Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Formal Analysis and Offline Monitoring of Electronic Exams

Ali Kassem · Yliès Falcone · Pascal Lafourcade

Abstract More and more universities are moving toward electronic exams (in short *e-exams*). This migration exposes exams to additional threats, which may come from the use of the information and communication technology.

In this paper, we identify and define several security properties for e-exam systems. Then, we show how to use these properties in two complementary approaches: model-checking and monitoring. We illustrate the validity of our definitions by analyzing a real e-exam used at the pharmacy faculty of *University Grenoble Alpes* (UGA) to assess students. On the one hand, we instantiate our properties as queries for ProVerif, an automatic verifier of cryptographic protocols, and we use it to check our modeling of UGA exam specifications. ProVerif found some attacks. On the other hand, we express our properties as Quantified Event Automata (QEAs), and we synthesize them into monitors using MarQ, a Java tool designed to implement QEAs. Then, we use these monitors to verify real exam executions conducted by UGA. Our monitors found fraudulent students and discrepancies between the specifications of UGA exam and its implementation.

1 Introduction

To broaden the access to their educational programs, several universities, such as Stanford and Berkeley, have initiated Massive Open Online Courses (*e.g.,* Coursera and edX). In such massive courses, e-exams are offered as a service to assess students. A student passing the exam receives a certificate. However, because of cheating concerns, these certificates are not widely accepted yet. Even in more traditional settings, universities have started to adopt e-exams to replace traditional exams, especially in the case of multiple-choice and short open answer questions. For example, pharmacy exams at *University Grenoble Alpes* (UGA)

Ali Kassem
INRIA, France
E-mail: Ali.Kassem@inria.fr

Yliès Falcone
Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, F-38000 Grenoble, France
E-mail: Ylies.Falcone@univ-grenoble-alpes.fr

University Clermont Auvergne, LIMOS, Aubière, France
E-mail: Pascal.Lafourcade@udamail.fr

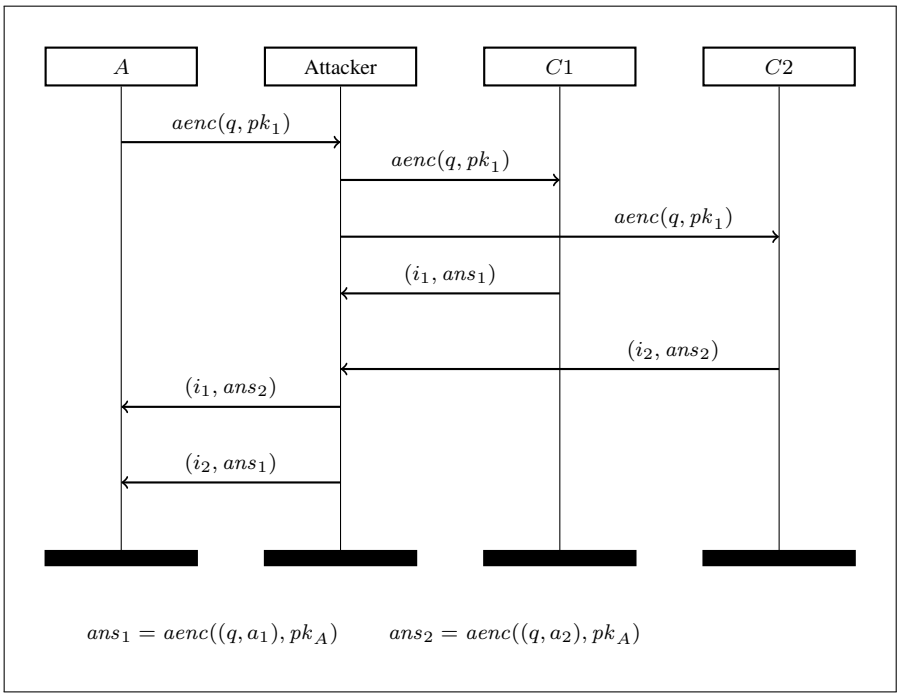


Fig. 1: An example of simple attack: swapping answers.

have been organized electronically using tablet computers since 2014 [Fig15], as part of the project *Épreuves Classantes Nationales informatisées*[1]. Other institutions, such as ETS, CISCO, and Microsoft have for a long time already adopted their own platforms to run, generally in qualified centers, e-exams required to obtain their program certificates.

