states define how the process is progressing, that is, how the program is getting executed. The state at which the process is in at the instant of its invocation is called its initial state.

Formally execution is defined as an infinite sequence. By repeating the terminating state in the paths obtained we construct the infinite sequences representing the execution. Such an infinite sequence of process states, starting with an initial state and each successive states $s_i, 0 < i < \infty$ being a valid state, that is $s_i \in S, 0 \leq i < \infty$, representing an execution is given by

$$\sigma = (s_0, s_1, s_2, \ldots). \tag{1}$$

It should be noted that the order of the states in the execution need not follow the subscript order of nodes obtained from the program, as it is only decided by the flow of execution. Thus all the executions $\sigma_i, 0 < i < \infty$ are obtained. By dropping the initial elements in the sequence of these executions, many infinite sequences are created which represent a later part of an execution in the process. That is by dropping an initial state in the execution above, which may be called $\sigma_1$, another $\sigma_1$ is created, as given by

$$\sigma_1 = (s_1, s_2, \ldots). \tag{2}$$

Now the infinite set of all infinite sequences of all permutations of $s_i, s_i \in S$ (which also contains all the executions and the later parts of executions like in equation 2), is denoted by $S^\omega$ is constructed, as given by

$$S^\omega = \{(s_1, s_2, s_3, \ldots), (s_2, s_2, s_2, \ldots), \ldots, \sigma_1, \sigma_1, \ldots, \sigma_2, \sigma_3, \ldots\} \tag{3}$$

Property is defined as a set of executions.

$$P = \{\sigma_1, \sigma_2, \sigma_3, \ldots\} \tag{4}$$

A property $P \subseteq S^\omega$ is a safety property $P_{safe}$ if for every sequence $\sigma \in P$, for every prefix of $\sigma$, there exists a sequence $\beta \in S^\omega$ which when appended will make the resulting sequence to be a member of the set $P$. This means in essence that the sequences $\sigma$s contain as each element states only from $S$.

The executions belonging to the safety property are called safe executions. If $\sigma_1, \sigma_2, \sigma_3$ and $\sigma_4$ are safe executions, then

$$P_{safe} = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \ldots\} \tag{5}$$

Prefix of an execution is an initial subsequence of the execution and it represents the execution of the process from the beginning up to some finite number of states. We define the set $pref(\sigma_i)$ as a set of all the finite prefixes of the execution $\sigma_i$. For example consider the execution $\sigma_1$ and the set of all its finite prefixes, denoted as $pref(\sigma_1)$, given by

$$pref(\sigma_1) = \{(s_0), (s_0, s_1), (s_0, s_1, s_2), \ldots\} \tag{6}$$

The union of all the sets of prefixes of each executions in $P_{safe}$, is given by $pref(P_{safe})$. During the union operation the elements which are common are collapsed into a single element, as there cannot be any repetition of members in the set. That is, the first element in the sets $pref(\sigma_1), pref(\sigma_2), pref(\sigma_3)$ and $pref(\sigma_1)$, which is the finite prefix of length 1 of $\sigma_1, \sigma_2, \sigma_3$ and $\sigma_4$ is the same, which is $(s_0)$. But during the union there will be only one element $(s_0)$ in $pref(P_{safe})$. Thus we have,

$$pref(P_{safe}) = \cup_{\sigma \in P_{safe}} pref(\sigma) \tag{7}$$

A safety property $P_{safe}$ is a monitorable property $P_{monitor}$, when $S^*/pref(P_{safe})$ is Turing recognizable. That is after examining a given execution, if we are able to determine in finite steps, if that execution is not in the set $P_{safe}$ then it is a monitorable property. Thus from the safety property we construct a monitorable property as given by,

$$P_{monitor} = \{\sigma_1, \sigma_2, \ldots\} \tag{8}$$

Once the above set, monitorable property is identified, we have to get a more useful definition of this set to be able to build a monitor for it.
Now we use the second given information, that is, the specified requirement and identify the monitored variables,

\[ V_m = \{ \text{variable}_1, \text{variable}_2, \ldots \} \]  

that are affecting the requirement. Then we construct the expression \( e \in \exp_{\gamma_{Pm}} \), which cover the specified requirement. If one expression is not feasible we may have to construct multiple expressions in complicated cases. Then we create the function valid value abstraction \( \gamma_{\exp_{Pm}} \). Its domain is \( S^\omega \) and its range is \( S^\infty \). This function takes the executions as input and gives out smaller sequences, of states which affect the expression \( \exp_{\gamma_{Pm}} \), and abstracts the rest of the states.

\[
\begin{align*}
\text{if } e & \in \exp_{Pm} & \gamma_{\exp_{Pm}}[e]_i = [e]_{s+1} \\
\text{if } e & \notin \exp_{Pm} & \gamma_{\exp_{Pm}}[e]_i \\
\gamma_{\exp_{Pm}}[s_i \sigma_{i+1}] = & \gamma_{\exp_{Pm}}[s_i \sigma'] \\
\gamma_{\exp_{Pm}}[s_i \sigma_{i+1}] = & s_i \gamma_{\exp_{Pm}}[s_i \sigma']
\end{align*}
\]

