Vulnerability Assessment of Web Services with Model-based Mutation Testing

Faezeh Siavashi
dept. Information Technologies
Åbo Akademi University
Åbo, Finland
Email: faezeh.siavashi@abo.fi

Dragos Truscan
dept. Information Technologies
Åbo Akademi University
Åbo, Finland
Email: dragos.truscan@abo.fi

Jüri Vain
dept. Software Science
Tallinn University of Technology
Tallinn, Estonia
Email: juri.vain@ttu.ee

***Abstract*—We present a model-based mutation testing approach, for evaluating the authentication and authorization of web services in a multi-user context. Model of a web service and its security requirements are designed using UPPAAL Timed Automata. The model is mutated to create invalid behavior which is used for test generation to reveal faults in the system under test. The approach is supported by a model-based mutation testing tool, μUTA, that automatically generates mutants, selects a collection of suitable mutants for testing and generates test cases from them. We modify a previously defined mutation operator and introduce three new operators for additional mutants. We define criteria for the mutation-selection and demonstrate the approach on a blog web service. Results show that the approach can discover authorization faults that were not detected by traditional methods.**

***Index Terms*—Model-Based Mutation Testing, Timed Automata Mutation, User Behavioral model, UPPAAL, Security Testing**

I. Introduction

Popular social web services such as Facebook, Twitter, concentrate large amounts of sensitive data related to their users and are expected to be responsible for their integrity and security. One of the top security risks that are reported by OWASP is the incorrect configured user and session authentication which enables attackers to exploit passwords, keys, or session tokens, or take control of users accounts to assume their identities [1].

Authentication and authorization are the two main ways of securing a web service and the data it maintains. Authentication is the process of verifying a user's identity, while authorization is the process of proving user's permission to access resources provided by a web service. For example, once a user is signed in to a blog (authentication), she can manage/edit her posted articles but is not allowed to manage/edit other users' articles (authorization). Furthermore, access control and authorization in web services are often defined based on user's roles in a group setting. For instance, a blog's administrator may have permission to manage all posted articles, whereas other users do not. Such role-based requirements make implementation of security systems of a web service more challenging.

As defined in BS 7799-3:2017 standard, vulnerability is a weakness of an asset or group of assets that can be exploited by one or more threats, where an asset is anything that has value to the organization, its business operations and their continuity, including information resources that support the organization's mission [13]. To ensure that the security system of a web service is implemented correctly, all possible scenarios of user activities should be tested in an ideal case. Since manual testing is error-prone and mostly not exhaustive, model-based testing (MBT) has gained more attention by offering automated test generation from the model of a system under test (SUT) and its environment. As models can be designed based on specific test oracles such as robustness and security, they create test cases to validate the systems based on the test oracles. The test cases are executed against the SUT, and the test outputs are compared with the expected outputs.

Leveraging MBT by adding mutation testing leads to more powerful testing technique, known as Model-based Mutation Testing (MBMT) [11]. In MBMT, the original test model is altered systematically by *mutation operators* creating multiple versions of a model (known as *mutant* models). The mutants can be used for automatic generation of invalid test inputs that are executed against the SUT. The goal in MBMT is to find whether any invalid tests can pass the testing, thus they can reveal unexpected behavior (i.e., fault) in the SUT. Hence, MBMT can expose the mistakes that are caused by missing requirements or incorrect implementation.

The concept of MBMT itself is not new, however to our knowledge, the security requirements such as authentication and authorization have not been assessed using such technique yet. Available modeling and testing approaches focus on the integrity of the specification based on individual user activities and not multi-user interactions.

In our previous work on testing web services with a similar MBMT approach, the model of the SUT was defined for single user activities in a web service with the purpose of evaluating the robustness of the SUT [36]. However, the authorization and privileges for multiple users were not designed and tested. In this paper, we extend and improve the MBMT approach with three main contributions, as follows:

- we extend the previous approach to evaluating the vulnerability of web services in multi-user context;
- we improve the mutation-selection process to achieve more efficient mutants (i.e., the mutants that reveal

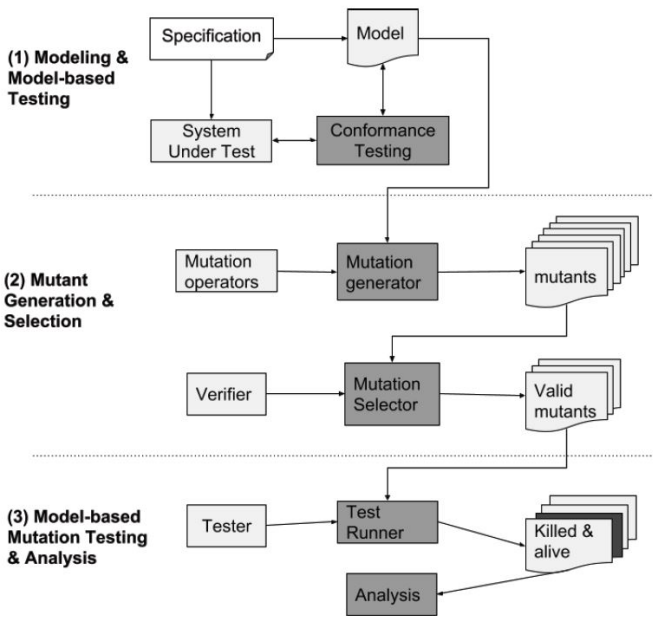

Fig. 1. Overview of our model-based mutation testing approach

faults);

- we introduce and evaluate three new mutation operators of Uppaal Timed Automata models and extend one of the previously defined mutation operators to create additional mutations.

We model a web service, its users and their authentication and authorization requirements. The model will be mutated to create invalid test inputs that target faults in the implementation of the web service. Besides, some mutation-selection criteria such as reachability are employed to provide more suitable mutants for testing.

Model-based mutation testing approach utilizes black-box testing for detecting vulnerabilities in web services, assuming that their source-code is not available.

Figure 1 presents an overview of the process. At first, a model is designed and verified by model checking. In the second step, mutants are generated by applying the mutation operators. In this paper, we select some suitable mutants from a list of mutation operators presented in [2] and [5], extend one of the operators and introduce three new operators to create further mutations. To increase the efficiency of MBMT, we apply a mutation-selection technique to the mutants and eliminates trivial mutants (i.e., mutants that are unreachable or incorrect). The selected mutants are called *valid* mutants and will be used for test generation.

In the third step, a test runner executes the valid mutants, mimicking possible faults in the SUT. If the SUT detects an invalid input, the corresponding mutant will be *killed*; otherwise, it will be *alive*. In the analysis of the results, the alive mutants will be assessed for detecting possible vulnerabilities in the implementation of the system. The efficiency of the mutation operators will be measured as well.

II. Preliminaries

We use Uppaal Timed Automata (Uppaal TA) as the base of modeling and mutating a web service. We first briefly describe TA and Uppaal TA formalisms and then review their formal definitions.

A. Overview of Timed Automata

Alur and Dill introduced the theory of Timed automata (TA) for modeling and verification of real-time systems [6]. TA are expressed as a set of locations and directed edges that can connect the locations to each other and extended with real-valued clocks. A timed automaton can execute individually or in synchrony with other automata. During execution of TA, all clocks increase with the same speed.

