Requirements and Specifications for Adaptive Security: Concepts and Analysis

T. T. Tun[†], M. Yang[★], A. K. Bandara[†], Y. Yu[†], A. Nhlabatsi[§], N. Khan[§], K. M. Khan[§], B. Nuseibeh[†‡]

[†] School of Computing & Communications, The Open University, Milton Keynes, UK
[★] Department of System Management & Strategy, University of Greenwich, London, UK
[§] Computer Science and Engineering, KINDI Computing Research Centre, Qatar University, Qatar
[‡] Lero - The Irish Software Research Centre, University of Limerick, Limerick, Ireland

ABSTRACT

In an adaptive security-critical system, security mechanisms change according to the type of threat posed by the environment. Specifying the behavior of these systems is difficult because conditions of the environment are difficult to describe until the system has been deployed and used for a length of time. This paper defines the problem of adaptation in security-critical systems, and outlines the RELAIS approach for expressing requirements and specifying the behavior in a way that helps identify the need for adaptation, and the appropriate adaptation behavior at runtime. The paper introduces the notion of adaptation via input approximation and proposes statistical machine learning techniques for realizing it. The approach is illustrated with a running example and is applied to a realistic security example from a cloud-based file-sharing application. Bayesian classification and logistic regression methods are used to implement adaptive specifications and these methods offer different levels of adaptive security and usability in the file-sharing application.

CCS CONCEPTS

• **Security and privacy**; • **Software and its engineering**;

KEYWORDS

Security requirements, Self-adaptation

1 INTRODUCTION

Requirements for software systems are rooted in an environment characterized by complex runtime conditions. In adaptive software systems, some of the runtime conditions are difficult to predict and approximate until the system has been deployed and used for a length of time. This poses a challenge for requirements engineering because incomplete knowledge of the environment can lead to incorrect specifications and violation of critical requirements at runtime [27].

SEAMS '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5715-9/18/05…$15.00
https://doi.org/10.1145/3194133.3194155

For example, consider the class of security attacks known as parameter tampering where an attacker modifies the HTTP parameter values sent from the client computer to the server computer in order to escalate privileges. Much of the existing work on detecting and preventing parameter tampering attacks (such as [22] and [14]) uses domain-specific constraints by validating the parameter values both at the client side and the server side. For example, in order to prevent unauthorized "discount" attacks in e-commerce applications, defense mechanisms against parameter tampering ensure that the values for the quantity parameter are never negative when received by the server. These constraints are typically specified and implemented before the system becomes operational.

This paper considers cases where it is not possible to write rules for detecting parameter tampering at design time. For example, when various subsets of a group of users share documents using a web-based file sharing service (such as Dropbox, ownCloud, etc), one security requirement is to prevent self-inviting, namely that, a user should not be allowed to invite himself to a shared folder he does not own. If an attacker is able to intercept a share invitation and add his own ID as an invitee via parameter tampering (this attack scenario is discussed further in Section 2.1), it is difficult to specify at design time the necessary behavior to prevent the attack because the identities of the attacking and targeted users are yet unknown. Therefore, design time constraints are largely ineffective for addressing the class of security problems this paper is considering.

In order to address the problem, this paper proposes an approach for specifying adaptive security behavior under partial knowledge of the system environment and for incorporating further knowledge about the environment discovered at runtime into adaptive specification. In doing so, this paper makes the following three contributions. First, the definition and illustration of a notion of user-driven adaptation which allows for incomplete knowledge of the environment but the software and parts of its environment cooperate at runtime to adapt appropriately (Section 2). Conceptually, requirements are divided into episodic and run requirements, and informally, adaptation means the modification of episodic behavior in order to satisfy run requirements (Section 3). Second, the implementation of the proposed notion of adaptation at runtime by means of *input approximation*: whenever the system receives an input for which the required output (as demanded by requirements) is not certain, there is a need for adaptation. The adaptation behavior is identified by finding a similar input value for which there is a required output value, and to produce that output value (Section 3).

The paper proposes an architecture and a method for implementing input approximation using machine learning techniques. Third, the demonstration of how parameter tampering attacks against the file sharing application ownCloud can be identified and prevented at runtime using statistical machine learning methods in a scalable fashion (Section 4). In particular, the evaluation of Bayesian and logistic regression methods for identifying possible self-invites indicates that the higher degree of security (fewer missed attacks) comes at the cost of reduced convenience (more false alarms).

There is a substantial body of work on expressing and analyzing requirements for adaptation. For instance, Whittle et al. [25, 26] have proposed RELAX, which supports a fuzzy logic-based approach to analyzing uncertainty in adaptive systems. Ghezzi et al. [5] have examined the problem of when to introduce a new controller to a running system without restarting it, and verifying that the new controller meets its specification. Souza et al. [23] have proposed a method for weakening requirements as a way to adapt to the changing context. Existing work on partial model-based reasoning such as [19] has shown how uncertainty in model refinement can be reduced by defining conditions that need to be met during model transformation.

The proposed approach is different from the existing work on requirements and specifications for adaptation in two different ways. First, current approaches tend to view adaptation as a way of *avoiding or preventing* requirement violations based on design time, and perhaps runtime, knowledge of the environment. While such approaches are important, this paper takes the view that adaptation should also be *in response* to requirement violations encountered at runtime without weakening the requirements. Second, adaptation tends to be regarded essentially as an optimization problem in existing approaches, where the objective is to choose a system behavior that gives the highest possible level of requirements satisfaction. This paper posits that adaptation is not just as an optimization problem, but also a problem of collaboration between parts of the environment and the software: typically users in the environment are able to guide the software to adapt to a certain desired behavior.

The rest of the paper is organized as follows. Section 2 introduces a running example and the adaptation problem that motivates our work, together with the notions of requirements for security and adaptation. Section 3 introduces the proposed approach and illustrates its features with the running example. Section 4 presents an evaluation of the use of statistical machine learning approaches to implementing an adaptive specification for preventing the parameter tampering attacks. Related work is discussed in Section 5 and concluding remarks are given in Section 6.

2 PRELIMINARIES

This section introduces the running example, recalls the Problem Frames notation, and discusses the adaptation problem.

2.1 Parameter Tampering Attacks

The security problem discussed here is related to several related types of security attacks. One well-known type of attack is called the *parameter tampering attack* [22] where an attacker intercepts and modifies data (which has been usually validated on the client side) in order to gain unauthorized privileges on the server. Another type of attack is called *cookie poisoning* [12], where an attack modifies the values stored on the web browser in order to bypass checks or escalate privileges. There are also various types of *code injection attacks* [1] where it is the code rather than the data that has been inserted into communication in order to gain unauthorized access (for alternative names and real-world examples of these attacks, see [13]).

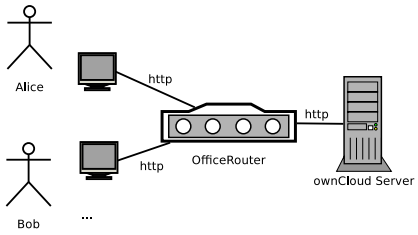


Figure 1: Network configuration

Scenario: In the scenario we are considering Alice, Bob, Sam and Tom work in an organization (Fig. 1). Different subgroups of them work on different projects, and they share documents using ownCloud, a cloud-based file sharing system. The lead person of the project creates a project-specific folder (inviter) and invites those who work on the project to share that folder (invitees). One security requirement of this application is that only those invited by the inviter are able to view the shared folder.