This migration toward e-exams introduces several security and trust issues, especially in case of remote e-exams. For example, consider the simple attack illustrated in Fig. 1. The attack assumes that the attacker has full control of the communication channels. He first receives a question from the exam authority A, and then delivers it to both candidates $C1$ and $C2$. Each of the candidates prepares an answer to the question, and sends it encrypted (using $aenc$) with the authority's public key pk_A. The attacker, who stands in the middle, blocks the answers, swaps them, and then sends them to the authority. We can prevent this attack by inserting the candidate identities (i_1, i_2) inside the encryption. Note that, such an attack can be detected with property Answer Authentication (cf. Section 4).

The most common threats are those related to students cheating. But there are other relevant threats which may come from exam authorities such as evaluating student answers wrongly, manipulating marks after evaluation, and providing fake diplomas. For example, in the Atlanta scandal, school authorities colluded in changing student marks to improve the rankings of their institution and get more public funds [Cop13]. BBC revealed another scandal where ETS was shown to be vulnerable to a fraud perpetrated by official invigilators in collusion with the candidates whose intent was to obtain a visa: the invigilators dictated the correct answers during the test [Wat14]. Moreover, errors and flaws may be introduced

[1] www.side-sante.org

during the implementation of an exam protocol, or at runtime. Henceforth, the traditional motivations for verifying the implementations of e-exams platforms naturally apply.

More or less effective mitigations exist, *e.g.,* conducting exams in a supervised room or using proctoring software such as ProctorU[2]. However, such mitigations are not sufficient, and e-exams systems must be analyzed and verified against various threats.

In this paper, we propose a framework for the verification of e-exam systems. We illustrate our approach by making use of model-checking to verify the correctness of exam specifications before implementing them, and monitoring to verify exam individual executions in order to detect implementation errors and misbehaviors. More precisely, the paper provides the following contributions:

- Definitions of the fundamental security properties of e-exam systems;
- Two instantiations of these definitions as queries for ProVerif [Bla01], an automatic verifier in the formal model, and Quantified Event Automata (QEAs) [BFH+12,Reg14];
- A case study to validate our definitions and the effectiveness of our approach: an analysis of UGA exam specifications using ProVerif, and an analysis of real executions of UGA exam using MarQ[3] [RCR15] (Monitoring At Runtime with QEA), a Java-based monitoring tool designed to support QEAs. UGA exam is used by 39 universities in France. This makes its analysis significant, not only to illustrate our framework, but also to validate its security. As a result of our analysis, we find counterexamples of the specifications, violations in the exam executions, and discrepancies between the specifications and the implementation. Our results emphasize the significance of security verification, and confirms the necessity of runtime monitoring to complement model checking. Indeed, some properties are violated at runtime even though they are safe according to the protocol specifications. Our analysis of UGA exam was done in partnership with the designers of the system. We have interacted with them to understand the specifications and the design of the exam system, in order to construct our models in ProVerif and in MarQ. Naturally, we communicated all results of our study to the designers in order to improve the UGA exam system.

Remark 1 (*Characteristics of e-exams*). At this point, we would like to stress some of the characteristics of e-exam systems. Some of the properties proposed in this paper resemble those that have been studied in other domains such as auctions and voting. For instance, Candidate Eligibility (cf. Definition 4) is similar to bidder eligibility and voter eligibility. However, there are fundamental differences between exam systems and the systems in other domains:

- In exam systems, preserving the association between a candidate and his answer is a desired property. However, losing the association between a voter (resp. bidder) and his vote (resp. bid) is not a problem. Actually, it could be a desired property in voting (resp. auctions).
- A main difference between exams and auctions is related to the adversary model: in exams, candidates might collude to increase their marks, while in auctions bidders are in competition to win.
- In exam systems, questions should be kept secret until the examination phase starts (*i.e.,* when candidates sit for the exam to answer its questions). This is in contrast to auctions and voting systems where the goods and candidates (for election) are previously known.

[2] `http://www.proctoru.com`

[3] `https://github.com/selig/qea`

- In exam systems, it is desired to keep candidates anonymous for the examiners during marking. A property which is falsified when the marks are assigned to the candidates. It is a specific property to the exam domain.
- In exams systems, the exam authority might collude with candidates in order, *e.g.,* to increase the ranking of its institution. A similar collusion might happen in voting systems for a certain candidate to win. However, we do not find such incentive for collusion in auction systems where bidders want to buy the good with a low price, while the seller wants to sell it with the highest possible price.