Clocks can be updated or reset along with transitions of TA. The transitions can be constrained by *guards* of edges and enable or disable transitions.

If there is more than one enabled transition at a time, then one of them will be chosen randomly. This characteristic provides more freedom to design non-deterministic behavior in the systems with random discrete events [20], such as real-time systems.

Definition 1. Timed Automata (TA)

Let C be a set of non-negative real valued variables of n clocks and $\mathcal{G}(C)$ be a set of guards on clocks that are conjunctions in form of $x \bowtie c$, where $x \in \mathcal{G}$, $c \in \mathbb{N}$, and $\bowtie \in \{\leq, <, =, >, \geq\}$. Let $v : C \rightarrow \mathbb{R}_{\geq}c$ indicates that a real value is assigned to every clock $c \in C$ and Let $\mathcal{U}(C)$ denote the set of *updates* of clocks. A timed automaton is a tuple (L, l_0, I, E), where:

- L is a finite set of *locations* and $l_0 \in L$ is *initial* location;
- $E \subset L \times A \times \mathcal{G}(C) \times 2^c \times L$ is a set of edges including an action, a guard and a set of clocks.
- $I : L \rightarrow \mathbb{G}(C)$ assigns location invariants.

A transition can be denoted by $l \xrightarrow{a,g,u} l'$, iff $(l, g, a, u, l') \in E$. A state in TA is defined in form of $s = (l, \overline{v})$, where l is a location and $\overline{v}$ is a non-negative clock value that satisfies the invariant of l. A TA progresses either by changing from a state to other by executing an edge,i.e., $(l, \overline{v}) \xrightarrow{a} (l', \overline{v}')$, or by staying in a location and passing time, i.e, $(l, \overline{v}) \xrightarrow{d} (l, \overline{v} + d)$, as long as the invariant of location l is true.

Definition 2. Timed Input-Output Automata (TIOA)

Timed Input-Output Automata (TIOA) was introduced as extensions of TA, in a way that the actions set, A is divided into two sets of inputs and outputs actions, A_i and A_o respectively [22]. The input actions model the behavior of the environment and output actions model the external actions of the system. Thus, for each input action in the system, there is an output. A TIOA, A is a tuple $< I_A, O_A, L_A, l_A^0, C_A, T_A >$, where:

- I_A is a finite set of inputs, labeled by "?", O_A is a finite set of outputs, labeled by "!",
- L_A is a set of locations that indicates the state of the system after the transition,
- l_A^0 is the initial location,
- C_A is a set of clocks instantiated to zero at l_A^0, and
- T_A is a set of transitions in the system.

The theory of TIOA is implemented in modeling frameworks such as UPPAAL.

B. Overview on UPPAAL Timed Automata & UPPAAL-TRON

UPPAAL is widely used modeling and verification tool for real-time and reactive systems. The tool and its formalism were introduced as Ph.D. thesis [31]. It extends TA with other data types in addition to clocks. In UPPAAL , due to the distinction between local and global variables, it is possible to model systems and their environment as separate interacting automata. Such functionality enables refining the specification of either of a system or the environment without having a significant change in the other. Moreover, various testing goals can be designed in the environment such as safety, robustness, user scenarios.

Definition 3. Uppaal Timed Automata

A UPPAAL TA model is a network of n timed automata that share variables, clocks and actions. A UPPAAL TA model A_i is a tuple $< L_i, l_i^0, C, A, E_i, l_i >$, $1 \leq i \leq n$.

A UPPAAL TA model of a system can be analyzed if a particular criterion will be satisfied during the execution of the model. This analysis is done by defining reachability properties in UPPAAL TA models as explained below:

In UPPAAL a model of a system and its environment can be synchronized by *channels* over edges. Channels are labeled by "!" as emitting and "?" as receiving. Thus, UPPAAL TA leverage modeling complex systems by supporting parallel transitions from different automata.

Reachability Analysis in UPPAAL TA

Reachability analysis in UPPAAL TA is implemented as a finite *symbolic state space* exploration by executing *symbolic computation steps*. A symbolic state denoted as (l, D), where l is a location of a timed automaton and D, clocks valuations, represents $\{(l', \overline{v}) | l' = l \wedge \overline{v} \in D\}$. The initial state of the automaton is (l_0, D_0), where $D_0 = \{\overline{v} | (l_0, \overline{v_o}) \xrightarrow{d} (l_0, \overline{v})\}$. The reachability for symbolic state indicates that the location l is reachable at some point of the time. A symbolic computation step is defined in the form of $(l, D) \xrightarrow{a} (l', D')$, representing an action followed by some delay. An action is reachable iff $(l, \overline{v}) \xrightarrow{a} (l', \overline{v}')$ and $D' = \{\overline{v}'' | (l, \overline{v}) \xrightarrow{a} (l', \overline{v}') \wedge (l, \overline{v}') \xrightarrow{d} (l'', \overline{v}'') \wedge \overline{v} \in D\}$.

Beside reachability, in UPPAAL TA, safety, deadlock-freeness and liveness properties also can be defined. The UPPAAL model checker contains an engine for verifying such properties [23].

Once a model of a system is verified based on its specification criteria, it can be used for validation of its actual behavior. UPPAAL utilizes an online testing tool, TRON which generates test cases from UPPAAL TA models and executes them against systems.

Conformance Testing With UPPAAL TA

UPPAAL TRON generates symbolic timed traces in UPPAAL TA models. A symbolic timed trace $TTrS$ of a UPPAAL TA model is a sequence of symbolic states, each state being defined as a tuple (l, D, v), where l is a location, D is the clock constraints and v a set of non-negative variables' values. Similar to the TA progress described above, a transition in UPPAAL TA from a symbolic state to another is possible either by an action or by some delay $(l, D, v) \xrightarrow{a/d} (l', D', v')$.

UPPAAL TRON takes environment constraints of a system into account. Thus, the model of a system and its environment are defined as TIOAs, where the model is split into two parts of the SUT and the environment that are synchronized by input/output actions. The interaction between the system and its environment are identified as observable actions in UPPAAL TRON. The actions among the SUT (or environment) with other systems are known as internal actions and are not observable. During the execution, these internal actions are abstracted as delays by UPPAAL TRON. Therefore, conformance testing contains delays and observable actions.

Definition 4. Relativized Timed Input/Output Conformance (rtioco)

For input enabled timed input/output labeled transition systems $i, s \in \mathcal{S}$ and $e \in \mathcal{E}$, relativized timed input/output conformance is defined asl below:

$$i \ \mathbf{rtioco}_e \ s \ , \forall \sigma \in \texttt{TTr(e)} . \mathbf{Out}(\langle i, e \rangle \ \mathbf{after} \ \sigma) \subseteq \mathbf{Out}(\langle s, e \rangle \ \mathbf{after} \ \sigma),$$

where $\mathcal{S}$ and $\mathcal{E}$ are TIOAs with observable inputs and outputs, i, s and e are initial states of the implementation under test, specification and environment respectively. `TTr(e)` is a set of timed i/o traces of the environment e, $\langle i, e \rangle$ and $\langle i, s \rangle$ are observable i/o actions that are synchronized. $\langle i, e \rangle$ **after** σ indicates that observable trace σ is executed on implementation i via environment e and $\langle i, e \rangle$ **after** σ means that an observable trace σ is evaluated on the specification s via environment e and returned a set of possible states. $\mathbf{Out}(states)$ contains a list of possible output actions or delays.