Attack: Bob knows that Alice, Sam, and possibly Tom also will be working on a secret project in the near future. He knows that he will not be invited to share the documents in the project folder, but he wants to see them nevertheless. So he writes a program that examines all HTTP requests on their office router to perform a parameter tampering attack as follows (Fig. 2). Bob's program running on the router looks for all HTTP requests for ownCloud document sharing where the inviter is Alice and invitees include Sam, and the program adds his own name to the invitee list. It means that whenever Alice invites Sam to join a folder, Bob also gets an invite, although it is not sent by Alice.

Following much of the existing work on detecting and preventing parameter tampering attacks (such as [22] and [14]), the scenario above assumes that the attacker can read the communication between the users and the server. In practice, this assumption can hold for a number of reasons including: (i) the web application does not use TLS/SSL, (ii) the attacker has compromised the router (through DNS hijacking for example), and (iii) the attacker has created an "evil twin" wifi access point [15].

In a parameter tampering attack, an attack may modify parameter values, field names, and the sequence of values communicated between the client and the server [22]. It is noted that in parameter tampering attacks, the attacker may have legitimate interactions with the system, and the attack itself may be disguised as one.

The general solution to parameter tampering attacks is to check and enforce static data integrity rules, both on the client side (typically via JavaScript programs for validating form data) and the server side (typically through code analysis [12]). This solution, however, does not apply in our ownCloud example: since Alice may

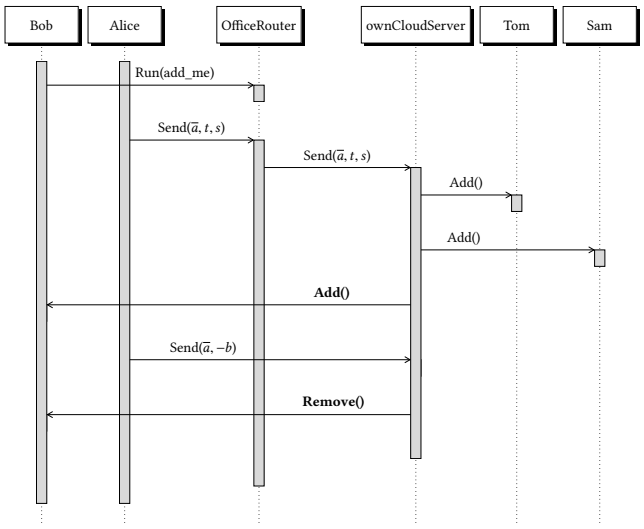


Figure 2: Attack sequence

sometimes want to share with Bob certain folders, Bob cannot be removed from invitee lists syntactically. Therefore JavaScript-based approach will work only if the information Bob inserts into the request string has distinct characteristics, such as the email address Bob uses in the attack being different from the one he uses for legitimate sharing. The problem here is that given a share request containing a group of invitees, we want to find out how likely is it that one of the invitees is an attacker (self-invited invitee).

Definitions: Let U be the finite set of users $\{a, b, s, t\}$ (abbreviations of Alice, Bob, Sam and Tom) who may create and share documents and folders using ownCloud. An inviter sends an invite to have other users to allow them access to a folder (for simplicity, we will leave out details about the folders being shared). The set of invites *Invites* is defined as a set of pairs of inviter (er) and some invitees (ee).

$$Invites = \{(er, ee) \mid (er, ee) \in U \times 2^U \ \wedge \ er \notin ee \wedge ee \neq null\}$$

Notationally, we will write an invite as comma separated values where the inviter has an overline. So, the request where *Alice* invites *Sam*, *Tom*, that is the relation $(a, \{s, t\})$, is simplified as $\overline{a}, s, t$.

When the server receives such an invite, all invitees are notified and they are able to access the shared folder (without having to accept the invitation to share). We assume that the inviter cannot be one of the invitees.

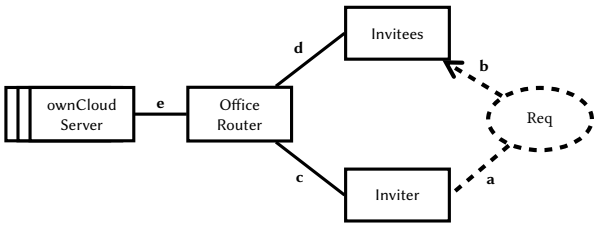
Similarly, an inviter sends an uninvite to have someone removed from the invitee list of a folder.

$$Uninvites = \{(er, ee) \mid (er, ee) \in U \times U\}$$

Again, we will write $\overline{a}, -b$ to say *Alice* uninvites *Bob*.

2.2 The Problem Frames Notation

Fig. 3 shows the ownCloud sharing problem using the Problem Frames notation [8] and the diagram should be read as follows. At the interface c, the phenomena description ER!{Send(invite), Send(uninvite)} means that the problem domain Inviter controls



Interface	Phenomena
e	OR!{Send(invite), Send(uninvite)}
	OCS!{Add[U], Remove[U]}
c	ER!{Send(invite), Send(uninvite)}
d	OR!{Add[U], Remove[U]}
a	ER!{share request[EE], unshare request[U]}
b	EE!{invited[U], uninvited[U]}

Figure 3: Problem Diagram: Sharing in ownCloud

the events Send(invite) and Send(uninvite), and these events are observed by the problem domain Office Router. Similarly, at the interface e, Office Router controls the events Send(invite) and Send(uninvite) that are observed by ownCloud Server, and the events Add[U] and Remove[U] controlled by ownCloud Server are observed by Office Router. Events at the interface d can be read in the same way.

The behavior of Office Router is to send an invite request to ownCloud Server whenever a request is received. The behavior of Inviter is such that when they want to share a folder with some other users (share request[EE]), they send invites via OfficeRouter to ownCloudServer. Occasionally, when someone needs to be removed from the invitee list, the inviter sends an uninvite. A sequence of one invite by a user (namely at OfficeRouter), optionally followed by one uninvite is called a *share episode* which is defined as:

$$SE = \{\langle i; u\rangle | i \in Invites \wedge (u \in range(i) \vee u = null)\} \quad \text{(SE)}$$

For example, the share episode $\langle \overline{a}, b, s; b\rangle$ says that there is an invite, where Alice invites Bob and Sam (invitees), and b was subsequently removed by a from the list of invitees. In another episode $\langle \overline{a}, b; -\rangle$, the inviter is a and the invitee is b, and no-one was uninvited.