Thus,

\[
\gamma_{\exp_{Pm}} : S^\omega \rightarrow S^\infty
\]

where \( \sigma' \) is an infinite sequence of states and \( e_{s_i} \) for \( e \in \exp_{\gamma_{Pm}} \) is the result of evaluating the expression over \( s_i \), with \( i > 0 \). The smaller subsequence obtained contains the states, whose corresponding nodes define in terms of code the property to be monitored. Based on this, using the monitoring tools a monitor is constructed. This is duly illustrated in the examples in the next sections. Whenever during an execution we are monitoring, this property which is defining the set is violated, it becomes an unsafe execution path. As there exists no possibility on the way this unsafe execution can undo this anomaly to become again a safe execution, it does not belong to the set \( P_{safe} \).

\[
P_{unsafe} = \{ \sigma_{x_1}, \sigma_{x_2}, \sigma_{x_3}, \ldots \}
\]

(11)

Here \( x_1, x_2, x_3 \) are some natural numbers used as subscripts, to identify some execution which are not the members of the set \( P_{safe} \) and thus members of \( P_{unsafe} \). Using the monitor constructed the first transition which makes the sequence deviate from being a safe execution will be detected and appropriate action is taken like raising an alarm or sending a message to the server.

3 The sensor detecting violation of monitorable property due to internal problem

Many current day websites have databases of the information about their registered users. Such information about the users is crucial to modulate the websites’ services to suit the users behavior/needs, to profile the website for advertisements and other maintenance related works. We construct a simplified version of this scenario using a Java program. The program uses XAMPP web server for using the services like Apache HTTP and MySQL. A Java based tool named Java-MOP is used to generate the monitors to monitor the executing Java program. The Java program uses a database named registered users that contains a table named personalinfo with three columns namely username, password and visits. The program has user defined functions, username() and password() which receive two strings for username and password via the command line respectively from the user. The program then compares the username and password entered with corresponding column values of all the rows in the personalinfo table. When a matching is found in row, the program invokes executeUpdate() function to increment the visits column by one. If there is a redundancy in the database, the Java program calls the executeupdate() function multiple times for a single visit. Such an execution is not a safe execution. This anomaly leads to inaccurate records of visits in the database. The code of the Java program we described until now, is given below. As described in section 2, nodes are identified in this program. They are denoted by \( s_i \) where \( i \) starts from 1 for the first node and goes on. They are shown in the comment area of the corresponding Java statement that generated it.

```java
import java.sql.*;
import javax.sql.*;
import java.io.*;
import java.util.*;
import java.sql.*;

public class jdbcdemo {
    // s1.
    static String triLine1, strLine2; 
    public void username() { // s2.
        Scanner in = new Scanner(System.in); // s4
        strLine1 = in.nextLine(); // s5
        System.out.println("Enter the username"); // s3
        Scanner in = new Scanner(System.in); // s4
        strLine2 = in.nextLine(); // s5
        System.out.println("Enter the password"); // s3
        Scanner in = new Scanner(System.in); // s4
        strLine2 = in.nextLine(); // s5
        System.out.println("Update personalinfo set visits = visits+1 where username = "+strLine1+" and password = "+strLine2+"""); // s19
        String query1 = "Select * FROM personalinfo"; // s18
        String query2 = "Update personalinfo set visits = visits+1 where username = "+strLine1+"" and password = "+strLine2+"""; // s20
        String dbClass = "com.mysql.jdbc.Driver"; // s17
        String query3 = "Select count(*) as usercount from personalinfo"; // s16
        String dbname, dbpassword; int c;
        jdbcdemo ob1 = new jdbcdemo(); // s11
        ob1.username(); // s12
        ob1.password(); // s13
        String dburl = "jdbc:mysql://localhost:3306/student"; // s14
        String username = "root"; // s15
        String password = "PASSWORD"; // s16
        String dbClass = "com.mysql.jdbc.Driver"; // s17
        String query1 = "Select * FROM personalinfo"; // s18
        String query2 = "Update personalinfo set visits = visits+1 where username = "+strLine1+" and password = "+strLine2+"""; // s19
        String query3 = "Select count(*) as usercount from personalinfo where username = "+strLine1+"" and password = "+strLine2+"""; // s20
        Connection con = DriverManager.getConnection(dbUrl, username, password); // s22
        Statement stmt = con.createStatement(); // s23
        Statement stmt1 = con.createStatement(); // s24
        Statement stmt2 = con.createStatement(); // s25
        PreparedStatement pstmt = null; // s26
        ResultSet rs1 = stmt2.executeQuery(query3); // s27
    }
}
```
if(rs1.next() == true)// s28
{if(rs1.getInt("usercount") > 0)// s29
{ResultSet rs = stmt.executeQuery(query1);// s30
while(rs.next())// s31
{dbname = rs.getString(1);// s32
dbpassword = rs.getString(2);// s33
if(strLine1.equals(dbname) & & strLine2.equals(dbpassword))// s34
{pstmt = con.prepareStatement(query2);// s35
pstmt.executeUpdate();// s36
System.out.println("Authentication Failed"); // s37
}
}
}

/** end of while */
Resultset res1 = stmt.executeQuery("SELECT SUM(visits)
FROM personalinfo");// s37
while (res1.next())// s38
{c = res1.getInt(1);// s39
System.out.println("Sum of column = " + c); // s40
}
*/