The practical implication of the previous definitions is that the implementation conforms the specification within a shared environment if and only if the observable i/o behavior in the model is always the same as the behavior of the implementation. Thus, the result of conformance testing with UPPAAL TRON will be one of the three cases: *passed*, *failed*, or *inconclusive*. When the specified behavior in the model conforms to the implementation, the test will pass; otherwise, it will fail. If the output is not in the set of inputs for the environment or no input/output is provided within the defined time (test timeouts), then the test result is interpreted as inconclusive.

C. Model-Based Mutation Testing

Mutation testing extends the fault detection capabilities of MBT by exposing more vulnerabilities of systems. It changes a system's program or its specification, to create new versions of the system (mutants). Mutation operators are rules that establish the mutants by altering the syntax of the program (or the specification). The syntax alternations usually result in different behavior in mutants. In model-based mutation testing, the mutants are generated from the test model of the SUT. The tests exhibit altered inputs (incl. faulty inputs) and input sequences against the SUT. Hence the mutants allow testing of the SUT with invalid.

Although mutation testing has shown to be more efficient than other test criteria [30], it suffers from a large number of mutants that are not suitable or have equivalent behavior to their original specification/program. Creating and executing all mutants are usually costly. In this paper, we follow the mutants selection principles, reachability, infection, and propagation (RIP) for killing mutants. The RIP model was primarily defined for code-based mutation testing [7]. We adopt the RIP model for model-based mutation testing. To kill a mutant:

1) It must be **reachable**, which means that the mutated part of the model is executed at some point of test run,
2) It must cause **infection** (i.e., changes the state), on the mutant model after the mutation is visited,
3) It can **propagate** the mutation through the model (i.e., the difference in the behavior of the mutant is observable).

If the reachability and infection are satisfied by a mutant, it is called *weak mutant* and if all conditions are met, it is called *strong mutant*.

In our MBMT approach, during mutant selection, we check whether each mutation is reachable and potentially infects the SUT during test execution. These conditions help to eliminate unfit mutants. In UPPAAL TA, reachability, liveness, and deadlock-free properties can be defined on states and paths. We define reachability criteria based on the model elements where the mutation is applied.

1) RIP condition for mutants in UPPAAL *TA:* In the context of UPPAAL TA, if a mutation applied on an edge, guard, or update, the **reachability** for *symbolic computation step* is defined using a global variable in the model and update its value on the action the mutation occurs. Let $m \in \overline{v}$, be a boolean variable that is initialized to zero and action a is a mutated action. A mutated action is reachable iff in $(l, \overline{v}), m = 0$ and in $(l', \overline{v'}), m = 1$, and $(l, \overline{v}) \xrightarrow{a} (l', \overline{v}')$ and $D' = \{\overline{v}''|(l, \overline{v}) \xrightarrow{a} (l', \overline{v}') \wedge (l, \overline{v}') \xrightarrow{d} (l'', \overline{v}'') \wedge \overline{v} \in D \wedge m = 1\}$

After a mutation is reached, if input/output actions are different from the original input/output actions, then it means that the mutation changes the original state and causes an **infection**. One way of detecting the infection is to apply *bisimulation* relation, which compares traces of the original model and its mutants and checks whether they are equivalent. We used this technique in [37]. The mutants that pass the reachability and infection conditions will be considered as valid mutants. Aichering and Jöbstl introduced such condition regarding refinement relation in [5].

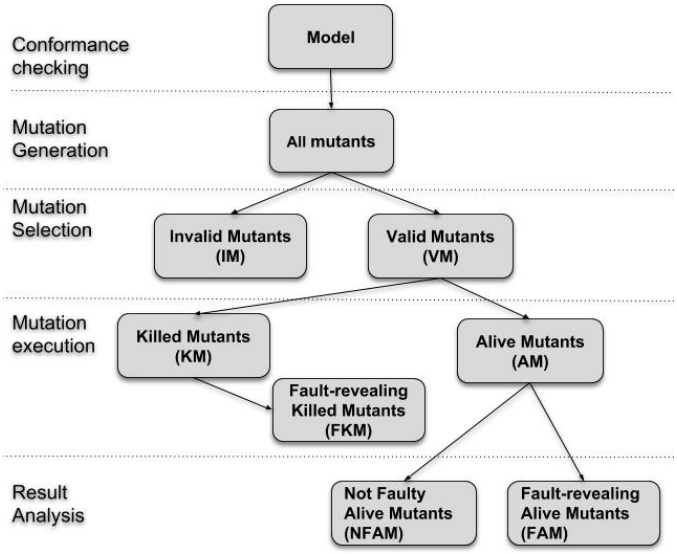

Fig. 2. Mutant classification in our approach

Mutated test cases are generated from the valid mutants the online testing tool UPPAAL TRON. The **propagation** condition can be checked during the test execution by comparing the observable behavior of the mutants with the behavior of the SUT. Since UPPAAL TRON evaluates the test results based on $rtioco$ relation, it is a proper tool for detecting propagation of the mutations at runtime.

If the mutation causes a change in observable input/output or the delays, and it can be detected during the test generation, then it considered as killed. Otherwise, if the SUT provides test outputs to the mutated test stimuli, then the mutated test case will not be detected and the mutant will be alive. The reason is that the SUT might be more permissive than its specification.

2) Mutation Classification in MBMT: In [11], Belli et al., an extended classification of the mutants for MBMT after executing them against the SUT. Such designation elevates the quality of testing and distinguishes whether the faults are raised by the mutants or by poor implementation of the SUT. In this study, however, we classify the mutants during mutation generation. The reason behind this is to identify more suitable mutants from the beginning and reduce the time of test execution and thus attain more efficient testing.

Figure 2 gives the classification in each step of our MBMT approach, starting from the mutation generation until the test analysis. After the model is designed and its conformance with the implementation is approved, we use the mutation operators to generate all possible combinations of mutants. Then, the reachability and infection properties of the mutants will be checked. The outcomes of this step are as follows:

- An **invalid** mutant model is either incorrect syntactically, or does not satisfy the mutation selection criteria (reachability and infection). Syntactically incorrect mutants are known as stillborn mutants.
- A **valid** mutant model must be a syntactically correct model and satisfy the mutation selecting criteria (reachability and infection).

Valid mutants are used for generating mutated test cases. During test execution they will be classified further as follows:

- A **killed** mutant model is a mutant which does not conform to the behavior of the SUT resulting in a failed or inconclusive verdict during testing. The killed mutants can be either trivial (killed by every test) or non-trivial, but in both cases, they are considered not interesting and thus eliminated.
- An **alive** mutant is a valid mutant that generates faulty behavior that cannot be observed or distinguished from the behavior of the SUT.
- If a mutant is killed but also reveals an anomaly in the SUT, then we count it as a **fault-revealing killed** mutant.