A *requirement* is a desired relationship between the variables *referenced* and *constrained* by the requirement, which in this case is the relationship between share request, unshare request at the interface a and shared and unshared at the interface b. More specifically, the requirement is a set of pairs of share requests and invited users, where only those users in the share requests are invited:

$$Req \triangleq \{(share\ request[EE], \{invited(u_1) \ldots invited(u_n)\}) | EE = \{u_1 \ldots u_n\}\}$$

2.3 The Adaptation Problem

It is difficult to write design-time constraints for when to remove a user Bob from an invitee list and when not. We observe that this difficulty is related to two issues. First, since the machine ownCloud Server cannot observe the requirement variables such as EE in the

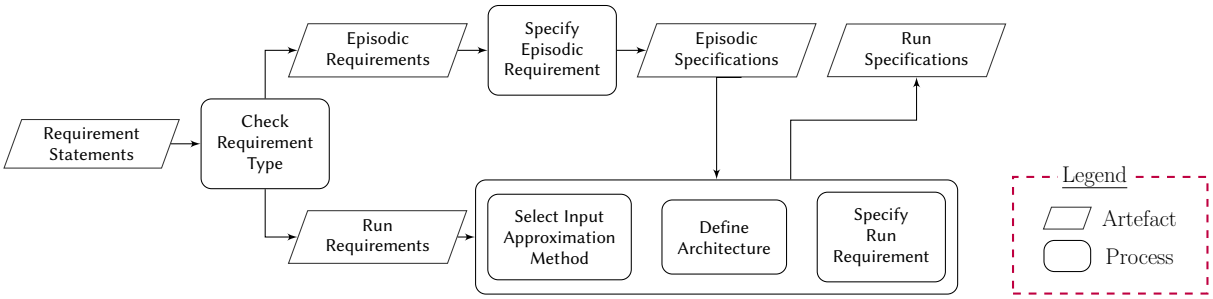


Figure 4: Overview of the RELAIS approach

phenomenon ER!share request[EE] directly, but must use the variable OR!Send(invite) instead, there is a *gap between the specification phenomena and the requirement phenomena*. Second, there is some *uncertainty about the behavior of the environment*, and in particular about Office Router. In the relationship between ER!Send(invite) and OR!Send(invite), for example, the router cannot guarantee that the invitee lists at the two interfaces are the same. The question of how unreliable a particular office router is cannot be known before the system becomes operational. Given these two issues, it is difficult to write a weakened specification at design time that will satisfy the requirement.

3 THE RELAIS APPROACH

This section discusses and illustrates the key concepts of the RELAIS (Requirements Engineering Language for Adaptive Information Security) approach. We will define these concepts independently of existing requirements engineering languages (in the style of [17]), and exemplify them before presenting the Problem Frames-based syntax. Alternative syntaxes, such as those based on goal- and agent-oriented requirements notations, are also possible and are further discussed in Section 5.

The RELAIS approach puts an emphasis on the distinction between *episodic* and *run* requirements. In a reactive system, the machine is often required to cause a sequence of event occurrences, perhaps within a time constraint, in response to each occurrence of a certain environmental event sequence. An episodic requirement demands that the corresponding response event sequence for each environmental event sequence has a certain property. An environmental event sequence followed by the corresponding response sequence is called one *behavioral segment*. For example, the sequence of observations projected on the variables share request and Invited, $\langle sharerequest(\{s,t\}), \{Invited(s), Invited(t)\}\rangle$ is one behavioral segment, and its property satisfies the requirement Req.

A *run requirement* demands that several behavioral segments of a requirement has a certain property. Two examples of an run requirement are as follows: "the number of unshare request must reduce over time", and "the ration of unshare requests to share requests must be less than 1 to 10 at all times".

The episodic and run requirements are similar in the sense that they are properties of the system behavior and that they can be violated by certain system behaviors. One main difference between these two types is the span: an episodic requirement is a property of a single behavioral segment, while a run requirement is a property of several behavioral segments.

Adaptation in the RELAIS approach means the modification of the episodic behavior in order to satisfy run requirements. For example, in order to ensure that an unshare request does not happen for Bob, the specification of ownCloud Server is modified so that Bob is not added to the share folder. One way to achieve this modification of the behavior is by modifying the value of invite at the interface e by removing Bob. We call this type of mechanism "input approximation" because it is a function that attempts to bridge the gap between the requirement and specification phenomena (Section 2.3) by approximating the mapping between the invite at the interface c to the variable at interface e.

As shown in Fig. 4, our approach begins with the separation of requirements into episodic and run requirements. Episodic requirements are first specified. An appropriate input approximation method is selected and an architecture is selected before a run requirement is specified. In our discussion of these concepts below, we will use the notion of *observation*. As used in the statistics literature [2], an observation refers to a recorded value of a variable of a simple or complex data type.

3.1 Observations of the system

We will describe the behavior of system $\mathcal{S}$ as a totally ordered finite set of observations of the vector $\mathcal{V}$ of typed variables $(f_1, f_2 \ldots f_n)$. The variables $f_1, f_2 \ldots f_n$ in the vector are of specific types such as boolean, numeric, textual and so on. For example, the vector (share request[EE], ER!Send(invite), OR!Send(invite), OCS!Add[U], OR!Add[U], invited[U]) contains all the variables to describe the behavior of the ownCloud system in Fig. 3. In this system, the following sequence of observations describes a behavior in which *Alice* invites *Tom* and *Sam* (*na* means not available):

$$\begin{aligned}&\langle(share\ request[s,t], na, na, na, na, na);\\ &(na, Send(\overline{a}, s, t), na, na, na, na);\\ &(na, na, Send(\overline{a}, s, t), na, na, na);\\ &(na, na, na, \{Add[s], Add[t]\}, na, na);\\ &(na, na, na, na, \{Add[s]; Add[t]\}, na);\\ &(na, na, na, na, na, \{Invited(Sam), Invited(Tom)\})\rangle\end{aligned} \tag{B1}$$

Partial observations of $\mathcal{V}$, known as projections of $\mathcal{S}$, are described by subscripting the positions of values in the vector. The relation between full and partial observations is surjective. The first

of the five observations above, and indeed all five observations, can be projected on the second variable as Send($\overline{a}, s, t$). Similarly, the entire sequence of the six observations in (B1) can be projected and simplified as:

$$\langle share\ request[s,t]; Send(\overline{a},s,t); Send(\overline{a},s,t); Add[s], Add[t];$$
$$Add[s], Add[t]; Invited(Sam), Invited(Tom)\rangle$$

Notice that since we are not dealing with real-time constraints in this work, the sequencing of observations only indicates temporal ordering. However, time constraints can be handled by including the clock as one of the observed variables [17].

From the point of view of the machine, the variables in the vector can be characterized as follows. The machine can read values of (or observe) some variables in vector $\mathcal{V}$ (such as OR!Send(invite)), and can assign values to (or control) some of the variables (such as OCS!Add[U]). The vector may also contain variables that the machine can neither read from nor write to (such as OR!Add[U]. The variables in the vector may have causal and logical relationships between them but those relationships are not defined explicitly in the vector.

3.2 Episodic Requirements

All possible sequences of observations is denoted as $S^* \subseteq 2^S$. An episodic requirement is a relation between two sequences of observations, $R_{epi} \subseteq S^* \times S^*$, where the domain is a sequence containing observations of referenced variables, and the codomain is a sequence of observations of constrained variables. Since we will typically project sequences of observations, we will write an episodic requirement as relation between two projected observations. For example, the relation ($\langle share\ request[Sam, Tom]\rangle_1$, $\langle Invited(Sam), Invited(Tom)\rangle_6$) says the values of the variables recorded, without saying anything about values other variables may have in the behavioral segment.