Paper organization. The rest of the paper is organized as follows. We recall ProVerif syntax and the QEA definition in Section 2. In Section 3, we define our event-based model for e-exam systems. We then define and instantiate several fundamental e-exam properties in Section 4. In Section 5, we verify the UGA exam specifications using ProVerif. Then in Section 6, we perform offline monitoring for real executions of UGA exam using MarQ tool. We discuss related work in Section 7 and conclude in Section 8.

2 Preliminaries

We respectively use ProVerif and MarQ for the model-checking and monitoring parts of our case study. In Section 2.1, we present ProVerif syntax and its verification capabilities. For more details about ProVerif and how to use it, we refer to the related papers [Bla01, Bla02, AB05a, AB05b] and ProVerif manual [BSC16]. Then, in Section 2.2, we present QEAs [BFH$^+$12, Reg14] and illustrate how to express a property as QEA using an example.

2.1 ProVerif

ProVerif is an automatic model checker designed for analyzing the security of cryptographic protocols. ProVerif takes as input a protocol and a property modeled in Horn clauses or in the applied π-calculus [AF01]. Note that the tool works with Horn clauses, so it translates an applied π-calculus input into Horn clauses. However it is more natural and easier to model protocols using the applied π-calculus format than the Horn clauses format. ProVerif then determines whether the property is satisfied by the protocol or not. In case of failure, ProVerif may provide a trace of the obtained attack.

Honest parties (*i.e.,* those that follow protocol's specifications and do not leak information to the attacker) are modeled using processes. Processes can exchange messages on public or private channels, create fresh random values and perform tests and cryptographic operations, which are modeled as functions on terms. They are equipped with a finite set of types, free names, and function symbols (constructors) which are associated with a finite set of destructors. The relation between constructors and destructors is described using equational theories. For example, $sdec(senc(m, k), k) = m$ means that term $sdec$, representing symmetric decryption, is the inverse function of $senc$, representing symmetric encryption, and that they are related as one expects in a correct symmetric encryption scheme.

To further illustrate the calculus notation of ProVerif, we consider the handshake protocol given as follows:

$$
\begin{aligned}
&C \to S : pk(skA)\\
&S \to C : aenc(sign((pk(skB), k), skB), pk(skA))\\
&C \to S : senc(s, k)
\end{aligned}
$$

```
1  let client(pkC:pkey,skC:skey,pkS:pkey) =
2      out(c,pkC);
3      in(c,x:bitstring);
4      let y = adec(x,skC) in
5      let (=pkS,k:key) = checksign(y,pkS) in
6      out(c,senc(s,k));
7      termClient(pkC, k).
8
9  let server(pkS:pkey,skS:skey) =
10     in(c,pkX:pkey);
11     new k:key;
12     event acceptServer(pkX, k);
13     out(c,aenc(sign((pkS,k),skS),pkX));
14     in(c,x:bitstring);
15     let z = sdec(x,k) in
16     0.
```

Listing 1: Client and server processes for handshake protocol.

where pk is a function that gives the related public key of a given secret key, $aenc$ is an asymmetric encryption, and $sign$ is a signature function. The goal of the protocol is for client C to share a secret s with server S.

Listing 1 presents ProVerif encoding of client and server processes for the handshake protocol. As an example, we describe the server process, which takes as arguments the public key pkS and private key skS of the server. It first waits for an input of type public key pkey on channel c (Line 10). After creating a fresh key k (Line 11), it emits the event acceptServer(pkX, k) (Line 12). Then, it outputs on c the key k and pkS signed with skS and then encrypted with pkX (Line 13). Finally, it waits for an input of type bitstring (Line 14), and applies the destructor sdec (corresponding to symmetric decryption) on the received value x and stores the obtained value in z (Line 15). Here 0 is the null process (Line 16). Note that, the application of the destructor sdec fails if the received message x does not have the correct form (encryption of some message with k).