/* end of while */
catch(ClassNotFoundException e) { // s44
e.printStackTrace(); // s45
}

catch(SQLException e) { // s46
e.printStackTrace(); // s47
}
 */
main } /end class
Now we define the set all the nodes obtained above as S,
given by
S = {s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14, s15, s16, s17, s18, s19, s20, s21, s22, s23, s24, s25, s26, s27, s28, s29, s30, s31, s32, s33, s34, s35, s36, s37, s38, s39, s40, s41, s42, s43, s44, s45, s46, s47}.
The figure below represents the control flow diagram obtained as mentioned in section 2. From this graph we obtain the following two paths as described in section 2.

\[ Path_1 = (s_{10}, s_{11}, s_{12}, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28}, s_{29}, s_{30}, s_{31}, s_{32}, s_{33}, s_{34}, s_{35}, s_{36}, s_{37}, s_{38}, s_{39}, s_{40}, s_{41}, s_{42}, s_{43}, s_{44}, s_{45}, s_{46}, s_{47}) \]

4Here there is an order in the possible repetition of the three two sub-sequences, indicated by the symbol we adopt from that used usually to indicate repetition in regular expressions. Though each repetition would create a distinct path, we collectively represent all of them by specifying value in the visit column corresponding to the row of matching username and password in the database table is incremented.

\[ Path_2 = (s_{10}, s_{11}, s_{12}, s_2, s_3, s_4, s_5, s_{13}, s_6, s_7, s_8, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{21}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28}, s_{29}, s_{42}, s_{43}) \]

This path is obtained when the user enters an invalid username and password. This login attempt gets rejected due to no match for the entered username and password is found in the database. These paths represent finite sequences of states. But the execution, \( \sigma_i \), where \( i \in N \) of a program is formally defined as an infinite sequence of states. To convert the above finite sequences into infinite sequences, we repeat the terminal state infinite times. Hence we create \( \sigma_i \) from \( Path_1 \) such that,

\[ \sigma_i = (s_{10}, s_{11}, s_{12}, s_2, s_3, s_4, s_5, s_{13}, s_6, s_7, s_8, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28}, s_{29}, s_{30}, s_{31}, s_{32}, s_{33}, s_{34}, s_{35}, s_{36}, s_{37}, s_{38}, s_{39}, s_{40})^+, s_{41}, s_{42}, ..., s_{47} \]

Similarly, we create \( \sigma_2 \) respectively from \( Path_2 \). Now along \( \sigma_i \) we observe that other infinite sequences \( \sigma_{i1}, \sigma_{i2}, \sigma_{i3}, ..., \sigma_{i38} \) created by dropping first, second, third, ... initial elements respectively are terminal sub-sequences of \( \sigma_i \) as shown below.

\[ \sigma_{i1} = (s_{11}, s_{12}, s_2, s_3, s_4, s_5, s_{13}, s_6, s_7, s_8, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28}, s_{29}, s_{30}, s_{31}, s_{32}, s_{33}, s_{34}, s_{35}, s_{36}, s_{37}, s_{38}, s_{39}, s_{40}, s_{41}, s_{42}, s_{43}, s_{44}, s_{45}, s_{46}, s_{47}) \]

Here it should be noted that \( \sigma_{i38} \) sequence is an infinite sequence of \( s_{41} \), which is the terminal state in \( \sigma_i \). Similarly we construct from \( \sigma_2 \) the infinite sequences \( \sigma_{21}, \sigma_{22}, \sigma_{23}, ..., \sigma_{238} \). By definition \( S^\omega \) is a set of all possible infinite sequences of \( s_i \in S \) including the above created infinite sequences of states.

\[ S^\omega = \{ (s_{11}, s_{12}, s_{2}, s_{3}, s_{4}, s_{5}, s_{6}, s_{7}, s_{8}, s_{9}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28}, s_{29}, s_{30}, s_{31}, s_{32}, s_{33}, s_{34}, s_{35}, s_{36}, s_{37}, s_{38}, s_{39}, s_{40})^+, s_{41}, s_{42}, ..., \} \]

Now we have to note that, the repetitions get collapsed, so that no two elements are same in the set \( S^\omega \), like \( \sigma_2 \) collapse into \( \sigma_{138} \). The sequences above depict infinite paths in the control flow graph as if there was an edge from every state to another including to itself. Using the executions got from the paths in graph we construct a set \( P \) as in equation 4. \( P = \{ \sigma_1, \sigma_2 \} \). Such a set of executions is called a property and we observe that \( P \subset S^\omega \). An observation is made in these elements. After initial subsequence \( \sigma[0,...i] \) in all the elements in \( P \), there exists an element \( \beta \in S^\omega \), to make the whole sequence to belong to the set \( P \). Consider \( \sigma_1 \). Let \( i = 0 \). Then the initial subsequence \( \sigma[0] \) is \( s_{10} \). Then we have a \( \beta \in S^\omega \) such that \( \beta = \sigma_1 \), making the concatenated sequence to be a valid member of \( P \). This essentially means the elements of the sequences which are members of the set \( P \) must be members of set \( S \). As we can easily verify that, we conclude as per equation 7, that the property \( P \) is called a

\[ ^5 \text{with no repetition considered} \]
safety property $P_{safe}$. That is, $P_{safe} = \{\sigma_1, \sigma_2\}$.