Episodic requirements may be violated when the environment does not have the necessary property that the invite list does not change after the inviter has sent it.

Given some behavior segments such as the following,

$$\langle share\ request[s,t]; Send(\overline{a},s,t); Send(\overline{a},s,t); Add[s], Add[t];$$
$$Add[s], Add[t]; Invited(Sam), Invited(Tom)\rangle$$

we can say whether they satisfy or violate a given requirement. The predicate Satisfy1 will be used to say this more precisely, where the subscripts r and c represent the projection of the segment to the referenced and constrained variables of the requirement:

$$\text{Satisfy1(seg, Req)} \triangleq (seq_r, seg_c) \in Req \qquad \text{(Sat1)}$$

Negation of the predicate Satisfy1 means that the segment does not satisfy the requirement.

3.3 Run Requirements

A property of several segments with respect to a requirement is called a run requirement. The predicate Satisfy2 can use used to say this more precisely.

$$\text{Satisfy2(segs, Req)} \triangleq \exists s \in segs \cdot \text{Satisfy1(s, Req)} \qquad \text{(Sat2)}$$

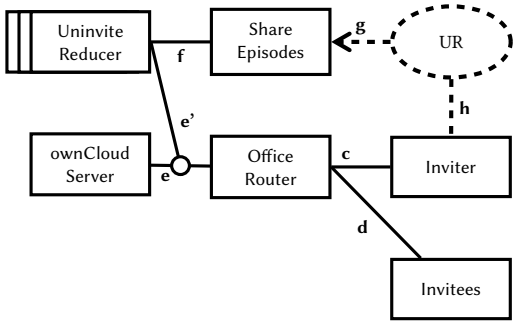
Negation of the predicate Satisfy2 means that there is no segment in the set segs that satisfies the requirement. We will use the operator # in #Satisfy2(segs, Req) to count the number of segments in *segs* satisfying the requirement *Req*; similarly, #¬Satisfy(segs, Req) is the number of time *Req* is not satisfied in *segs*.

For example, assuming *owc* contains all behavioral segments in the ownCloud system, #¬Satisfy2(owc, R) = 0 says that the requirement Req must never be violated. Similarly, a weaker form of a run requirement can be expressed as a relative number of the requirement violation. For example,

$$\frac{\#\neg\text{Satisfy2(owc, R)}}{\#\text{Satisfy2(owc, R)}} \leq 0.1$$

says that the requirement R must not be violated for more 10% of the time it is satisfied. Therefore, even when an episodic requirement is violated from time to time, the corresponding run requirement can still be satisfied. In this sense, run requirements are regarded as second-order.

In the ownCloud example, every share episode where someone is uninvited is a segment of behavior that violates the security requirement, and every share episode where no-one is uninvited is a segment of behavior that satisfies the security requirement. The adaptation requirement in this example is to minimize the number of times the security requirement is violated, meaning that the number of times a inviter has to uninvite an invitee reduce as more share requests are processed by the system. This adaptation requirement is intended to improve security.



Interface	Phenomena
e'	UR!{Send(invite)}
	OR!{Send(uninvite)}
f	UR!SE}
g	SE!{Reduce over time}

Figure 5: Problem Diagram: Adaptation in ownCloud

3.4 Adaptive Specification

When an episodic specification can violate its requirements from time to time, the behavior of the specification can be improved by introducing a new adaptive specification while extending the problem world. Fig. 5 shows a small extension to the Problem Frames syntax for writing run requirements. Notice that the lower half of the diagram contains the same domains as in Fig. 3. One way to

read the insertion of Uninvite Reducer between ownCloud Server and Office Router is in terms of pipes and filters (as in Unix-like OSs). Uninvite Reducer acts as a filter, where the interface e is the pipe containing some parameter values.

The intuitive idea is that at the point of the circle, Uninvite Reducer intercepts all send events controlled by Office Router, and Uninvite Reducer can manipulate the values of invite before it is visible to ownCloud Server. Through the interface f, Uninvite Reducer maintains all share episodes in the system. Here, Share Episodes captures the fact that reports of requirement violation by users by means of unshare request is used to guide the adaptation. This is related to that of requirements monitors in the ReqMon framework [18], but is different in the sense that it is not a software instrument and that it is intended to guide the adaptation.

The adaptive specification Uninvite Reducer can be described as a function that, given the history of invites and uninvites, calculates the probability of a user being uninvited later by the inviter. More specifically, a valid specification of Uninvite Reducer for such a requirement needs to answer the question $\langle \overline{a}, b, s, t; ? \rangle$: that is when a invites b, s and t, how likely is it that any of the recipient is later uninvited (i.e. a self-invited user)? The possible outcomes are: – (no-one), b, s or t. The specification will then remove the likely self-invited user from the share request so that the inviter does not have to uninvite the user later.

In a realistic setting, the number of share episodes will be large, and the uninvite data is noisy because some of the uninvite events may be due to mistakes by the inviter rather than attacks by a malicious user. The next section considers the use of statistical methods for implementing and evaluating Uninvite Reducer.

4 EXPERIMENTAL EVALUATION

This section describes the use of Bayesian classifiers and the logistic regression method to implement Uninvite Reducer and evaluate the performance of both approaches using data generated by simulations of share episodes in ownCloud. In practice, such data is available from the access logs on the server.

4.1 Implementation Using Statistical Methods

In order to implement Uninvite Reducer, we can construct a conditional probability model for a share request sr to find the user among the invitees of the request mostly likely to be uninvited. This can be cast as a Bayesian classification problem for ordinal class labels or a logistic regression problem.

Let $x_1, ..., x_n$ be the vectorisation of the user set (that is representing each member of the set by a binary variable in the vector), where $|U| = n$. Let $\mathcal{X}$ be the merger of two user vectors $x_1, ..., x_n, x_{n+1}, ..., x_{n+n}$, so that the first n number of variables represent the inviter (therefore, exactly one variable is true in the group), and the rest are for invitees (therefore, in this group of variables, the inviter is always false, and one or more other variables are true). Let the class labels $\mathcal{Y}$ be the enumeration $|U|$, so that $y_1, ..., y_n$ denotes the user names. The invitee most likely to be uninvited in a share request, $\hat{y}$, can be calculated as:

$$\hat{y} = \underset{i \in (1..n)}{\arg\max} \Pr(y_i) \prod_{j=1}^{2n} \Pr(x_j | y_i) \tag{1}$$

ALGORITHM 1: Generate Share Requests

Input: $nobs$ and $nuser$ representing numbers of observations and users respectively.
Output: A share request matrix.

```
1  m ← nobs; n ← nuser;
2  inviters ← matrix (m, n);
3  invitees ← matrix (m, n);        // Empty matrices with m rows and n columns
4  sspace1 ← (1, 0_1, ..., 0_{n−1}); // A vector for sample space containing one 1 and
     n-1 number of 0s
5  for i ← 1 to m do
6      inviters[i, ] ← rand_perm(sspace1); // Random permutation of the
         sample vector
7  end
8  invitees ← inviters;
9  for j ← 1 to m do
10     if invitees[j, k] == 1 then
11         invind ← k ;          // invind stores the index of the inviter in the row
12     end
13     sspace2 ← (1, sample(binary, n − 2)) ; // A vector with one 1 and n-2
         number of random binary values
14     tmp ← rand_perm(sspace2);
15     newrow ← append(tmp, 0, invind) ; // Ensure that the inviter is not
         an invitee
16     invitees[j, ] ← newrow;
17 end
18 share_requests ← column_bind(inviters, invitees);      // Join the
     matrices side by side
```

