Team Edit Automata for Testing Security Property *

Zhenrong Yang, Aiman Hanna, Mourad Debbabi
Concordia Institute for Information Systems Security
Concordia University, Montreal, Quebec, Canada
{zhenr_ya, ahanna, debbabi}@encs.concordia.ca

Abstract

This paper introduces a mathematical model, called Team Edit Automata, for evaluating software security properties. We use the model to describe security properties and their correlation in the software programs. The component automata can suppress and insert actions and report possible flaws. They are used to specify individual security properties. The team is composed of multiple component automata interacting through shared actions. It models the situation where some program events are concerned by multiple security properties jointly. The paper concludes by a case study of detecting memory management and pointer manipulation flaws in C/C++ programs.

1. Introduction

When concerning security evaluation of a software program, we must first ask three questions. What are the security properties we intend to evaluate? How can we formally specify these properties to guide our evaluation? How can we detect flaws in the software program that violate the security property specification?

This paper tries to answer the last two questions. We introduce a new mathematical model, called Team Edit Automata, for describing and evaluating software security properties. The architectural building blocks of a team edit automaton are component edit automata (also mentioned as “component automata” in the following text when there is no ambiguity). They are used to model individual security properties.

The team defines two things. Firstly, it defines the shared actions among component automata so that they can interact. Secondly, it defines the team’s response to the program input actions which are concerned by multiple component automata in the team. The team allows us to specify correlative security properties in software program more accurately.

We consider this paper to be contributive to the study of security evaluation in the following ways.

- We adapt the Edit Automata model for testing purpose and use program monitors modeled by the adapted automata to detect security flaws in software programs.
- We introduce a new architectural model, called Team Edit Automata, to combine multiple Edit Automata into a team. The model allows us to specify security properties more expressively and accurately.
- We demonstrate the practical aspect of our new model by using it to detect a common set of security flaws in C/C++ programs.

In section 2 of this paper, we present the previous works related to ours. In section 3, we define the problem of testing security properties and present our approach to it. In section 4, we formally define the Team Edit Automata model and in section 5 we demonstrate how this model is applied to the detection of some notorious security vulnerabilities in C/C++ programs. In the conclusion (section 6), we suggest future research works towards an automatic general-purpose security testing tool.

2. Related work

Static program analysis and dynamic program analysis are two main approaches to software security evaluation in literature and practice. In the field of static program analysis, Chen and Wagner [3] introduced MOPS, an infrastructure using model checking techniques to verify safety properties. Necula, McPeak, Weimer and others [7] introduced CCured, a tool mainly targeting memory safety violations in C programs. Engler and others [9] introduced metacompilation (MC) as a new approach to static program analysis. In the field of dynamic program analysis, a variety of
tools such as Libsafe, PurityPlus, and Immunix tools [5] have been developed.

2.1. Edit Automata

Using state machines to describe security properties is an active research thread. Schneider [8] introduced Security Automata to halt program execution when the programs’ behavior violates security properties. Ligatti, Bauer, and Walker [6, 1] extends the enforcing capability of the program monitors by introducing Edit Automata as a sequence transformer. In addition to simply halting the program execution, program monitors described by Edit Automata can suppress and insert actions to the programs. The model is proved to be able to enforce all security properties.

Analogous to an intentions file in database transactions, an edit automaton can temporarily suppress a sequence of actions that may violate the security property it monitors. Hence, instead of allowing potentially illegal actions to take effect, the edit automaton records the sequence internally and waits for the action that can guarantee the sequence to be legal. If such action arrives, the automaton will emit all previously suppressed actions and continue to process the upcoming actions. This way, it reserves the semantics of the program actions and enforces the security property as well. If the action is absent, the automaton halts the program execution (by suppressing all future actions).

2.2. Team Automata

In the study of groupwares, researchers use automata theory to construct formal models to specify the interaction and dependency within the systems. Particularly, Ellis and others [2, 4] introduced Team Automata as a mathematical model of the groupware systems. The model defines a way where multiple collaborative automata can be interconnected to form a team and where multiple teams can be interconnected to form a larger-scaled team.

A team automaton consists of multiple component automata, which are similar to ordinary automata except that their actions are classified into three categories - input actions, output actions, and internal actions. The categorization of actions can be viewed as an extra attribute of the actions. The team relies on this attribute to connect multiple automata. For example, two automata are connected if the output action from one component automaton is the input action to the other.

The team automata defines shared actions among component automata such that a single action can be "broadcasted" to multiple components. This is a behavior very similar to what we intend to add to our model.

3. Testing security properties

In this section, we present our approach to testing security properties. We decompose the evaluation of software security properties into three major phases:

1. Property specification, where we specify the security properties to be evaluated. The specification designates: i) what are security-sensitive program actions; ii) what behavior is allowed or disallowed by the properties; iii) how different properties may affect each other; and iv) on what conditions do we report flaws. We use Team Edit Automata presented in this paper to address the latter three issues.