Prefix of an execution is an initial subsequence of the execution. Consider $\sigma_1$. The set of all the finite prefixes of $\sigma_1$ is $\text{pref}(\sigma_1)$, which is given as

$$\text{pref}(\sigma_1) = \{(s_{10}), (s_{10}, s_{11}), (s_{10}, s_{11}, s_{12}), \ldots (s_{11}, s_{12}, s_{13}, s_{14}, s_{15}, s_{16}, \ldots, s_{41})\}$$

Similarly we obtain the sets of prefixes of all the $\sigma$s in $P_{safe}$, which are, $\text{pref}(\sigma_2)$. We define the set $\text{pref}(P_{safe})$ to be the set obtained by the union of the sets $\text{pref}(\sigma_1)$ and $\text{pref}(\sigma_2)$. Thus we have,

$$\text{pref}(P_{safe}) = \text{pref}(\sigma_1) \cup \text{pref}(\sigma_2)$$

i.e. $\text{pref}(P_{safe}) = \{(s_{10}, (s_{10}, s_{11}), (s_{10}, s_{11}, s_{12}), \ldots \}$

Now we prove that the set $S^* / \text{pref}(P_{safe})$ is Turing recognizable.

Proof: We observe that each of the elements in the set $\text{pref}(P_{safe})$ are of finite length and so is the given sequence $\text{sigma}_test$, which is constructed on element at a time, corresponding to the monitoring of each state of the process since its invocation. Thus we construct a Turing Machine/algorithm below, which accepts a string if and only if it does not belong to the set $\text{pref}(P_{safe})$.

1. Before observing the first state of the test execution, set the length of $\text{sigma}_test$, $L = 0$.

2. Observe the next step of execution and build the test sequence $\text{sigma}_test$ by one more element and store it. Increment by one, the length variable $L$.

3. Pick all the sequences in the set $\text{pref}(P_{safe})$ of length $L$ and create a set $\text{pref}(P_{safe})$.

4. Now compare the test sequence $\text{sigma}_test$ with the first element in the set $\text{pref}(P_{safe})_L$. If there are no elements in the set $\text{pref}(P_{safe})$, then go to step 7.

5. If there is a match then go to step 2.

6. Else delete the picked element from the set $\text{pref}(P_{safe})_L$, and go to step 4.

7. Declare that $\text{sigma}_test$ is not in $\text{pref}(P_{safe})$ Stop.

The key point in the above proof is that, if something goes wrong in the execution to make it unsafe, then it must happen after finite steps and thus it would be detected by the above algorithm in finite steps.

As $P_{safe}$ is a safety property and $S^* / \text{pref}(P_{safe})$ is Turing recognizable, we conclude from equation 8, that the set $P_{safe}$ is a monitorable property. Let's denote $P_{safe}$ from now onwards as $P_{monitor}$. Thus $P_{monitor} = \{\sigma_1, \sigma_2\}$.

---

We reiterate that the main goal of monitoring in the present example, is to detect if there is any inconsistency in counting the visits by the registered users. This compels us to be particularly interested in the states where the user login is facilitated and updating the number of visit by the user is made.

In [8], some integer variables in the program code are the monitored variables. While the $\text{Monitor}_m$ in our example are Boolean variables $\text{user-login}$ and $\text{update-visit-count}$ representing login attempt of user and the updating of the count of registered user's visits in the database. These variables do not appear in the program code as variables. They correspond to the states in the execution facilitating the login and corresponding updating of the visit-count. The Boolean variable $\text{user-login}$ takes value 1, when login attempt is successful. It takes the value 0, when the attempt is unsuccessful. The Boolean variable $\text{update-visit-count}$ takes the value 1 when there is incrementing of the count by one. It takes the value 0 otherwise. $V_m = \{\text{user-login}, \text{update-visit-count}\}$.

The specified requirement in the example used in [8] is given by an algebraic expression specifying the integer variables, and the accepted range for them, each connected by a conjunction. In our example, the specified requirement is given by a Boolean expression. The expression $exp_{V_m}$ over monitored variables $V_m$ made to cover the specified requirement is given below.

$$exp_{V_m} = ((\text{user-login}) \land (\text{update-visit-count})) \lor ((\neg(\text{user-login}) \land (\text{update-visit-count}))) \land \neg((\text{user-login}) \land (\text{update-visit-count})))$$

The above expression is a conjunction of two expressions. The first expression consists of the what should happen and the second consists of negation of what should not happen. The first expression has two parts connected by a disjunction; first one indicates a successful login and a successful updating of count while the second a failure of both. The second expression has three parts connected by disjunction. The first one indicates a successful login corresponding to multiple updating of the visit count. Second one indicates a successful login with out corresponding updating of count of visit. The third one indicates an unsuccessful login corresponding to multiple updating of the visit count. This motivates to monitor the subsequence of states which affect the variables $V_m$ determining the evaluation of $exp_{V_m}$. The abstraction of the states from the whole sequence, which do not affect the evaluation of the specified requirement as covered by $exp_{V_m}$, is the function called value abstraction. $\gamma_{exp_{V_m}}$. We note that, the variable $\text{user-login}$ is operated on/ gets effected during the subsequence between states $s_{12}$ and $s_{13}$. The variable $\text{update-visit-count}$ is operated on/ gets affected at the state $s_{41}$. When value abstraction function $\gamma_{exp_{V_m}}$...
is applied to $\sigma_1$, the resultant sequence is $\sigma_{1 va}$ is given by,