Destructor applications can be used as follows: let z = M in P else Q where M is a term that contains some destructors. When such a statement is encountered, there are two possible outcomes: (i) if the term M does not fail (*i.e.,* for all destructors in M, matching rewriting rules exist), then z is bound to M and the branch P is taken; (ii) otherwise, the Q branch is taken. For brevity, the sub-term else Q is omitted when Q is 0. Note that, the term (=pkS,k:key) = checksign(y,pkS) in client process (Line 5) succeeds if y is of the form sign(pkS,k) for some k of type key.

To capture the relation between the constructors (senc, aenc in ProVerif, and sign) and the corresponding destructors (sdec. adec, and checksign), one has to define the following rewriting rules:

```
reduc forall m:bitstring, k:key; sdec(senc(m,k),k) = m
reduc forall m:bitstring, k:skey; adec(aenc(m,pk(k)),k) = m
reduc forall m:bitstring, k:skey; checksign(sign(m,k),pk(k)) = m
```

ProVerif also supports the following processes:

- if M=N then P else Q: conditional process, which behaves like P if M can be rewritten as N using the defined rewriting rules, and behaves like Q otherwise.
- P|Q: parallel composition of P and Q.
- !P: replication of P, that is an unbounded number of copies of P running in parallel.

– P.Q: is the composition of P and Q.

Note that, processes can be annotated with parametric events which flag important steps in the protocol execution without changing their behavior.

The ProVerif language is strongly typed. Default types include bitstring and channel. In our example, the terms c and s are respectively of types bitstring and channel. They have to be defined as follows:

```
1 free c: channel.
2 free s: bitstring [private].
```

By default, free names are known by the attacker. Secret free names have to be declared private. Types different from the default ones have also to be defined. For the handshake protocol, the user has to define the types key, pkey, and skey as follows:

```
1 type key.
2 type pkey.
3 type skey.
```

Finally, ProVerif uses the following syntax: $<>$ for "not equal" operator, $\&\&$ for conjunction, and $||$ for disjunction.

ProVerif can verify reachability properties ("*unreachability of bad event*") and correspondence assertions ("*event e_2 is preceded by event e_1*"). This allows us to check the reachability of a certain event and to check a certain relation between events, respectively. We make use of these two capabilities to reason about security properties of e-exam systems. For instance, checking whether the attacker can reach the secret s in the handshake protocol can be done by declaring the query: query attacker(s). Similarly, checking the reachability of an event e can be represented by the query: query event(e).

In addition to preserving the confidentiality of the secret s, the handshake protocol is also intended to ensure the authentication between the client and the server. To verify that: "when the client with key pkC thinks that he has finished a protocol run with the server using the symmetric key k, he actually did", we can define the following query:

```
query x:pkey, y:key; event(termClient(x,y)) ==> event(acceptServer(x,y))
```

where the events acceptServer and termClient are defined as follows:

– event acceptServer(pkey, key) is emitted when the server accepts to run the protocol with the client whose public key is supplied as the first argument, and the proposed key is supplied as the second argument.
– event termClient(pkey, key) is emitted when the client (with public key supplied as first argument) terminates the protocol while accepting the key that is supplied as the second argument.

The latter query ensures that for every possible execution trace and for any public key x and symmetric key y, the event termclient(x,y) is preceded by the event acceptServer(x,y). Note that such a basic correspondence can be extended to have conjunctions and disjunctions of events in the right-hand side of the arrow. ProVerif also supports injective correspondences, such as:

```
query x:pkey, y:key; inj−event(termClient(x,y)) ==> inj−event(acceptServer(x,y))
```

which holds if every occurrence of termClient(x,y) is preceded by a unique occurrence of acceptServer(x,y). Moreover, one can define nested correspondences in the form: E ==> F

where some events in F are replaced with correspondences. Nested correspondences allow us to check event order. However, ProVerif, as well as similar formal tools, does not support queries to verify properties of the form "*event* e_1 *is followed by event* e_2".

In ProVerif, the default attacker is a Dolev-Yao attacker [DY83], which has a complete control of the network, except the private channels. He can eavesdrop, remove, substitute, duplicate and delay messages that parties send to one another, and even insert messages of his choice on public channels.

We note that ProVerif makes some internal abstractions to translate protocols into Horn clauses. The main abstraction is that the translation ignores the number of repetitions of actions. This is due to the fact that clauses can be applied any number of times. Another abstraction is that ProVerif represents nonces as functions of previous exchanged messages. Thus, the position of nonces also effects the translation. As a result, ProVerif may output false counterexamples, that is an attack that is not actually valid. Moreover, ProVerif may state that a property "cannot be proved", which is a "do not know" answer. However, ProVerif is sound, that is, if it does not find a flaw then there is no flaw in the protocol. Moreover, when ProVerif proves the security of a protocol, then the protocol is secure for an unbounded number of sessions. This is one of the main features of ProVerif tool.