2. Monitor instrumentation, where we translate the security property specification to the implementation of program monitors. We then instrument the monitors in the programs’ source code at lines where the actions concerned by the security properties are taken.

3. Program execution and flaw report. In this phase, we generate test cases and execute the programs. We then re-arrange the errors reported by the in-lined monitors for more comprehensive presentation.

The Team Edit Automata model plays a very important role in our approach. It is used to describe security properties; it represents the internal state transition engine of the program monitors; and finally, it is also used to define when a flaw should be reported.

4. Team edit automata

In this section, we give the formal definition of Team Edit Automata. The model combines the powerful enforcing capabilities of Edit Automata into a component-interactive architecture. It is a team composed of one or multiple component edit automata connected through action signatures - definitions that designate the source and destination of actions. Particularly, the team can define which flaws detected by its component automata should be reported and which ones be suppressed.

4.1. Notation

The following discussion is presented with the notation similar to what is used in [6]. We consider a software program as a set of program executions \( \mathcal{A} \). We use the notations \( \mathcal{A}^* \) and \( \Sigma \) to respectively denote the set of all sequences of actions of a program execution and an arbitrary set of executions such that \( \Sigma \subseteq \mathcal{A}^* \). We use \( \varepsilon \) to represent a single execution and the symbol \( \cdot \) to denote the empty sequence of program actions. We use \( \sigma \) and \( \tau \) to define single sequences.
of actions and \( a \) to denote single program action. Finally, we use the notation \( \sigma; a' \) to define the concatenation of two sequences of actions \( \sigma \) and \( a' \).

### 4.2. Component edit automata

Component edit automata are building blocks of our model. Each component automaton specifies one security property.

**Definition 1** A component edit automata \( C \) is a 4-tuple \( \langle Q, I, A, \delta \rangle \), where:

- \( Q \) is a nonempty finite or countably infinite set of automaton states;
- \( I \) is a nonempty initial state set such that \( I \subseteq Q \);
- \( A \) is a set of actions;
- \( \delta \) is a partial function \( A \times Q \rightarrow Q \) for transition relation.

The operational semantics of component edit automata, as defined by \( \delta \), is specified as follows:

\[
(q, \sigma) \xrightarrow{a} (q', \sigma')
\]

- If \( \sigma = a; \sigma' \) and \( \delta(q, a) = (q', a') \)
- If \( \sigma = a; \sigma' \) and \( \delta(q, a) = (q', \cdot) \)
- If \( \sigma = \sigma'; a; \sigma'' \) and \( \delta(q, a) = 0 \)

Accordingly, a component edit automata can insert more action(s), suppress the input action, and report possible flaws in the program.

The “insert” and “suppress” operations are similar to those defined in *Edit Automata*. As discussed in section 2.1, they guarantee that the component edit automata be able to specify all security properties.

The “report flaw” operation reports a possible flaw. Upon an input action “\( a \)”, for which the transition function \( \delta \) is not defined, the automaton outputs a special action “\( r \)”, signaling the team of a possible flaw, followed by all previous input actions. Instead of halting the system, the automaton stays in its current state and continues processing remaining input actions (\( \sigma'' \)). We make this design decision so that we can detect multiple flaws in each run of the target programs rather than halt the program execution at the first detected flaw.

The final judgment of the detection of flaws is delegated to the team, which make judgments based on outputs from all correlated component automata (those respond to the same input action).

### 4.3. Team Edit Automata

A Team Edit Automata is composed of one or more component edit automata. It groups correlative security properties (as component automata) in a team, coordinates the interaction among these components and ensures the team to respond to program inputs with explicitly defined outputs.

In our model, we use action signatures to describe the shared actions among component edit automata.

**Definition 2** An action signature \( a^T \) is a 3-tuple \( \langle \{C_j\} \cup \{-\}, a, \{C_k\} \rangle \) for \( j, k \leq i \) where \( \{C_j\} \) and \( \{C_k\} \) are sets of component edit automata and \( a \) represents an action.

An action signature has the following two types:

- \( \langle \{C_j\}, a, \{C_k\} \rangle \) defines an output action \( a \) from any component edit automata in set \( \{C_j\} \) to be piped as input to all component edit automata in set \( \{C_k\} \) (Pipe);
- \( \langle - , a, \{C_k\} \rangle \) defines an action \( a \) as the input action to all component edit automata in set \( \{C_k\} \) (Input).