$$\sigma_{1 va} = \{ \ldots, s_{12}, \ldots, s_{13}, \ldots, s_{36}, \ldots \}$$

such that the states other than these three important ones in the subsequence are abstracted. The value abstraction done captures the subsequence where first the username function is executed (denoted as $s_{12}$) and then second, the password function (denoted as $s_{13}$) is executed and third, the executeUpdate function (denoted as $s_{36}$) is executed. Similarly we have to do value abstraction to other execution in the set $P_{monitor}$. We now observe that, all such obtained value abstracted sequences are of two types; one which have all these three states $s_{12}, s_{13}$ and $s_{36}$, in the given order once as in $\sigma_1$ and another with the first two states $s_{12}$ and $s_{13}$, in the given order as in $\sigma_2$ once and then none of the three states appear in the sequence ever, to satisfy the given requirement. Here we note that the smaller sequences got after the value abstraction, define precisely if a sequence $\sigma \in S_\omega$ belongs to the set $P_{monitor}$ with the given requirement. The commonality among all the elements of the set $P_{monitor}$ is expressed by $H$ as below.

$$H = (H^* \cdot S_{12} \cdot H^* \cdot S_{13} \cdot H^* \cdot S_{36} \cdot H^{\omega})$$

$$\bigcup (H^* \cdot S_{12} \cdot H^* \cdot S_{13} \cdot H^{\omega})$$

(12)

where $H^*$ and $H^{\omega}$ are respectively the finite and infinite subsequences of states without the states $s_{12}, s_{13}$, and $s_{36}$ in them. $H$ specifies three states and the order in which they should appear as in equation 12. Here we realize that all the many possible sequences that could happen distinctly from the execution $\sigma_1$ is drastically reduced to only those that abide by the above definition of the set given by $H$ above. We followed here the formal procedure in [8], to identify the subsequence of states, which define the monitorable property satisfying the specified requirement. We now identify the nodes corresponding to these states. They represent three function calls called username, password and the execute-update in the program. Therefore we deduce here that the order of these function calls as the defining monitorable property satisfying the given requirement. Now using Java MOP tool which supports MOP paradigm we generate monitors for sequence of function calls (username(), password(), executeUpdate()) which is the identified safety property. We specify this property using the logical formalism of Extended Regular Expression (ERE) as ere:(username password executeUpdate) used as a plug-in in the tool. The tool then automatically generates monitor for the specified property and uses AspectJ to integrate it together with the code.

3.1 An unsafe execution path

As mentioned earlier, we have an internal error which causes an anomaly during run-time. There are multiple entries for a particular user in the database. Thus when this user logs in the corresponding count is incremented at (In our experiment there were two entries for a particular user) multiple rows. This leads to an inaccurate count of the visits of registered users in the database. The execution followed by the program during the successful log in of this particular user is as follows.

$$\sigma_3 = (s_{10}, s_{11}, s_{12}, s_{2}, s_{3}, s_{4}, s_{5}, s_{13}, s_{6}, s_{7}, s_{8}, s_{9}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{21}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28}, s_{29}, s_{30}, s_{31}, s_{32}, s_{33}, s_{34}, s_{36}, s_{31}, s_{32}, s_{33}, s_{34}, s_{36}, s_{37}, (s_{38}, s_{39}, s_{40})^*, s_{41}, s_{41}, \ldots)$$

We observe that until the state $s_{33}$ appears for the first time, the sequence is safe. When the next state after $s_{36}$ is $s_{31}$ already an anomaly is created. But when the next state is again $s_{33}$, the order of function calls is violated, as the update-visit-count function is repeating and is not preceded by a successful username and password function calls. By violating the sequence as in $\sigma_{va}$ of function calls, The property defined by $H$ is being violated. Thus, the particular element in prefix pref($\sigma_3$), as given below,

$$pref(\sigma_3) = (s_{10}, s_{11}, s_{12}, s_{2}, s_{3}, s_{4}, s_{5}, s_{13}, s_{6}, s_{7}, s_{8}, s_{9}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{21}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28}, s_{29}, s_{30}, s_{31}, s_{32}, s_{33}, s_{34}, s_{36}, s_{31}, s_{32}, s_{33}, s_{34}, s_{36}, s_{37}, (s_{38}, s_{39}, s_{40})^*, s_{41}, s_{41}, \ldots)$$

cannot find a $\beta \in S_\omega$, such that this prefix can be concatenated to make an infinite path of execution belonging to $P_{monitor}$ satisfying the given requirement. This was duly demonstrated by our experiment in which monitor raises an alarm during the execution when $s_{36}$ is repeated. The anomaly is caught after four states delay from its beginning at $s_{31}$. During execution when the specified property is violated the monitor then triggers the user defined violation handler which in our case is displaying the statement “MOP: FAIL” or any other user defined statements. The monitor then takes the specified recovery actions which in our case is to immediately stop the execution of the program.