2.2 Quantified Event Automata

Quantified Event Automata are expressive formalism to represent parametric specifications to be checked at runtime. An Event Automaton (EA) is a (possibly) non-deterministic finite-state automaton whose alphabet consists of parametric events and whose transitions may be labeled with guards and assignments. The EA syntax is built from a set of event names $\mathcal{N}$, a set of values $\mathtt{Val}$ and a set of variables $\mathtt{Var}$ (disjoint from $\mathtt{Val}$). An event is a pair $\langle e, \overline{p} \rangle \in \mathcal{N} \times \mathtt{Sym}^*$ where $\mathtt{Sym} = \mathtt{Val} \cup \mathtt{Var}$ is the set of all symbols. We denote by $\mathtt{Event}$ the set of all events and we use a functional notation to denote events: $\langle e, \overline{p} \rangle$ is denoted by $e(\overline{p})$. An event $e(\overline{p})$ is ground if $\overline{p} \in \mathtt{Val}^*$. A variable can be mapped to a value using a *binding*, *i.e.,* an element of $\mathtt{Bind} = \mathtt{Var} \to \mathtt{Val}$. A binding can be applied to a symbol as a substitution, *i.e.,* replacing the symbol if it is defined in the binding. This can be lifted to events. A *trace* is defined as a finite sequence of ground events.

EAs and QEAs make use of guards and assignments on transitions. Guards are predicates on bindings, *i.e.,* total functions in $\mathtt{Guards} = \mathtt{Bind} \to \mathbb{B}$ where $\mathbb{B} = \{true, false\}$, and assignments are total functions on bindings, *i.e.,* elements of $\mathtt{Assign} = \mathtt{Bind} \to \mathtt{Bind}$. EAs are formally defined as follows.

Definition 1 (Event Automaton [BFH+12]). An EA is a tuple $\langle \mathcal{S}, \Sigma, \delta, s_0, \mathcal{F} \rangle$ where $\mathcal{S}$ is a finite set of states, $\Sigma \subseteq \mathtt{Event}$ is a finite alphabet, $\delta \subseteq (\mathcal{S} \times \Sigma \times \mathtt{Guards} \times \mathtt{Assign} \times \mathcal{S})$ is a finite set of transitions, $s_0 \in \mathcal{S}$ is the initial state, and $\mathcal{F} \subseteq \mathcal{S}$ is the set of final states.

The semantics of an EA is close to the one of a finite-state automaton, with the natural addition of guards and assignments on transitions. A transition can be triggered only if its guard evaluates to true with the current binding ("values of variables"), and the assignment updates the current binding.

A QEA defines a language (*i.e.*, a set of traces) over instantiated parametric events. Formally, it is an EA with some (or none) of its variables quantified by $\forall$ or $\exists$. The variables of an EA $\mathtt{E}$ are those that appear in its alphabet:

$$\mathtt{vars}(\mathtt{E}) = \{x \in \mathtt{Var} \mid \exists e(\overline{p}) \in \mathtt{E}.\Sigma : x \in \overline{p}\}$$

where $\texttt{E}.\Sigma$ denotes the alphabet of EA E. Note that not all variables need to be quantified. Unquantified variables are left free and they can be manipulated through assignments and updated during the processing of the trace.

Definition 2 (Quantified Event Automaton [BFH$^+$12]). A QEA is a pair $\langle \texttt{E}, \Lambda \rangle$ where $\texttt{E}$ is an EA and $\Lambda \in (\{\forall, \exists\} \times \texttt{vars}(\texttt{E}) \times \texttt{Guards})$ is a list of quantified variables with guards.

QEAs are depicted graphically in this paper. The initial state of a QEA has an arrow pointing to it. The shaded states are final (*i.e.,* accepting) states, while white states are failure states (*i.e.,* non-accepting states). Square states are closed-to-failure, *i.e.,* if no transition can be taken, then there is an implicit transition to a failure state. Circular states are closed-to-self (aka skip states), *i.e.,* if no transition can be taken, then there is an implicit self-looping transition. We use the notation $\frac{[guard]}{assignment}$ to write guards and assignments on transitions, $:\hat{=}$ for variable declaration then assignment, $:=$ for assignment, and $=$ for equality test.