An alternative to the naive Bayes approach is logistic regression [7], commonly used when the dependent variable has only two possible values. In our attack scenario, the dependent variable is the label for potential attackers, denoted as z_i, which has two possible values: a user y_i is an attacker ($z_i = 1$) or not ($z_i = 0$). The independent variables are the inviter and invitee vectors $x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+n}$. Let p be the probability of an event $z_i = 1$, and thus, $1 - p$ is the probability of $z_i = 0$. The logistic regression model can be built as

$$\log \frac{p}{1-p} = \beta_0 + \beta_1 x_1 + \cdots + \beta_{n+n} x_{n+n} \tag{2}$$

where log is the logarithm and β_i is the coefficient for each independent variable x_i.

4.2 Data

Algorithm 1 is used to generate a matrix containing a given number of share requests (*nobs*) for a given number of users (*nuser*). There are two parts in the matrix (*share_requests*): for the inviters part (*inviters*), each user is represented by a binary variable, but only one of them can take the value 1, and the rest must take the value 0 (only one inviter in every request). The algorithm achieves this by performing a random permutation of a vector with appropriate values (Lines 4–7). For the invitees' part (*invitees*), there are two constraints to satisfy: the inviter must not be an invitee, and the number of invitees per share request must be greater than zero. The algorithm achieves this by first recording the index of the inviter in each share request (Lines 10–12), and filling the rest of the columns with random binary values, before inserting 0 at the position of the

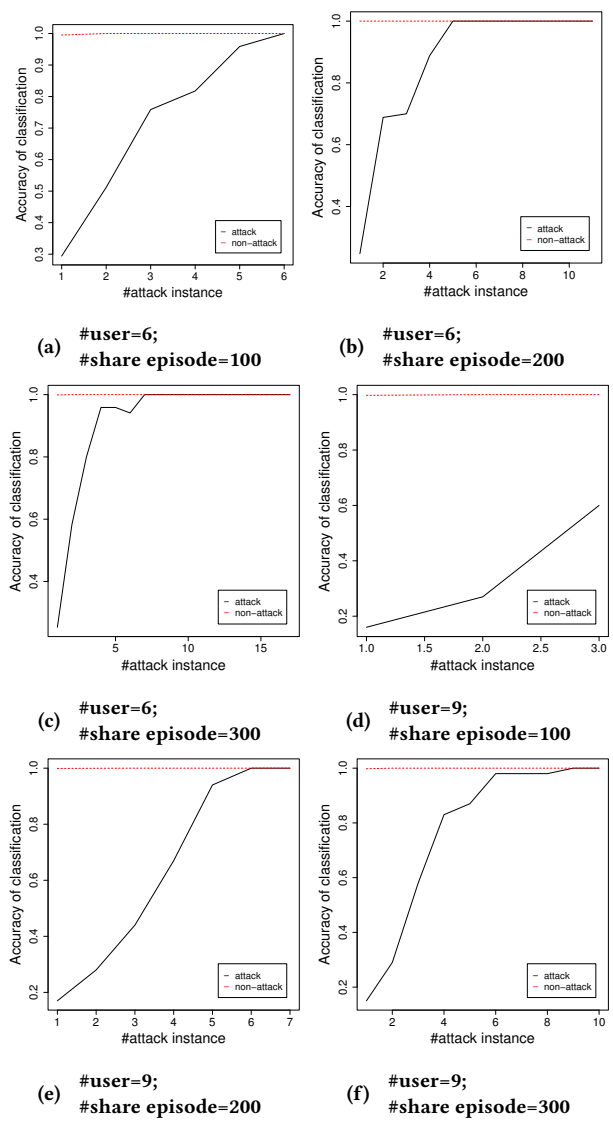


Figure 6: Performance of Naive Bayes Classifiers

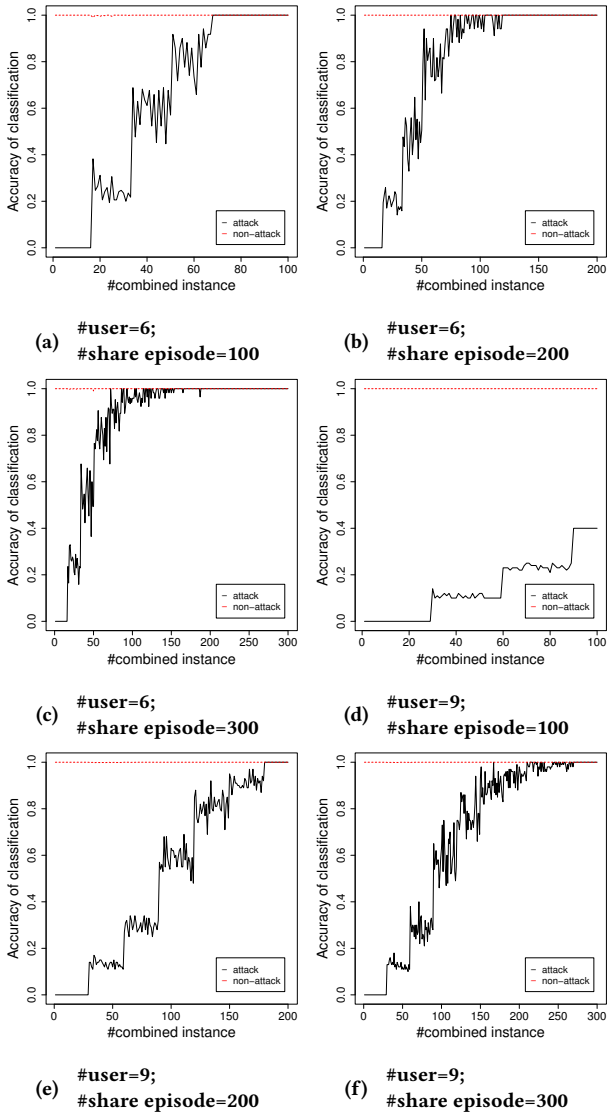


Figure 7: Performance of Naive Bayes Classifiers as the number of share requests increases

inviter. The algorithm ensures that there is at least one invitee by adding one 1 in the sample space (Lines 13–16). The *inviters* and *invitees* matrices are then bound side by side[1].

4.3 Labelling

Having generated the share request data, we assign the labels indicating whether each observation is an attack or non-attack. Since the question of any user being an attacker is a binary classification problem, class labels 0 and 1 are used for non-attack and attack respectively. In the training data, we assign the label 1 to every observation where a particular inviter is 1, and some invitees including the attacker are also 1. All other observations get the label 0. In the generation of 100 observations with six users, the number of uninvite cases (label 1) is around 6% (the percentage changes when the number of users changes). So the balance between the two labels is hugely in favor of non-attack, which is realistic because in practice the number of attacks is relatively low. This issue of class imbalance is well known [9]. Therefore, it is not meaningful to look at the overall accuracy of classification. We will instead examine the accuracy for each class in order to avoid potential bias [21].