Since infection criterion ensures that the mutants are not equivalent to the original model, we do not have any equivalent mutants. Analyzing alive mutants classifies them as follow:

- If the SUT behaves the same as a mutant, then there is a fault in the implementation of the SUT. Such mutant is called **fault-revealing alive** mutant.
- If a mutant's behavior does not harm the SUT and does not cause any change in the state of the SUT, then it is known as a **not faulty alive** mutant. Distinguishing between such mutant and fault-revealing alive mutants should be checked by manual inspection.

III. Modeling web services with Uppaal Timed Automata

As mentioned earlier, UPPAAL TA can be used for modeling behavior of a system as well as its environment, which includes behavior of other systems and users that interact with the system under test. The interaction between a system and its environment is via observable channel synchronizations in the UPPAAL TA model.

First, we demonstrate a simple communication between a user and a web service with an excerpt of the actual model and then we explain how the model is extended for multi-user communication and their security specification.

Figure 3 shows two automata, a user and a web service. The automata are synchronized via channels following the request-response paradigm specific to web services. When a user wants to post a new article in a blog, she sends a request to the web service by clicking corresponding button or link in the browser. For each HTTP request, a channel is defined emitting from `User1` and receiving in `Blog`. The response to the request corresponds to a channel synchronization in the opposite direction, i.e., emitting from `Blog` and receiving in `User`. To edit and article, the user first gets access to the article ($manage\_article$) and then she submits the changes ($edit\_article$).

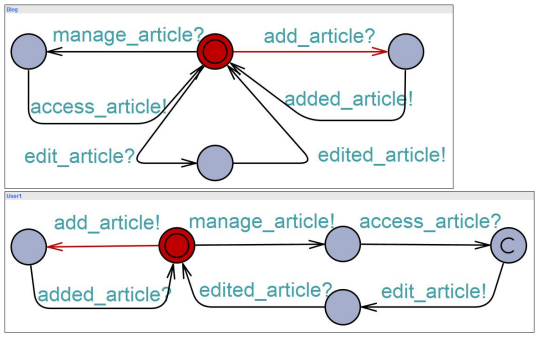

Fig. 3. Excerpt model of user interaction with a Blog

The extended version of model consists of `Blog`, `User1` and `User2` automata which describe multi-user interactions in a blog web service as shown in Figure 4. For demonstration, a limited number of user activities are modeled. From a user's point of view, the activities are limited to posting articles, commenting, reading, editing and deleting. In the web service's side, the events are in the form of HTTP requests that trigger corresponding functions creating or modifying resources or interacting with other systems.

The users automata are modeled with two different *user-id*s. The model contains some shared resources for tracking articles and comments and are accessible by both users. Each user can create new articles (add comments to the articles), or have access to available resources arbitrarily. The shared resources are defined as follows:

- *cu, sa, su* and *sc*, indicate "current user", "selected article", "selected user" and "selected comment" respectively. These variables enable random selection of users, articles and comments.
- *Users*, *max_art* and *max_cmt* variables indicate the number of users, maximum number of articles and maximum number of comments, respectively. Bounding the maximum numbers in the model prevents the possibility of a state space explosion.
- $article[Users][max\_art]$ assign articles to the users, thus, *article[2][3]* means that each user can create up to three articles. For instance, *User1* can create three articles which are mapped to *article[0][0], article[0][1] and article[0][2]*. The initial values are set to zero.
- $Comment[Users][max\_art][max\_cmt]$ contains data about the *user-id* of the person who comments on the articles. Thus, when *User1* (with id=0) adds a comment on *article 1* of *User2*, then we have *Comment[1][0][0]=0+x*. The constant value x is an offset number that prevents confusing the initial values (zero) with the *user-id* (id=0).

The $article$ and $Comment$ variables are updated by functions $addart()$ and $delart()$, $addcmt()$, and $delcmt()$.

As it is shown in user automata, the values of *user-id* and *article* are selected randomly:

$$u : int[0, Users - 1], a : int[0, max\_art - 1]$$

These values will be assigned to *su* and *sa* variables identifying the *selected user* and *selected article*. For instance, if u and a are zero and 1 and if the selected article exists (i.e., guard: $article[0][1] > 0$), then *manage_ar* can be executed. The model is non-deterministic and either of the two users can start sending requests.

General security requirements are extracted from the requirements and are presented in Table I. In each request, the user credentials should be sent to the web service and verified. The authentication is specified in the user automata by updating the *cu* variable, which is used as guard conditions in `Blog`. Besides, only the owner of a resource has the right to delete and edit his comments or articles. Therefore, in `Blog`, the *cu == su* guard is used to check this condition.

For instance, the first requirement in Table I includes an authentication condition, i.e., *the user should be verified*, and an authorization condition, i.e., *the user is the owner of the article*. The authentication condition is defined in such a way that the model updates the shared variable `cu` (i.e., *cu=0* or *cu=1*). When $dele\_cmt$ is requested, in `Blog` the authorization is defined by comparing the owner of the request and the owner of the article (*cu*). The way in which the security requirements of the model have specified allows us to scale the model up for verifying the behavior of more than two users.

IV. Model-Based Mutation Testing with μUTA

As shown in 1, our approach on MBMT includes three main steps. Once the test model conforms with the SUT (i.e., step (1)), it can be used for generating mutants. In this section, we describe step (2) and step (3) of the approach, namely *Mutation Generation and Selection*, and *Model-based Mutation testing and Analysis*. The μUTA tool automates step (2) and, partially, step (3). The analysis of the test results is done manually.

A. Mutation Generation & Selection

The idea of generating mutants for TA for testing the dynamic behavior of real-time systems was first presented by Nilsson et al. in [29]. Then, various mutation operators on timed automata elements were formally defined by Aboutrab et al. [2] and Aichernig et al. [3]. From the the previously proposed operators, we have selected the ones shown in Table II, which apply to *actions* and *guards*. However, the mutation operators for *invariant* and *locations* are not selected because they are not applicable in this particular case study.

- **Change Name of actions (CN)** – replaces the name of an action with the name of other actions in the model. Thus, the expected sequence of the inputs to the implementation will be different.
- **Change Targets of actions (CT)** – changes the target location of an action to another location in the model. This operator breaks the flow of test inputs and violates the state of the model. Both input and output actions can be mutated by this operator.
- **Change Sources of actions (CS)** – changes the source location of an action to another location. Similar to CT, this operator mutates the sequence of input/outputs.

In addition, we introduce three new mutation operators as follows:

- **Remove Actions (RA)** – randomly deletes one action at a time and creates a mutant. Omitting an action will manipulate the sequence of input/output actions.
- **Duplicate Actions (DA)** – randomly copies an action in different parts of a model, thus alternates the sequence of inputs and outputs by repeating actions in unexpected states of the model.
- **Remove Guards (RG)** – randomly selects an action and removes its guard. Actions that are mutated by RG will be always enabled.

Previously, the mutation operators on guards were defined to negate conditions, in [3], or to alter timing constraints [2]. We modify this operator and split it in two new operators:

- **Change Guards Logical operators (CGL)** – changes logical operators (i.e., $==$, $<=$, $>=$, $!=$, $<$ and $>$) in guards.
- **Change Guards Variables (CGV)** – alters values of the variables that are used in guards and creates additional mutants that cannot be defined by other mutation operators.