Semantically, the Pipe and Input action signatures combine individual component edit automata into a group:

**Input** action signature correlates security properties that share concerns of a common set of program actions. For example, a program may have both file access and user authentication policies implemented. The file access policy forbids any user except the system administrator from sending packets from a socket connection once s/he opens any sensitive local file. On the other hand, the user authentication policy allows successfully authenticated users to send packets of information to the network. In this case, a user trying to send some packets of information to the network is concerned by both policies. An Input action signature may correlate these two policies into one team and define how to coordinate possibly conflicting properties. Hence, the specification of the expected implementation of the security properties in a software program can be defined more accurately.

Pipe action signature correlates security automata whose concerned program actions may affect each other’s state(s). In the same program as in above example, users may successfully authenticate themselves as the system administrator, open the sensitive local file, then start sending it over the network. To evaluate whether the program correctly implements the file access policy to deal with this scenario,
we can let the corresponding component edit automata to query the users’ authentication information then judge the correctness of the implementation. This is doable but ad-hoc in nature. Or, we can make a production of the component edit automata of the two policies and use the new production automata to monitor the implementation of both security properties. This is feasible but does not scale well. In our model, the interaction among the component edit automata is defined by allowing the output of some automata to be piped to the input of others.

We now give the definition of Team Edit Automata as follows.

**Definition 3** A Team Edit Automaton $T$ is a 3-tuple $(\{C_i\}, A^T, \delta^T)$, where:

- $\{C_i\}$ ($i \in N$) is a nonempty set of component edit automata. The Cartesian product of their states and initial states constitutes the states and initial states of the team respectively;
- $A^T$ is a set of action signatures $a^T_j$ ($j \in N$);
- $\delta^T$ is a partial function $\{A \times \{A_k\}\} \rightarrow \{A_l\}$ ($k, l \leq i$) for flaw judgment and output transformation.

$\delta^T$ determines the team’s observable outputs. Particularly, it defines what output the team should emit if multiple component edit automata emit different outputs upon an identical input action. In this case, $\delta^T$ is particularly defined as a partial function $\{A \times \{A_k\}\} \rightarrow \{A_l\}$ ($k, l \leq i$ and $r \notin \{A_l\}$). By explicitly defining $\delta^T$ for such conditions, testing engineers are able to specify the expected security property implementation more accurately.

A special usage of $\delta^T$ is to describe the condition, on which the team reports flaws. In this case, $\delta^T$ is defined as a partial function $\{A \times \{r\} \cup \{A_k\}\} \rightarrow \{A_l\}$ ($k, l \leq i$) where $r$ is the report-flaw output action from component automata. If a $r$ action is output by multiple component automata upon an input action $a$ in $A$, $\delta^T$ defines the team’s behavior of whether it should report the flaw(s) or not. Note that given the input action $a$, we are able to find all correlative component automata by looking in the definition of action signatures.

5. Case study

In this section, we use the Team Edit Automata model to assist in detecting memory management and pointer manipulation flaws in C/C++ programs.

We use two types of component edit automata to describe the legal behavior of pointers and dynamic allocated memory respectively. The relationship between a dynamic memory block and its referencing pointers is specified by action signatures $A^T$.

5.1. Modeling dynamic memory blocks

The component edit automata modeling dynamically allocated memory block are defined as:

$C_{mem} =$

\[
\{ q_{onHeap}, q_{freeStore}, q_{freeStore}, q_{dealloc}, q_{dieOut} \}, \\
\{ q_{onHeap}, q_{freeStore}, q_{freeStore} \}, \\
\{ a_{free}, a_{delete}, a_{delete}, a_{deref}, a_{addRefPtr}, a_{dealloc}, a_{refPtr}, a_{dealloc} \}, \\
\delta_{mem}, \\
Var_{StartAddr} \\
\}
\]

where state transitions defined by $\delta_{mem}$ is illustrated in Figure 1.

The nodes in Figure 1 represent the automaton states and the directed lines represent the transitions. A label above a directed line represents an input action. The label below a directed line represents the output actions. If there is no directed line for the current action, then the automaton emits a “$r$” action followed by the current action implicitly.

Informally, the automaton illustrated in Figure 1 starts monitoring a dynamic memory block when it’s newly allocated. Upon a deallocation input action, the automaton emits it then insert an “invalidatePtr” action. The latter is piped to the automata which model the memory block’s referencing pointers. The life cycle of the automaton stops when no more pointer references the memory block.

5.2. Modeling pointers

The component edit automata modeling pointers are defined as follows:

$C_{ptr} =$

\[
\{ q_{null}, q_{initialized}, q_{dieOut} \}, \\
\{ q_{null} \}, \\
\{ a_{assignNull}, a_{free}, a_{delete}, a_{delete}, a_{deref}, a_{addRefPtr}, a_{assignWidPtr}, a_{assignAddr}, a_{endLifeCycle} \}, \\
\delta_{ptr}, \\
Var_{MemStartAddr}, \\
Var_{CurrAddr} \\
\}
\]

where the definition of $\delta_{ptr}$ is illustrated in Figure 2.