4 The sensor detecting violation of monitorable property due to external problem

This example is a simplified scenario of registered user accessing his account information from a system. Here the Java program which enables this feature to the registered users, accepts three inputs from the user: username, password, file name in an executable file. The first two are used for user authentication. The file name is used to identify
the name with which the system stores the user’s account information in a file. The third input which is given in an executable file, will be written on a text file inside the system, and then the file corresponding to that file name is to be retrieved from the secure area of the system and its contents are to be printed out on screen. Here the user given executable file is created by the user by compiling his own C program Register.c. He only knows the file name on which his file name is to be written. But apart from that he does not know what processing will happen afterwards based on his writing his file name on that file. He is not allowed to directly execute the executable file he created, but he needs to give it as an input to the Java program which is doing the authentication and processing. Once he gives the executable file as input, the Java program using "exec" command, executes this executable file. Here the "OS command injection" method, which is the second most prominent threat over Internet today is used to attack the system. Inside the C program Register.c, he had to open the file Users.txt in write mode by specifying the command "write". Due to malicious intentions, user specifies "append". This makes this user’s register name written on the file Users.txt, in the append mode, that is, it does not erase the previous contents. That is, who ever, had visited earlier to get his account information, his username is also retained in the file Users.txt and this malicious user’s file name is also added in the next line. Thus the Java program processes both the file names and gives out the contents in the files associated with both the file names. This malicious user was not "entitled/authorized" to know about the account details of the other user. But he came to know about it. Thus it breached the confidentiality of the user who visited the system earlier. This anomaly happens mainly due to the accessing two files associated with two file names and outputting their contents, after one successful authentication of a user. As in section 3, we present below the Java program.

```java
import java.io.*;
import java.util.*;
import java.sql.*;
import java.io.File;

public class FileRead4 // s1.
{
    static String strLine1, strLine2, strLine3;
    public void username() // s2.
    {
        Scanner in = new Scanner(System.in); // s3.
        System.out.println("Enter USERNAME"); // s4.
        strLine1 = in.nextLine(); // s5.
        } // s6.
    public void password() // s7.
    {
        System.out.println("Enter PASSWORD"); // s8.
        strLine2 = in.nextLine(); // s9.
    } // s10.
    public void main(String args[]){ // s11.
        try { FileRead4 fr = new FileRead4(); // s13.
            fr.username(); // s14.
            fr.password(); // s15.
        } // s12.
    } // s16.
}

public class FileRead4 // s17.
{
    static String strLine1, strLine2, strLine3;
    public void username() // s18.
    {
        Scanner in = new Scanner(System.in); // s19.
        System.out.println("Enter USERNAME"); // s20.
        strLine1 = in.nextLine(); // s21.
        } // s22.
    public void password() // s23.
    {
        System.out.println("Enter PASSWORD"); // s24.
        strLine2 = in.nextLine(); // s25.
        } // s26.
    public void main(String args[]){ // s27.
        try { FileRead4 fr = new FileRead4(); // s29.
            fr.username(); // s30.
            fr.password(); // s31.
        } // s28.
    } // s32.
}
```

The Java program processes both the file names and gives out the contents in the files associated with both the file names. This malicious user was not "entitled/authorized" to know about the account details of the other user. But he came to know about it. Thus it breached the confidentiality of the user who visited the system earlier. This anomaly happens mainly due to the accessing two files associated with two file names and outputting their contents, after one successful authentication of a user. As in section 3, we present below the Java program.

```java
import java.io.*;
import java.util.*;
import java.sql.*;
import java.io.File;

public class FileRead4 // s1.
{
    static String strLine1, strLine2, strLine3;
    public void username() // s2.
    {
        Scanner in = new Scanner(System.in); // s3.
        System.out.println("Enter USERNAME"); // s4.
        strLine1 = in.nextLine(); // s5.
        } // s6.
    public void password() // s7.
    {
        System.out.println("Enter PASSWORD"); // s8.
        strLine2 = in.nextLine(); // s9.
    } // s10.
    public void main(String args[]){ // s11.
        try { FileRead4 fr = new FileRead4(); // s13.
            fr.username(); // s14.
            fr.password(); // s15.
        } // s12.
    } // s16.
}
```