4.4 Results

Fig. 6 shows the performance of naive Bayes classifiers as the number of share requests, as well as the number of episodes where the attacker is uninvited, increases. In each run, x number of attack

[1] All our code in R is available from `https://github.com/ttt23/SEAMS-2018`.

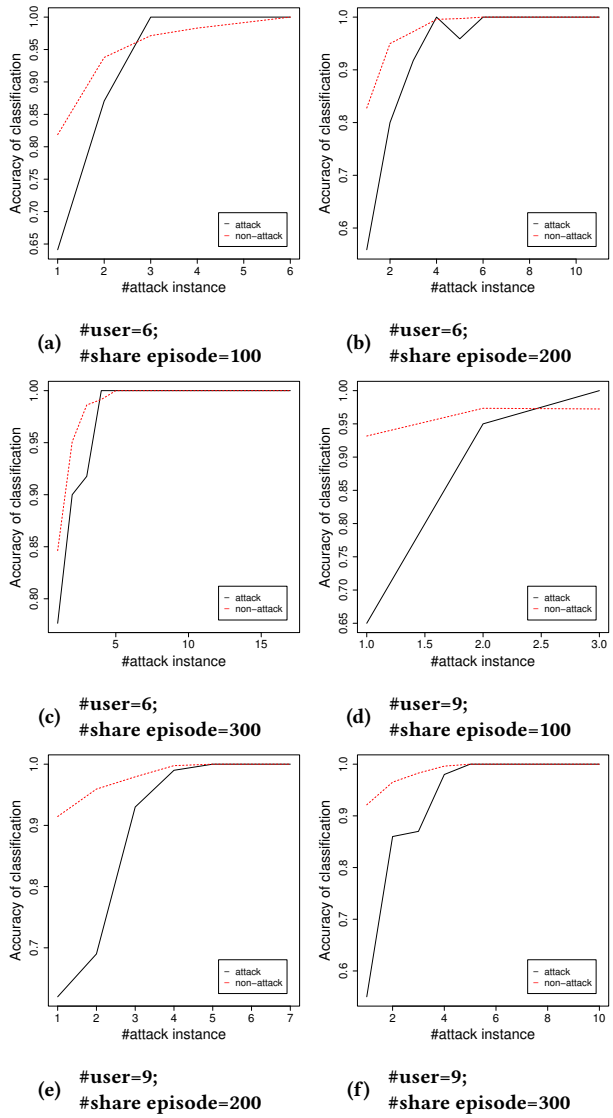


Figure 8: Performance of Logistic Regression Classifiers

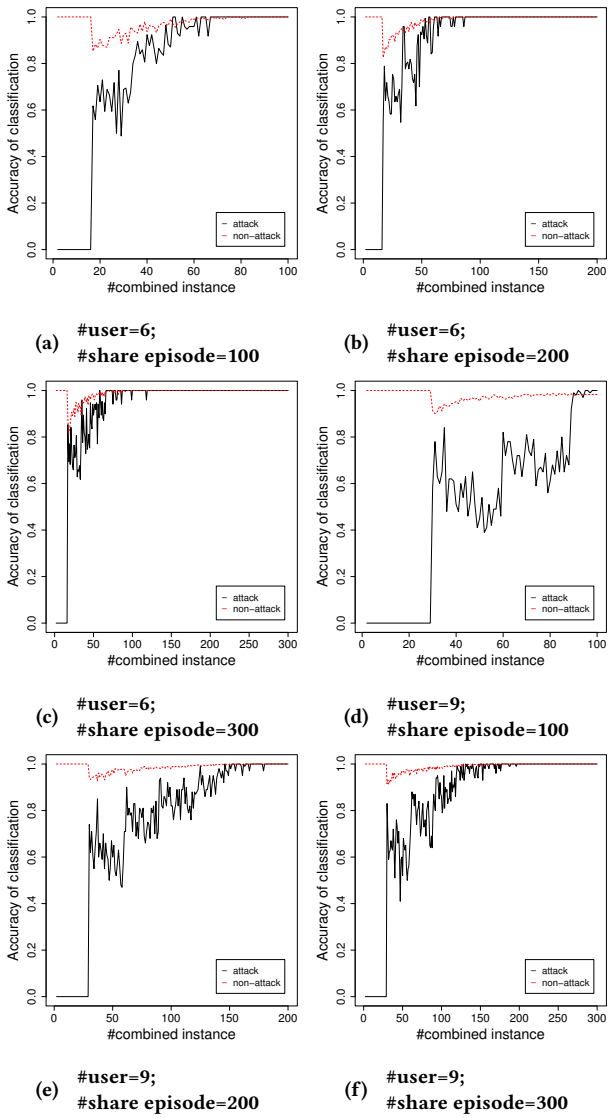


Figure 9: Logistic regression performance as the total number of observations increases

cases are randomly selected, together with a proportional number of non-attack cases according to the class balance. The figure for each run is computed by sampling 10 times from the same dataset of 100 observations (in the style of 10-fold cross validation [10]). Therefore, for Fig. 6a for example, we have constructed and tested 60 classifiers. When constructing a classifier, the entire dataset is used for testing in every case.

As Fig. 6a shows, the accuracy for classifying attack cases starts from around 30% and increases with the attack cases. As the classifiers observe more attack cases, the accuracy increases and achieves 100% when six attack observations are made. That is, in order to identify an attacker correctly, a classifier needs to see at least six uninvite requests if the number of observations is 100. In contrast, the accuracy for classifying non-attack cases is always 100%. This is because the classifiers observe only one attack case but several non-attack cases at the beginning, and are thus biased towards the non-attack cases. In practice, it means that the users of this system will not get false positives, but some attack cases will be missed initially when there are not enough attack cases to sample. As Fig. 6b and 6c show, the accuracy of classifying attack cases is close to 100% when the number of attack cases increases to five.

Fig. 6d—6f show the changes in the accuracy as the number of users increases to nine, and as result the class ratio changes to around three attack cases in 100 observations. The accuracy of classifying attack cases is comparable in cases of six users and nine users if the accuracy per number of attack cases is considered. Both 6e—6f show that accuracy for attack cases is close to

100% when the observations include six attack cases. In cases of nine users, the accuracy for classifying non-attack cases is also perfect, because there is always a sufficient number of non-attack observations even when the number of attack cases is one.

Fig. 7 shows the performance of naive Bayes classifiers as the total number of observations increases over time, again according to the class ratio. In each run, x number of observations are randomly selected from the same dataset of 300 observations, where the balance between attack and non-attack cases is kept at 6%. The accuracy of classifying non-attack cases hovers around 100%, while the accuracy of classifying attack cases gets better slowly. The increase in the number of users makes the classifiers generally less accurate for the security cases.