The automaton shown in Figure 2 starts monitoring a pointer variable once it’s declared and stops when the variable exits its binding scope. Whenever the pointer is assigned to reference a memory block, the automaton emits the action and inserts actions to inform the automaton modeling referenced memory block. The $Var_{MemStartAddr}$ stored with the automaton is used as a unique id for the
Figure 1. Component Edit Automata modeling dynamic memory block

Figure 2. Component Edit Automata modeling pointers

team to pipe the action to the right component automaton of the memory block.

5.3. The team

With two types of component automata defined above, Here is how the team is constituted:

\[
T_{\text{mem-pts}} = \left\{ \left\{ C_{\text{MemStartAddr}} \right\} \cup \left\{ C_{\text{ptr }i} \right\}, A_{\text{mem-pts}}^T, \delta_{\text{mem}}^T \right\} \quad \text{for } i \in \mathbb{N}
\]

where \( A_{\text{mem-pts}}^T \) is defined as:

\[
A_{\text{mem-pts}}^T = \left\{ \begin{array}{c}
\left\{ C_{\text{MemStartAddr}}, \text{invalidatePtr}, \left\{ C_{\text{MemStartAddr}} \right\} \right\}, \\
\left\{ \left\{ C_{\text{MemStartAddr}}, a_{\text{free}}, C_{\text{MemStartAddr}} \right\} \right\}, \\
\left\{ \left\{ C_{\text{MemStartAddr}}, a_{\text{delete}}, C_{\text{MemStartAddr}} \right\} \right\}, \\
\left\{ \left\{ C_{\text{MemStartAddr}}, a_{\text{delete}} \right\}, C_{\text{MemStartAddr}} \right\} \right\}, \\
\left\{ \left\{ C_{\text{MemStartAddr}}, a_{\text{addRefPtr}}, C_{\text{MemStartAddr}} \right\} \right\}, \\
\left\{ \left\{ C_{\text{MemStartAddr}}, a_{\text{delRefPtr}}, C_{\text{MemStartAddr}} \right\} \right\}
\end{array} \right\}
\]

and \( \delta_{\text{mem}}^T \) is defined as:

- if \( \sigma = r; \sigma' \) then \( (a, \sigma) \rightarrow \sigma' \) (report flaw)
- if \( \sigma = \tau; \sigma' \) then \( (a, \sigma) \rightarrow \sigma \) (continue execution)

and \( \tau \neq r \)
In this example, the team is composed of one component automaton monitoring a dynamic memory block (mentioned as memory block monitor below) and multiple component automata monitoring pointers referencing the memory block (mentioned as pointer monitor below). Six shared actions are defined in the team. The action of invalidPtr corresponds to the scenario of invalidating referencing pointers after a memory block is deallocated. The actions of a_free[], a_delete[], and a_delete[] correspond to passing a pointer to memory deallocation function calls. The actions of a_addRefPtr and a_delRefPtr are triggered when a pointer starts or stops referencing a memory block respectively. The team reports any error detected by any of its component automata and emit all out actions from its component automata that are not piped to others.

5.4. Implementation and result

We implemented the team presented above as program monitors and instrument the monitors to some C/C++ programs in order to detect flaws in the programs. The result shows that the in-lined monitors are able to detect a list of following security flaws:

- dereferencing wild pointers or null pointers
- calling deallocation function on wild pointers
- memory leak
- double-freeing
- mismatching memory allocation and deallocation function calls, e.g. calling delete on memory allocated with alloc(size_t)

The component edit automata can be extended to monitor the initialization status of the allocated memory block. With the length constraint on this paper, we do not present the extended team automaton. Our implementation of the extension can detect a list of extra flaws, including: pointer arithmetic flaw, out-of-boundary memory access, and reading uninitialized memory.

6. Conclusion

This paper has introduced a mathematical model called Team Edit Automata for security evaluation. We use it to describe the security properties of software programs and the conditions on which flaws can be detected and reported. The model suggests that it is possible to take advantage of automata theory to specify software security properties (comparing to safety properties in many previous research works) for evaluation purpose.

We believe that the model is useful in the field of software security evaluation. It extends the previous approaches of using state machines for safety property specification so that more security properties can be modeled. It forms a framework for studying the correlation among security properties in the software programs - a research field which has been mostly unexplored.

Our future work is to define a general purpose language, based on this model, for specifying security property and for performing security evaluation. Researchers have contributed a lot of efforts [3, 1] to address this issue. With the help of the Team Edit Automata model, the new language may describe more security properties in a more accurate manner. We hope that with continued research into the field, software engineers and security managers will be able to define the exact security properties implemented in the programs and get a comprehensive evaluation report of program vulnerabilities.

References