**String dbUrl = "jdbc:mysql://localhost:3306/student"; // s10.**
**String username = "root"; // s17.**
**String password = "PASSWORD"; // s18.**
**String dBClass = "com.mysql.jdbc.Driver"; // s19.**
**String query1 = "Select count(*) as usercount From personalinfo where username = "+strLine1+"" and password = "+strLine2++;" // s20.**
**Class.forName("com.mysql.jdbc.Driver"); // s21.**
**Connection con = DriverManager.getConnection(dbUrl,username,password); // s22.**
**Statement stmt = con.createStatement(); // s23.**
**ResultSet rs = stmt.executeQuery(query1); // s24.**
**if(rs.next() == true) // s25.**
**{ // s26.**
**System.out.println("The number of time println called is "+count); // s27.**
**count++; // s28.**
**} // s29.**
**else { // s30.**
**System.out.println("Thank You"); // s31.**
**catch (Exception e){ // s32.**
**System.err.println("Error: " + e.getMessage()); // s33.**
**} // s34.**
**} // s35.**
```
```java
while((line = input.readLine())!= null) // s36.
{ // s37.
    System.out.println(line); // s38.
} // s39.
```
```java
while((strLine = br.readLine()) != null) // s40.
{ // s41.
    System.out.println(strLine); // s42.
} // s43.
```
```java
String strLine4 = in1.nextLine(); // s44.
if(strLine4.equals("CONFIRM")) // s45.
{ // s46.
    String strLine5 = br.readLine(); // s47.
    if(strLine5.equals("CONFIRM") // s48.
    { // s49.
        System.out.println("Place the .exe file of C program in the respective folder and enter CONFIRM when done"); // s50.
        Scanner in1 = new Scanner(System.in); // s51.
        String strLine6 = in1.nextLine(); // s52.
        if(strLine6.equals("CONFIRM") // s53.
        { // s54.
            System.out.println("The number of time println called is "+count); // s55.
            count++; // s56.
            } // s57.
        } // s58.
        System.out.println("The number of time println called is "+count); // s59.
        count++; // s60.
        } // s61.
```

Now we define the set all the nodes obtained above as $S$, given by
$$S = \{ s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{21}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28}, s_{29}, s_{30}, s_{31}, s_{32}, s_{33}, s_{34}, s_{35}, s_{36}, s_{37}, s_{38}, s_{39}, s_{40}, \ldots \}$$
proved for a different class of languages, then we can construct a monitorable property. Theorem 2 states that the class of monitorable properties is Turing recognizable. This is a significant result, as it allows us to verify the correctness of systems against these properties.

Next, we consider the construction of a monitor. A monitor is a system that observes the behavior of a set of processes and reports when a property is violated. We can construct a monitor by using a finite automaton to recognize the monitorable property. The automaton is constructed by converting the regular expression into a nondeterministic finite automaton (NFA) and then converting the NFA into a deterministic finite automaton (DFA). The DFA is then used to monitor the system.

Finally, we consider the problem of monitoring a set of processes. We can monitor a set of processes by using a monitor for each process in the set. However, this approach is often not practical, as it can be expensive to monitor each process individually. An alternative approach is to use a single monitor to monitor the entire set of processes. This approach is more efficient, but it is more difficult to implement, as it requires a more sophisticated monitoring mechanism.

In conclusion, we have seen that monitorable properties are an important class of properties for verifying system behavior. The construction of monitors for these properties is a challenging but important problem, and one that has been the subject of much research in recent years.
of the register file with user’s information. The Boolean variable user-login takes value 1, when login attempt is successful. It takes the value 0, when the attempt is unsuccessful. The Boolean variable print-register-file takes the value 1 when the printing of the content of the register file of a user is successful. It takes the value 0 otherwise. Thus, \( V_m = \{ \text{user}\-\text{login}, \text{print}\-\text{register}\-\text{file} \} \) The expression \( \exp_{V_m} \) over monitored variables \( V_m \) made to cover the specified requirement is given below.

\[
\exp_{V_m} = (((\text{user}\-\text{login}) \land (\text{print}\-\text{register}\-\text{file})) \lor \\
(\text{user}\-\text{login}) \land (\text{print}\-\text{register}\-\text{file})) \lor \\
((\text{user}\-\text{login}) \lor (\text{print}\-\text{register}\-\text{file})) \lor \\
((\text{user}\-\text{login}) \land (\text{print}\-\text{register}\-\text{file}))
\]

The above expression is a conjunction of two expressions. The first expression consists of what should happen and the second consists of negation of what should not happen. The first expression has two parts connected by disjunctions; first one indicates a successful login and a successful printing of a register file content, the second indicate a failure of both and the third indicates the successful login of the user but a failure of printing the information in the file\(^{14}\). The second expression has two parts; first indicates a printing of contents of multiple files preceded by a single successful login and the second indicates printing of a file’s content when the login attempt fails.

This motivates to monitor the subsequence of states which affect the variables \( V_m \). Now we apply the value abstraction over the sequences in the set \( P_{\text{monitor}} \) using the above expression. We note that, the variable user-login is operated on gets effected during the subsequence between states \( s_{14} \) and \( s_{15} \). The variable print-register-file is operated on gets affected at the state \( s_{44} \). When value abstraction function \( \gamma_{\exp_{V_m}} \) is applied to \( \sigma_1 \), the resultant sequence is \( \sigma_{1, va} \) is given by,

\[
\sigma_{1, va} = \{ \ldots, s_{14}, \ldots, s_{15}, \ldots, s_{44}, \ldots \}
\]

such that the states other than these three important ones in the subsequence are abstracted. The value abstraction done captures the subsequence where first the username function is executed (denoted as \( s_{14} \)) and then second, the password function (denoted as \( s_{15} \)) is executed and third, the print-register-file function (denoted as \( s_{44} \)) is executed. Similarly, we have to do value abstraction to other executions in the set \( P_{\text{monitor}} \).