μUTA implements the mutation operators which generate mutants from the given Uppaal TA model. Distinguishing between the valid and invalid mutants with the tool is done by defining some model-checking conditions. The tool uses *verifyta* [9] to select valid mutants. Verifyta is an Uppaal interface that can verify an Uppaal TA model based on given list of conditions that are formalized as queries. The mutants are killable if they do not satisfy at least one of the following conditions:

Deadlock-freeness and livelock-freeness – If the mutants have deadlock or live-lock, they are killable. Deadlock means that a system enters to a state where no further action is enabled, while live-lock means that a system reaches to a condition that continually switches among some states forever, without any progress. Uppaal supports verifying deadlock-freeness, however, livelock-freeness for mutants are automatically defined by μUTA.

Mutation reachability – As we described in Section II-C, for the mutation operators that alter actions (i.e., CN, CT, CS, and DA), we apply symbolic computation steps. In Uppaal a boolean variable is declared and initialized to `False`, and on the mutated action, it will be updated to `True`. Therefore, if the action is fired, then the variable will be set to `True`. Consequently, the reachability of the mutated action can be checked symbolically during simulation.

input/output traces – To ensure that the input/output trace in the mutant is different than input/output trace of the original

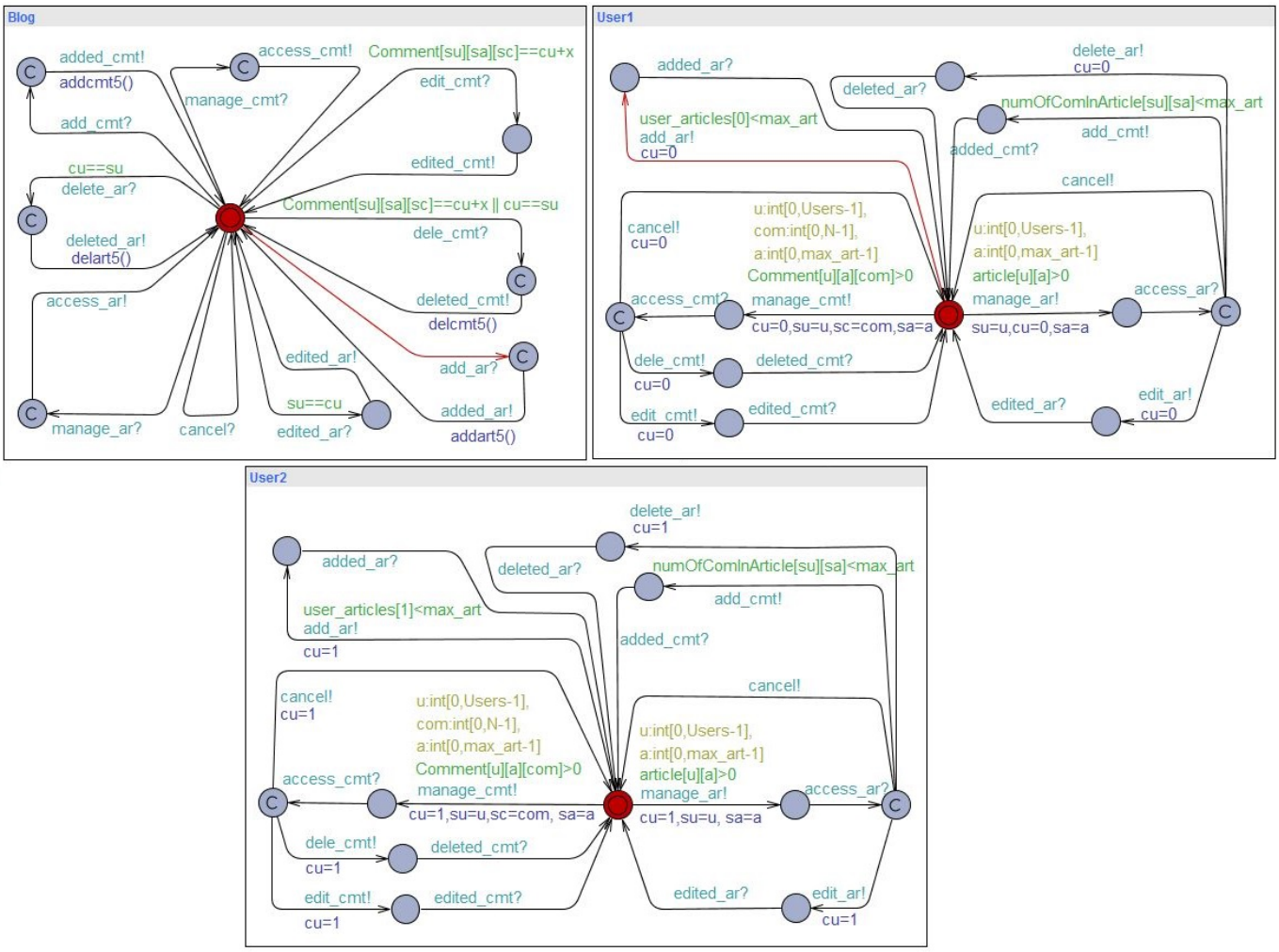

Fig. 4. A UPPAAL TA model of interactions of two bloggers within a Blog

model, we create a set of traces that contain all edge coverage and all location coverage. If a mutant model does not follow the traces, then it is considered as invalid; otherwise, it is valid. The comparison is automatically done via *bi-simulation* following the approach presented in [37].

B. Mutation Testing & Analysis

In the last step of the approach, we use the valid mutants for test generation against the `Blog` web service and analyze the results. A tester sets up individual test sessions using the UPPAAL TRON testing tool. The test execution from the model-level inputs to actual HTTP requests is established by a test adapter, which converts input/outputs actions into their corresponding requests/responses. A HTTP request will be sent to the `Blog` web service as an URL, and UPPAAL TRON waits for the response from the web service. The test adapter converts the response into a receiving action in the model. The result of each test will be categorized into alive and killed mutants, based on the verdict of the test session.

The analysis of the result is a manual process. Distinguishing whether an alive mutant addresses a genuine bug or it is in fact an equivalent model is yet a manual process. Moreover, not all possible bugs that are found during the mutation analysis will reveal vulnerabilities in the implementation or indicate wrong specifications. A bug proves that the expected and actual behavior do not conform, while vulnerability is a specific bug that manifests the possibility of exploiting the system under test. Therefore, deciding whether an alive mutant is a bug, which reveals vulnerability should be done manually based on the experience of the tester. Defining useful mutation operators has a significant impact on computational time and test effort. Typically, mutation score is a standard for evaluating the mutation operators.