We now construct logistic regression classifiers by using the model described in Section 4.1 and the same dataset generated in the naive Bayes experiments. Fig. 8 shows the performance of the logistic regression approach when the numbers of users and overall observations vary. Unlike the naive Bayes classifiers, the accuracy of the logistic classifiers for classifying both non-attack and attack cases starts from around 85% and 60% respectively (Fig. 8a—8c). The accuracy gets better as the number of attack observations increases. In order to identify both an attack and non-attack cases correctly, the classifiers need to observe at least four attack cases and around 64 non-attack cases. Increasing the number of users (Fig. 8d—8f) has little impact on the performance when the number of attack cases are comparable.

Finally, Fig. 9 shows the performance of logistic regression classifiers when the total number of instances increases according to class ratio. It shows that accuracy for attack cases is zero until around 17 instances are observed when the number of users is six (Fig. 9a—9c), suggesting that the classifiers need to see at least two attack cases (similar to Fig. 7). The same is true when there are nine users. Accuracy for both cases improves as the number of combined instances increases.

In general, it shows that both approaches perform well for detecting potential attack cases. When there is a short history of uninvite, Naive Bayes is generally good at identifying non-attack cases but is poor at identifying attack cases. On the other hand, the logistic regression method can identify attack cases with high accuracy even when there are few attack observations. The downside is that these classifiers also give a higher number of false positives, as some of the non-attack cases are incorrectly classified as attack cases, especially when the accuracy for the attack cases picks up.

5 RELATED WORK

Since this paper is primarily about requirements and specification of systems with adaptive security behavior, the work discussed here is related to the areas of requirements engineering for adaptive security systems, the use of statistical approaches to security and the detection and prevention of parameter tampering attacks. Existing research on the modeling of security requirements, threats, and requirements evolution is not directly related to this work and is therefore not covered here.

5.1 Adaptation

There are several related notions around the term adaptation including *context-awareness*, and *self-adaptation*. Many of these notions and their significance to research have been discussed many times previously (for instance [20]). Instead of repeating the discussion, we will focus on how the term adaptation is used.

Perhaps the earliest example of adaptation is PID controller, which uses feedback from the environment to compute deviation from the desired target value and applies correction when necessary. The notion of adaptation described in this paper is similar to that. However, our focus is on the requirements, and how they can be structured. Broy et al. [3] give a definition of adaptation, which allows us to distinguish adaptive behavior from non-adaptive behavior. According to their definition, a system is *non-adaptive* if its behavior is determined exclusively by the user input. A system is *adaptive* if the output is determined both by the input values the user provides to the system, as well as other values (they are called indirect/implicit inputs) available to the system, which the user may or may not be aware of. Therefore, from the user's point of view, the machine may appear non-deterministic, although the system is not actually non-deterministic, since it uses implicit input values to determine the behavior. The implicit values are part of the context. Adaptation is *non-transparent* if the user cannot observe the context that affects the machine behavior; *transparent* if the user can observe part of the context but not change it; and *diverted* if the user can control part of the context. The main limitation of this definition is that adaptation is fundamentally about the user's perception of how their input is processed by the machine. Critically, in their definition a user does not guide the system from exhibiting unwanted behavior to exhibiting wanted behavior. Furthermore, their notion of adaptation does not distinguish between those parts of the environment that may cooperate and those that may not when adapting, which is critical in security problems.

5.2 Requirements Engineering Approaches to Adaptation

RELAX [26] proposes a way of writing declarative requirements in order to allow for environmental uncertainty. To achieve this, requirements engineers first separate requirements that must be satisfied at all times (invariants) from requirements that do not have to be satisfied under certain environmental conditions. Requirements in the latter category are rephrased using special operators to indicate the fact that they may not be fully satisfied, if at all, from time to time, due to some uncertainty about the environmental properties. Those operators include AS MANY AS POSSIBLE, and AS EARLY AS POSSIBLE. What the RELAX language and the approach provide is a way of weakening requirements at design time so that the specifications allow more behavior that can satisfy the requirements under different environmental conditions. In our example, it means rewriting requirements such as Req by adding operators for weakening it. However, such weakened requirements are in general non-deontic (non-obligatory) in the sense that the system is no longer obliged to satisfy them at all.

Conceptually, the improvement we have brought over the RELAX approach is a way of relating run requirements with episodic requirements. As a result episodic requirements are still binding,

and that adaptation is achieved not just by relaxing, but by allowing the specification for run requirements to modify the specification of the episodic requirements. The user is expected to help 'correct' the system behavior over time through error reporting. User participation is therefore fundamental to our notion of adaptation. Unlike our approach, RELAX provides no implementation architecture for adaptation, and as a result it is not clear how the user might (not) be involved in the adaptation process, and how the system may change its behavior if the user can help identify unwanted behavior. Finally, RELAX allows uncertainty in both the input values and the output values (system actions). We are currently restricted to uncertainty in input values but not in output values.

In the ADAM approach for handling uncertainty in non-functional requirements, such as response time and usability, there are two main levels of abstraction [6]. In the modeling phase, they first describe the "abstract functionalities" of a system, i.e. the system implementing the functional requirements, using a workflow, such as a UML activity diagram. Each abstract functionality may have one or more concrete implementations. Each implementation has annotations of their impact on non-functional requirements. For example, the "product lookup" abstract functionality may be implemented by an API call to searchupc.com, or through manual input of the user. For each implementation, there will be different values for response time, usability and energy consumption. The problem is to find a composition of the concrete functionalities at runtime that give the best satisfaction of all non-functional requirements.

There are some similarities with our approach, such as the probabilistic characterization of the environment, and incompleteness of knowledge about the environment at design time. One key, perhaps complementary, difference is that we are concerned with finding the most likely correct input value, before choosing the output value that is already associated by the requirement with the input value. In other words, once the input is known, the output is certain. In their work, that is not the case. They are concerned with choosing output values that will satisfy non-functional requirements to the best extent possible. However, unlike in the RELAIS approach, their specifications of non-functional requirements do not modify the behavior of concrete functionalities: they simply choose the best configuration of the pre-defined behaviors.

The notion of awareness requirements [23, 24] has also been used to describe the requirements for adaptation and evolution in software systems. Awareness requirements are about the success and failure of other requirements. An awareness requirement for example can state that a particular requirement must never fail. Similarly, it can also state the ratio of success to failure, or state the trend over time and so on. Patterns of awareness requirements have been presented in [24].

There are some similarities between their work and the RELAIS approach. Their notion of an awareness requirement is similar to the notion of run requirements, but our treatment of run requirements as a property of behavioral segments is more precise. Both episodic and run requirements in the RELAIS approach are properties of system behavior, rather than properties and properties of properties. More importantly, run requirements are not just about expressing the desired properties, but also about modifying the behavior of episodic specifications in order to adapt.

There are several architectural approaches to dealing with adaptation [11, 16, 28]. An important theme in this line of work is the description and analysis of how components and their connections may change at runtime in response to environmental conditions. The general architecture for adaptation used by RELAIS is similar to the wrapper architectural style. The RELAIS approach also emphasizes the role of requirements and how they can be structured in order to highlight the need for adaptation at runtime.