We now observe that, all such obtained value abstracted sequences are of two types; one which have all these three states \( s_{14}, s_{15} \) and \( s_{42} \), in the given order once as in \( \sigma_1 \) and \( \sigma_4 \), and another with the first two states \( s_{14} \) and \( s_{15} \), in the given order as in \( \sigma_2 \) and \( 3 \) once and then none of the three states appear in the sequence ever. Here we note that the smaller sequences got after the value abstraction, define precisely if a sequence \( \sigma \in S_m \) belongs to the set \( P_{\text{monitor}} \). The commonality among all the elements of the set \( P_{\text{monitor}} \) is expressed by \( H \) as below.

\[
H = (H^* \bullet S_{14} \bullet H^* \bullet S_{15} \bullet H^* \bullet S_{44} \bullet H^*)
\]

\[
\bigcup (H^* \bullet S_{14} \bullet H^* \bullet S_{15} \bullet H^*)
\]

where \( H^* \) and \( H^* \) are respectively the finite and infinite subsequences of states without the states \( s_{14}, s_{15} \) and \( s_{44} \) in them. We followed here the formal procedure in [8], to identify the subsequence of states, which define the monitorable property satisfying the specified requirement. \( H \) here specifies three states and the order in which they should appear as in equation 13. We now identify the nodes corresponding to these states. They represent three function calls used username, password and the println in the program. Therefore we deduce here that the order of these function calls as the defining monitorable property satisfying the specified requirement.

As earlier, we specify the order of calling the functions \((\text{username}(), \text{password}(), \text{println}())\) by ERC as \( erc=\text{(username password println)} \) in JavaMop to build the monitor.

### 4.1 An unsafe execution path

As mentioned earlier, we have an external problem, created by a malicious user’s input, which causes an anomaly during run-time. During the execution of the file provided by him, the program instead of opening the file in the "write" mode, will open in the "append" mode and retains the file name entered by the user who logged in before this malicious user. Thus when the program gets the register file’s name to print the content, it will get two files names and prints the contents of both. Thus the malicious user got to know about the information about another user, to which he was not entitled, thus breached the security by violating the confidentiality of that user. The execution path followed by the program during the successful log in of this particular malicious user and acting upon his malicious input is as follows.

\[
\sigma_4 = (s_{12}, s_{13}, s_{14}, s_{2}, s_{3}, s_{5}, s_{15}, s_{6}, s_{7}, s_{8}, s_{9}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{21}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28}, s_{29}, s_{30}, s_{31}, s_{32}, s_{33}, s_{34}, s_{35}, s_{36}, s_{37}, s_{38}, s_{39}, s_{40}, s_{41}, s_{42}, s_{43}, s_{44}, s_{45}, s_{10}, s_{11}, s_{45}, s_{46}, \ldots)
\]

We observe that until the state \( s_{45} \) appears for the first time, the sequence is safe. When the next state is again \( s_{38} \) leading to the loop repeating of the state \( s_{44} \), the order of function calls is violated, as the print-register-file function is repeating and is not preceded by a successful username and password function calls. By violating the sequence as in \( \sigma_{4, va} \) of function calls, The property defined by \( H \) is being violated. Thus, the particular element in prefix \( \text{pref}(\sigma_4) \), as given below,

\[
\text{pref}(\sigma_4) = (s_{12}, s_{13}, s_{14}, s_{2}, s_{3}, s_{5}, s_{15}, s_{6}, s_{7}, s_{8}, s_{9}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{21}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28}, s_{29}, s_{30}, s_{31}, s_{32}, s_{33}, s_{34}, s_{35}, s_{36}, s_{37}, s_{38}, s_{39}, s_{40}, s_{41}, s_{42}, s_{43}, s_{44}, s_{45}, s_{10}, s_{11}, s_{45}, s_{46}, s_{38})
\]

cannot find a \( \beta \in S_m \), such that this prefix can be con-
catenated to make an infinite path of execution belonging to $P_{\text{monitor}}$. But the earliest we could detect the anomaly was when the printing happened at $s_{44}$, which was six states later than when the anomaly started. When $s_{44}$ is repeated as in previous example the monitor detects the violation and displays the message.

5 Sensors in GIDA

Sensors act as the input devices in GIDA. The construction of monitors discussed in this work is used to build sensors to detect anomalies in the application layer. The desirable characteristics of the sensors include detecting different kinds of anomalies at the earliest, collecting all the relevant information possible about the anomalies in order to identify the form and the nature of the anomaly, initiating an emergency response even before relaying the information to the central server if the anomaly is critical in nature and the sensors should work in tandem to identify the cause of the anomalies.

6 Related Work

Kim et al. in [8] present the definitions and procedure to analyze the monitorable property in a program. They give an example of how to construct a valid value abstraction, through the expression of the monitored variable over a given requirement property. Viswanathan in [1], building on the work in Kim et al. in [8], presents the formal foundations of the run-time analysis of the software systems. Falcone et al. in [10] present a formal framework for defining the class of spaces of monitorable and enforceable properties. Based on such formal definition, they are able to obtain a formal structure of monitors as defined as Strett automata. Stoller et al., in [11] present a work to predict if a property will be satisfied or not by filling the gaps in the observation of the states of the program. They use HMM to estimate the state and arrive at a probabilistic measure to determine this. Acraini et al., in [12] present a work in which Java programs are annotated to link to ASM element which capture the property being monitored for. Using the CoMA monitor based on AspectJ, the executions are validated with respect to the ASM specifications.

7 Conclusion

We observe that by not restricting the abstraction of a program written in high level language to keeping track of variables expressed as standard data types in it, we are able to identify and monitor executions for the violations of a bigger space of monitorable properties. Though to achieve this, a diligent devising of valid value abstraction over the specified requirement is needed. Two such disparate monitors work as sensors to detect anomalies due to an internal error and an external intrusion to which the security system could respond were demonstrated.

References