Mutation Fault Detection The ratio of mutants (killed or alive) that reveal faults in the SUT to the number of generated mutants is measured by

$$MFD = \frac{FAM_i + FKM_i}{VM_i},$$

TABLE I
SECURITY REQUIREMENTS FOR A WEB LOG

Security requirements	Authorization	Authentication
A user can delete any comment under his article.	Guard in Blog ($comments[su][sa][sc] == cu$)	update in User1/User2 (cu=0, or cu=1)
An **article** can only be edited/deleted by its owner.	Guard in Blog (cu==su)	”
A **comment** can only . be edited by its owner	Guard in Blog (cu==su)	”
A **comment** can only be deleted by its owner OR by the owner of the article.	Guard in Blog ($Comments[su][sa][sc] == cu \|\| cu == su$)	”

TABLE II
MUTATION OPERATORS (* NEW OPERATORS)

Element	Mutation Operator	Description
Action	CN	Change names of actions
	CT	Change targets of actions
	CS	Change sources of actions
	RA*	Remove actions
	DA*	Duplicate actions
Guard	RG*	Remove guards
	CGL	Change guards logical operators
	CGV	Change guards variables

where FAM_i indicates the number of fault-revealing alive mutants, FKM_i is the number of fault-revealing killed mutants, and VM_i is the number of all the valid mutants generated.

Mutation testing has some fundamental problems. One of the problems is caused by having identical mutants generated by two or more mutation operators. Redundant mutants not only increase the test generation and test execution time but also have an impact on the validity of the assessment. However, this problem is tackled in our approach. The mutation operators in the μUTA tool are carefully designed, implemented and tested to prevent generating redundant mutants. Each mutation operator is implemented in such a way that provides unique mutants.

Another main problem is the equivalency. Equivalent mutants are those that are behaviorally equivalent to the original model. In code-based mutation testing, equivalency has been proven to be undecidable. However, in timed automata, it can be prevented. Our technique for solving such problem is to compare input and output traces of each mutant with the original model.

V. EVALUATION

A. Blog Web service

We selected the Blog web service as an example of a web service that provides multiple user interactions and supports some of the general security requirements such as authorization and authentication. The Blog web service is designed in REpresentational State Transfer (REST) [34] architectural style and provides functionality for creating new user accounts, posting new articles, commenting, deleting/editing posts and comments, managing user's profile, similar to common social networks. The web service is implemented in Python using Flask web developing micro framework [19]. We chose Flask as it has a simple and flexible structure which does not restrict developers to specific formats. This feature, however, makes the web applications prone to error and an interesting topic for testing.

B. Tool chain

Our MBMT approach is supported by a set of tools that automate the mutation generation, selection and execution procedures. The UPPAAL model-checker is used for modeling and verification the original test model. For online conformance testing, we use UPPAAL TRON.

μUTA tool contains some components: a *generator*, a *selector* and a *test runner*. The generator systematically creates mutants form a given UPPAAL TA model based on given set o mutation operators. The selector is a component that creates model-checking rules for each mutant and utilizes the *verifyta* tool [9] to validates them. If the rules are satisfied by verifyta, then the mutants will be classified as valid, otherwise discarded (i.e., invalid mutants). The test runner sets up test sessions using UPPAAL TRON, orchestrates the execution of valid mutants against the implementation of the SUT, and synthesizes the test results in a report.

To execute test cases that are generated by the mutants, we developed a test adapter, which translates the model-level test cases into test scripts that create HTTP requests that include test inputs. The responses from the web service under test will also translated by the test adapter into model-level responses.

The μUTA tool is developed using Python language and the test adapter is developed in Java language.

C. Results

The results of the MBMT approach for the Blog case study are presented in detail in Table III. From the `Blog` model, in total 2962 mutants were generated, whereas only 138 were valid for testing (i.e., VM). The generation and validation process took 24 hours and 44 minutes on a PC running Windows 7 Enterprise 64-bit operating system with Intel quad-core CPU, and 16 GB RAM. The test execution time for each valid mutant set to 150 seconds. It took about 200 minutes to run all the valid mutants against the SUT.

TABLE III
MODEL-BASED MUTATION TESTING PROCESS AND THE RESULTS.

	Mutation Generation and Selection			Mutation Testing			Analysis		
Operators	time	#of IM	#of VM	#of KM	#of FKM	#of AM	# of FAM	#of NFAM	MDF
CN	0:51:24	302	2	2	0	0	0	0	0
CT	01:18:22	160	20	12	0	8	0	8	0
CS	01:08:34	171	9	1	8	0	0	0	88.9%
RA*	00:02:05	9	9	9	0	0	0	0	0
DA*	18:43:57	2099	79	78	0	1	0	0	0
RG*	00:24:30	3	1	1	0	0	0	0	0
CGL	0:36:08	8	12	12	0	0	0	0	0
CGV	1:39:00	210	6	4	0	2	2	0	33.3%
Total	24:44:00	2962	138	119	8	11	2	8	7.2%

In total three different defects are found, one of them is revealed during the test execution and two others are found during the analysis.

From 138 valid mutants, 119 mutants were killed during the test execution (i.e., KM), eight mutants caused a crash in the SUT during the test and by investigating them, we found that all of the crashes belong to a bug in the implementation. Therefore, they are labeled as fault-revealing killed mutants (FKM).

Eleven mutants passed the test execution and remained as alive mutants (AM). All FKMs were generated by the CS operator and revealed a vulnerability in the implementation of the SUT. From eleven AMs, eight of the mutants were generated from the CT operator, two from CGV and one from DA, respectively.

The alive mutants were investigated by comparing their behavior to the original model, and whether they were representing any vulnerabilities. Deciding whether an AM shows faults (FAM), or does not create a fault (NFAM) is done manually. For instance, if a mutation changes the input/outputs in a way that the mutant does not generate additional behavior, then the mutant is NFAM. For example, in the `User1` automaton, a mutant is generated using the CT operator, which changed the target of `access_ cmt?` edge to the initial location, then the actions `dele_ cmt!`, `edit_cmt!` and `cancel!` cannot be executed (they are not accessible via the `access_ cmt?` action), thus the mutant is not able to generate the same traces as the original model. However, the mutant's traces will not be faulty and it will be not faulty alive mutant (NFAM).

To this extent, we investigated the AMs and found that none of the eight alive mutants generated by the CT operator were able to create faulty behavior and thus they were counted as not faulty alive mutants (NFAM). Similarly, the single alive mutant generated by the DA operator did not address any vulnerabilities in the SUT and was counted as a NFAM. The two CGV mutants, however, were able to reveal two distinct vulnerabilities in the SUT and were labeled as faulty alive mutants (FAM). The vulnerabilities were caused by mistakes in two separate parts of the implementation and are explained below.

D. Describing the bugs and identifying vulnerabilities

As described in Table IV, there are three bugs that were detected by the MBMT approach.

The first bug was detected during the test execution. The mutations were applied on the "delete" request for non-existing resources. The expected behavior of the SUT in this situation is to reject the request by a standard HTTP response, such as "`404: Not Found`". It can be done by adding a condition in the source code to check if the resource is available. However, this condition was missing in the code. Thus, invalid delete requests could crash the execution of the SUT. Eight FKMs generated by CS revealed the same bug.

In the analysis of the alive mutants, we detected two other bugs that show incorrect authorizations in two different HTTP requests: deleting articles and editing comments. In both cases the expected behavior of the SUT is first to verify whether the user is the owner of the resource, however, since such condition was missing in the source code, the SUT wrongly allows unauthorized users to modify resources. Two FAMs generated by CGV detected these bugs.

Vulnerabilities are the specific bugs that can exploit the system under test. If the bugs can cause abusing the system or unintended behavior occur in the system, then they are the system's vulnerabilities. Therefore from the bugs that we detected, we concluded that all three bugs are Blog vulnerabilities.