5.3 Statistical approaches for security

Anomaly detection techniques [4] have been applied extensively for improving security (such as in intrusion detection in computer networks), fraud detection in banking systems and so on. Classification techniques are frequently used when partial labels are available. In this work, we have shown that classification techniques can be used for adaptation as well as for security.

5.4 Detecting and Preventing Parameter Tampering Attacks

Much of the existing work on parameter tampering attacks use syntax-based approaches to detect and prevent potential attacks. These approaches include static analysis of design for potential weaknesses [12], and dynamic analysis to prevent attacks [22], and they tend to assume that the attackers are not insiders, and therefore have not legitimate access to the systems. In our attack scenarios, we assume that attackers are insiders, and are therefore more difficult to detect. As a result, the detection of potential attack cannot be sound: it will depend on the availability of good training data. Having said that we have shown that generic classification algorithms can learn to accurately classify the attacks very quickly.

6 CONCLUSION

This paper has proposed a notion of runtime adaptation for security-critical systems, where knowledge about the environment is partial at design time. The challenge is to use knowledge discovered at runtime to identify appropriate adaptation behavior in order to improve the system security. We have described the input approximation method for adaptation, and how the method can be implemented using constraint-based program methods and statistical machine learning techniques. Unlike existing approaches that aim to prevent security breaches at runtime, the RELAIS approach aims to exploit knowledge about the environment discovered at runtime (during failures), and use the knowledge to identify appropriate adaptation behavior in the future. The proposed approach is particularly useful, both conceptually and practically, when dealing with parameter tampering attacks where it is not possible to design all the defense mechanisms at design time. Having said that, we suggest that adaptation via input approximation is a general concept that can be applied to any system whose behavior is determined by its input values and we plan to explore this in future work.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for insightful comments and suggestions, and Michael Jackson for guidance and encouragement. This work is supported by SFI grant 13/RC/2094, QNRF NPRP 5-079-1-018, and ERC Advanced Grant no. 291652 (ASAP).

REFERENCES

[1] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*. ACM, New York, NY, USA, 281–289. https://doi.org/10.1145/948109.948147

[2] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.

[3] M. Broy, C. Leuxner, W. Sitou, B. Spanfelner, and S. Winter. 2009. Formalizing the Notion of Adaptive System Behavior. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*. ACM, New York, NY, USA, 1029–1033. https://doi.org/10.1145/1529282.1529508

[4] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *Comput. Surveys* 41, 3, Article 15 (July 2009), 58 pages. https://doi.org/10.1145/1541880.1541882

[5] C. Ghezzi, J. Greenyer, and V.P.L. Manna. 2012. Synthesizing dynamically updating controllers from changes in scenario-based specifications. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, USA, 145–154. https://doi.org/10.1109/SEAMS.2012.6224401

[6] Carlo Ghezzi, Leandro Sales Pinto, Paola Spoletini, and Giordano Tamburrelli. 2013. Managing Non-functional Uncertainty via Model-driven Adaptivity. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press, Piscataway, NJ, USA, 33–42. http://dl.acm.org/citation.cfm?id=2486788.2486794

[7] D. W. Hosmer and S. Lemeshow. 1989. *Applied logistic regression*. Wiley, USA.

[8] Michael Jackson. 2001. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] Nathalie Japkowicz and Shaju Stephen. 2002. The Class Imbalance Problem: A Systematic Study. *Intell. Data Anal.* 6, 5 (Oct. 2002), 429–449. http://dl.acm.org/citation.cfm?id=1293951.1293954

[10] Ron Kohavi. 1995. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2 (IJCAI'95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1137–1143. http://dl.acm.org/citation.cfm?id=1643031.1643047

[11] J. Kramer and J. Magee. 2007. Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering (FOSE)*. IEEE Computer Society, USA, 259–268. https://doi.org/10.1109/FOSE.2007.19

[12] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM'05)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1251398.1251416

[13] MITRE. 2015. External Control of Assumed-Immutable Web Parameter. https://cwe.mitre.org/data/definitions/472.html. (December 2015). Last accessed 16 January 2016.

[14] Anders Møller and Mathias Schwarz. 2014. Automated Detection of Client-State Manipulation Vulnerabilities. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4, Article 29 (Sept. 2014), 30 pages. https://doi.org/10.1145/2531921

[15] Diogo Mónica and Carlos Ribeiro. 2011. WiFiHop—Mitigating the Evil Twin Attack through Multi-hop Detection. In *16th European Symposium on Research in Computer Security (ESORICS)*. Springer, Berlin, 21–39. https://doi.org/10.1007/978-3-642-23822-2_2

[16] P. Oreizy, N. Medvidovic, and R. N. Taylor. 1998. Architecture-based runtime software evolution. In *International Conference on Software Engineering*. IEEE Computer Society, USA, 177–186.

[17] David Lorge Parnas and Jan Madey. 1995. Functional Documents for Computer Systems. *Science of Computer Programming* 25, 1 (1995), 41–61.

[18] William N. Robinson. 2006. A requirements monitoring framework for enterprise systems. *Requir. Eng.* 11, 1 (2006), 17–41. https://doi.org/10.1007/s00766-005-0016-3

[19] Rick Salay, Marsha Chechik, Michalis Famelis, and Jan Gorzny. 2015. A Methodology for Verifying Refinements of Partial Models. *Journal of Object Technology* 14, 3 (2015), 3:1–31. https://doi.org/10.5381/jot.2015.14.3.a3

[20] Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 4, 2, Article 14 (May 2009), 42 pages. https://doi.org/10.1145/1516533.1516538

[21] M. Shepperd, D. Bowes, and T. Hall. 2014. Researcher Bias: The Use of Machine Learning in Software Defect Prediction. *Software Engineering, IEEE Transactions on* 40, 6 (June 2014), 603–616. https://doi.org/10.1109/TSE.2014.2322358

[22] Nazari Skrupsky, Prithvi Bisht, Timothy Hinrichs, V. N. Venkatakrishnan, and Lenore Zuck. 2013. TamperProof: A Server-agnostic Defense for Parameter Tampering Attacks on Web Applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY)*. ACM, New York, USA, 129–140. https://doi.org/10.1145/2435349.2435365

[23] V.E.S. Souza, A. Lapouchnian, and J. Mylopoulos. 2012. (Requirement) evolution requirements for adaptive systems. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, USA, 155–164. https://doi.org/10.1109/SEAMS.2012.6224402

[24] Vítor E. Silva Souza, Alexei Lapouchnian, William N. Robinson, and John Mylopoulos. 2011. Awareness Requirements for Adaptive Systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, New York, NY, USA, 60–69. https://doi.org/10.1145/1988008.1988018

[25] Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. 2009. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *International Requirements Engineering Conference*. IEEE Computer Society, USA, 79–88.

[26] Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. 2010. RELAX: a language to address uncertainty in self-adaptive systems requirement. *Requirements Engineering* 15, 2 (2010), 177–196.

[27] Pamela Zave and Michael Jackson. 1997. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6, 1 (Jan. 1997), 1–30. https://doi.org/10.1145/237432.237434

[28] Ji Zhang and Betty H. C. Cheng. 2006. Model-based Development of Dynamically Adaptive Software. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. ACM, New York, USA, 371–380. https://doi.org/10.1145/1134285.1134337