E. Efficiency of MBMT

Mutation Fault Detection (MFD) score is presented in the last column in Table III. CS has the highest score of 88.9%, followed by CGV with 33.3%, whereas other operators did not reveal any fault and got zero scores. The result shows that having a large number of mutations may not necessarily provide better results. For instance, DA generated the highest number of valid mutants. However, all of them were killed during the test execution. In contrast, CS has only nine valid mutants, which eight of them revealed one fault.

F. Testing Effort

To evaluate whether applying the reachability criterion is efficient regarding mutation generation and testing effort, we conducted two experiments on the same case study: (1) MBMT without verifying mutations' reachability properties,

TABLE IV
DESCRIPTION OF DETECTED VULNERABILITIES

Fault description	Operator	on which step
The SUT stops working when it receives a *delete* request to an non-existing article	CS	during testing
The SUT allows the user to delete an article without authorization check	CGV	during analysis of alive mutants
The SUT allows the user to edit Comments without authorization check	CGV	during analysis of alive mutants

and (2) MBMT including verification of mutations' reachability.

In the first experiment, the time of the mutation generation was roughly 120 hours, which is five times more than the second experiment. Moreover, the number of valid mutants for testing in the first experiment was 348, whereas in the second experiment we had 138 valid mutants. The test execution and analysis of the results in both experiments were similar. This comparison confirms that applying the reachability criterion significantly reduces the time of mutation generation and selection and consequently the time of test execution.

G. Evaluation of Mutation Operators

In this work, we extended one of the previously defined mutation operator to create mutants on values of variables in the guards (CGV) and were able to detect two authorization defects. The results suggest that additional modification on the mutation operators may provide more efficient mutants.

The new mutation operators (RA, DA, and RG) offer a large number of valid mutants for testing. However, none of the mutants were able to reveal any fault. The mutation operator for removing actions (RA) simulates a condition in the web service where there is no response to a request or there is a response to a non-existing request.

The mutation operator for duplicating actions (DA) suggests that a request (or a response) will appear unexpectedly. Finally, removing guards (RG) creates a test model in which mutated actions do not have guards, thus they are always enabled.

Despite none of the new mutation operators were fault revealing in our case study, they still provide new faulty behavior that cannot be created by any other mutation operators. Nevertheless, these mutation operators can deliver valuable mutants and should be investigated more.

H. Relation Between Mutations and Real Faults

To understand what kind of faults can be simulated by certain mutation operators, we compare the faults with the corresponding mutations. The first fault in Table IV is caused by the changing source of a transition (CS). It shows that deleting a resource was not properly implemented in the source code. The source code is implemented in such a way that it does not have the necessary condition of checking the availability of a resource, before deleting it. Such condition should always be included in the source code. Thus, CS can show lack of such condition by breaking the test sequences.

The second and third faults are caused by Change Guard Variables (CGV). The correlation between the faults and the mutation operator is straightforward: the operator changes the guards' values which simulate changing user's id in HTTP requests. Changing guards enables some transitions that otherwise are disabled.

VI. RELATED WORK

Due to the importance of testing security of web services, many studies and tools have been proposed in the literature. To draw the position of our research among the available studies, first, we describe related model-based mutation testing approaches, then we present some of the most related work on the model-based design and testing of web services and compare the available studies with our work, and then we focus on security testing approaches of web services.

A. MBMT approaches and improvements

MBMT has been gaining attention in software testing field, especially safety and security applications. Apart from theoretical research and experiments, one of the keys to making mutation testing available tool support. In a recent survey conducted by Papadakis et al. [30], available model-based mutation testing approaches are discussed. We only review some of the tools that are more similar to our work.

One of the well-defined tools is MoMuT that supports automatic mutation of different modeling languages, such as UML statecharts and Timed Automata [4], [3]. MoMuT for UML contains mutation operators for a large number of elements in UML. It only supports non-deterministic models. Thus test cases are linear sequences of inputs and outputs. MoMuT for TA uses UPPAAL TA models and applies timed input-output conformance (tioco) check between the specification and mutants. The conformance checking is done via SMT solver Z3.

Larsen et al. present an efficient technique of MBMT using Ecdar tool, which belongs to UPPAAL family [24]. The tool creates a strategy for refinement check for MBMT. The tool supports only deterministic models.

Belli et al. present MBMT with directed graphs using only two elementary mutation operators (insertion and omission) [11]. They describe the test algorithms and mutation generation technologies that are used to experiment with two different case studies. Their work is supported by a chain of tools including an event sequence graph (ESG)-based mutant and test set generator (MTSG). SIMULTATE is a toolset that uses fault injection technique on Simulink models to perform mutation analysis for certain parts of systems[32].

Several optimization techniques have been studied to reduce the cost of test execution. Devorey et al. presented featured mutants model (FMM) [15]. Their technique significantly reduces the time of testing by integrating multiple executions into a single one.

B. Modeling and Testing of web services

Model-driven development is a standard way of developing web services and can be used both for design and generation tests. Modeling web services has been presented by formal verification methods, model-checkers, specification languages, and theorem proving. Bozkurt et al. investigated a comprehensive survey on testing web services presenting available web service testing strategies [12].

Typically, web services are modeled with specification languages, simulated, verified and tested. Majority of the studies use model-checkers to verify the correctness of specifications, while some of the studies use models for verification as well as test generation. We review the studies that are similar to our approach.

Model-based testing of web services with symbolic transition systems (STS) is introduced by Frantzen et al. [18]. Belli et al. presented an event-oriented approach for MBT of a composite of web services in [10] and described expected behavior as a positive model and generated tests. They also defined some possible fault scenarios as a fault model which specifies undesired situations in communications of the web service composition and used for negative testing.

Modeling behavior of web services and their compositions are studied using UML [8], [28]. In order to automate the test generation, UML model of web services has been transformed into executable models. For instance, UML to UPPAAL TA transformation has been deployed in [16], [14] and [33].

In [33], we specified model of a web service composition using UML state diagrams and verified the requirements. Then, we transformed the model into UPPAAL TA models and generated the tests from the UPPAAL TA. In this study, however, we used the UPPAAL TA model to generate mutants to assess the vulnerabilities of the web services. Besides, in this paper, we focus on evaluating the authorization and authentication of web services.

C. Testing security of web services – different approaches

Security of web services is one of the fast evolving subjects in software testing. The reason behind such challenging issue is that the complexity of web services especially online social networks is growing fast. Mistakes in implementing and defining user credentials in web service are still one of the most common faults that are reported.

Several studies have been done on testing security in web services using fault injection, cross-site scripting (XSS) or modeling attacks scenarios. Salas and Martins presented a new approach to analyzing the robustness of Web Services by fault injection [35]. In their approach, attacks are simulated which can be used for evaluating the penetration on the web services. However, the process is not automated.

Dragoni and Massacci presented a framework which establishes communications between client and server based on the defined privileges as well as behavioral constraints [17]. They showed that the framework works as expected against both cooperative and malicious behavior.

A comprehensive analysis is done on all available mutation testing method presenting the current state of the art in this field and the open challenges [21]. Lee and Offutt [26] introduced an Interaction Specification Model which formalize the interactions among Web components. They defined a set of mutation operators for XML data model to mutate the inputs of the Web components. Li and Miller presented mutation testing methods using XML schema to create invalid inputs [27]. Mutation testing is extended to XML-based specification languages for Web services.

Lee et al. presented ontology-based mutation operators on OWL-S, which is an XML-based language for specifying semantics on web services [25]. They mutate semantics of the specifications of their case study such as data mutation, condition mutation.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we described three new improvements on our MBMT approach. Our approach includes the specification of a web service model in UPPAAL TA, generating mutants from the model, eliminating the trivial mutants and generating mutated tests. We presented the μUTA tool, which automates the mutation generation, mutation-selection, and mutation execution. As one of the improvements, we demonstrated how to select fewer, but suitable mutants by following mutation testing principles.

The second improvement of the approach was the demonstrating its relevance for the testing security of web services concerning multi-user context. As a consequence of intensive user collaborations in such services, user authentication credentials should be protected from malicious parties and adversaries. The dependability of social web services depends on offering privacy and security. We modeled and verified a Blog web service with two users including their privacy.

Lastly, we extended one of the previously defined mutation operators and introduced three new mutation operators to generate additional mutants. The evaluation of the approach showed that the mutation-selection criteria speed up the MBMT approach preserving the same quality of the test.

In future work, we plan to provide further experiments on web services and employ smarter mutations by combining primary mutation operators. Higher order mutations might be more efficient since infection of the first order mutations can be quickly discarded by other guards in the first place; thus they will not be selected for testing. Therefore, creating less restricted models using multiple mutations would help in generating stronger mutants.

Some of the problems that should be addressed in future work are scalability of test models and context-aware test generation. For large-scale web applications, which concurrent operations are prone to vulnerabilities, modeling and testing

the critical parts using the approach would be helpful. For these issues, we can focus on various test modularization principles such as aspects, contracts etc. and validating that part rather than including other parts of the system.

ACKNOWLEDGMENT

This work has been funded by the ECSEL MegaM@Rt^2 project. This project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737494. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Sweden, France, Spain, Italy, Finland and the Czech Republic.

We would like to thank Tewodros Deneke and anonymous reviewers for comments that significantly improved the manuscript.

REFERENCES

[1] The Open Web Application Security Project (OWASP). [Online; accessed 11-June-2018].

[2] M. Aboutrab et al. Specification mutation analysis for validating timed testing approaches based on timed automata. In *36th Annual IEEE Computer Software and Applications Conference, COMPSAC 2012, Izmir, Turkey, July 16-20, 2012*, pages 660–669, 2012.

[3] B. Aichering et al. Time for Mutants – Model– Based Mutation Testing with Timed Automata. In *Tests and Proofs*, pages 20–38. Springer, 2013.

[4] B. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. MoMuT::UML model-based mutation testing for UML. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–8, April 2015.

[5] B. K. Aichernig and E. Jöbstl. Towards symbolic model-based mutation testing: Combining reachability and refinement checking. *arXiv preprint arXiv:1202.6123*, 2012.

[6] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[7] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[8] C. Armstrong. Modeling web services with uml. In *OMG Web Services Workshop*, 2002.

[9] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[10] F. Belli et al. A holistic approach to model–based testing of Web service compositions. *Software: Practice and Experience*, 44(2):201–234, 2014.

[11] F. Belli et al. Model-based mutation testing - approach and case studies. *Sci. Comput. Program.*, 120:25–48, 2016.

[12] M. Bozkurt, M. Harman, and Y. Hassoun. Testing web services: A survey. 2011.

[13] BS 7799-3:2017. Information security management systems – guidelines for information security risk management.

[14] M. E. Cambronero et al. Validation and verification of web services choreographies by using timed automata. *Journal of Logic and Algebraic Programming*, 80(1):25–49, 2011.

[15] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P.-Y. Schobbens, and P. Heymans. Featured model-based mutation analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 655–666. ACM, 2016.

[16] G. Dıaz et al. Model checking techniques applied to the design of web services. *CLEI Electronic Journal*, 10(2), 2007.

[17] N. Dragoni and F. Massacci. Security-by-contract for web services. In *Proceedings of the 2007 ACM Workshop on Secure Web Services*, SWS '07, pages 90–98, New York, NY, USA, 2007. ACM.

[18] L. Frantzen et al. Towards model-based testing of web services. 2006.

[19] M. Grinberg. *Flask Web Development: Developing Web Applications with Python*. " O'Reilly Media, Inc.", 2014.

[20] A. Hessel et al. *Testing Real-Time Systems Using UPPAAL*, pages 77–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[21] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[22] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed i/o automata. *Synthesis Lectures on Distributed Computing Theory*, 1(1):1–137, 2010.

[23] K. G. Larsen et al. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[24] K. G. Larsen, F. Lorber, B. Nielsen, and U. M. Nyman. Mutation-based test-case generation with ecdar. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 319–328, March 2017.

[25] S. Lee et al. Automatic mutation testing and simulation on owl-s specified web services. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 149–156. IEEE, 2008.

[26] S. C. Lee and J. Offutt. Generating test cases for XML-based Web component interactions using mutation analysis. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 200–209, Nov 2001.

[27] J.-h. Li et al. Mutation analysis for testing finite state machines. In *Electronic Commerce and Security, 2009. ISECS'09. Second International Symposium on*, volume 1, pages 620–624. IEEE, 2009.

[28] E. Marcos et al. Representing web services with uml: A case study. In *International Conference on Service-Oriented Computing*, pages 17–27. Springer, 2003.

[29] R. Nilsson, J. Offutt, and J. Mellin. Test case generation for mutation-based testing of timeliness. *Electronic Notes in Theoretical Computer Science*, 164(4):97 – 114, 2006. Proceedings of the Second Workshop on Model Based Testing (MBT 2006).

[30] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman. Mutation testing advances: an analysis and survey. *Advances in Computers*, 2017.

[31] P. Pettersson. *Modelling and verification of real-time systems using timed automata: theory and practice*. 1999.

[32] I. Pill, I. Rubil, F. Wotawa, and M. Nica. Simultate: A toolset for fault injection and mutation testing of simulink models. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 168–173, April 2016.

[33] I. Rauf et al. An integrated approach for designing and validating rest web service compositions. In *10th International Conference on Web Information Systems and Technologies*, volume 1, pages 104–115. SCITEPRESS Digital Library, 2014.

[34] L. Richardson and S. Ruby. *RESTful web services*. O'Reilly, 2008.

[35] M. Salas and E. Martins. Security testing methodology for vulnerabilities detection of xss in web services and ws-security. *Electronic Notes in Theoretical Computer Science*, 302(Supplement C):133 – 154, 2014. Proceedings of the XXXIX Latin American Computing Conference (CLEI 2013).

[36] F. Siavashi et al. On mutating uppaal timed automata to assess robustness of web services. In *Proceedings of the 11th International Joint Conference on Software Technologies*, volume 1, pages 15–26. SCITEPRESS-Science and Technology Publications, 2016.

[37] F. Siavashi et al. *Testing Web Services with Model-Based Mutation*, volume 743, page 4567. Springer, 2017.