The semantics of a QEA is given in terms of a set of EAs (that are "instances" of the QEA). Instead of re-iterating the complete semantics of QEA (fully described in [BFH$^+$12]), we provide an intuitive semantics that serves the purpose of this paper, and illustrate the semantics below in Example 1. By processing a trace, a QEA generates a set of EAs by matching the values in the trace with the quantified variables, each EA being associated to a binding. A "slice" of the trace is fed to each instantiated EA: a slice contains the events that match the binding associated to the EA. The (possibly partial) binding associated to an EA completes with the reception of events and when firing transitions by either matching already bound variables or binding unbound variables. Then, the quantifiers determine which of these EA must accept the trace it receives. The following example illustrates the graphical notation and the semantics of QEAs.

Example 1 (*Quantified Event Automaton*). We consider the following property: "for any i, event $e_2(i)$ is preceded by event $e_1(i)$". Such a property can be expressed using the QEA depicted in Fig. 2, with the alphabet $\Sigma_p = \{e_1(i), e_2(i)\}$ and initial state (1). The alphabet Σ_p contains only the events $e_1(i)$ and $e_2(i)$, so other events in the trace are ignored. The QEA has two states (1) and (2), which are both final states. It has one quantified variable i, and zero free variables. As its initial state is final, then the empty trace is accepted by this QEA. State (1) is a square state, so an event $e_2(i)$ at this state leads to an implicit failure state, which is what we want as it would not be preceded by event $e_1(i)$ in this case. An event $e_1(i)$ in state (1) leads to state (2) which is a skipping state, so after event $e_1(i)$ any sequence of events (for the same value of i) is accepted. The quantification $\forall i$ means that the property must hold for all values that i takes in the trace, *i.e.,* the values obtained when matching the symbolic events in the specification with concrete events in the trace. For instance, to decide whether the trace $e_1(i_1).e_2(i_2).e_2(i_1).e_1(i_2)$ is accepted or not, first it is sliced based on the values that can match i, which results in the following two slices: $i \mapsto i_1$: $e_1(i_1).e_2(i_1)$, and $i \mapsto i_2$: $e_2(i_2).e_1(i_2)$. Then, each slice is checked against the event automaton instantiated with the appropriate value for i. The slice associated to i_1 is accepted as it ends in the final state (2), while the slice associated to i_2 is rejected since event $e_2(i_2)$ leads from state (1) to an implicit failure state. Therefore, the whole trace is not accepted by the QEA because of the universal quantification on i.

Remark 2 (Verification constraints in monitoring - Monitorability) In Section 2.1, we discuss the limitations of formal analysis with ProVerif. In general, as monitoring is a non-exhaustive verification technique (only one run of the system is analyzed), it does not suffer from the same limitations as formal verification (in the sense of Section 2.1). In monitoring

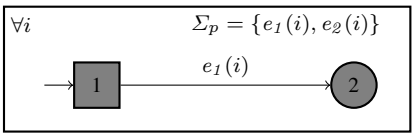


Fig. 2: A QEA expressing "event $e_2(i)$ is preceded by an event $e_1(i)$".

a specification against a system, it is only required that i) the events involved in the specification are observable by the monitor, and ii) the semantics of the specification language is based on single traces [FFM12, FFJ+12]. Moreover, the semantics of the specification language should be based on finite-traces, or if it based on infinite traces, the specification should be then *monitorable* [FFM12]. Monitorability is used to define the conditions under which a specification can be monitored. Several definitions of monitorability were defined (cf. [KKL+02, PZ06, FFM09, FAI15]). We note that defining conditions on the monitorability of specifications is irrelevant in this paper since QEAs are endowed with a finite-trace semantics.

3 An Event-based Model of E-exams

We define an *e-exam run* (or *e-exam trace*) as a finite sequence of events. Such event-based modeling of e-exam run is appropriate for expressing and verifying e-exam properties using both model-checking and monitoring. Using a model checking approach (and a tool such as ProVerif), all the possible runs are analyzed in a "*white-box* fashion" and the security of a protocol model is validated or a flaw is discovered. Using a monitoring approach, individual runs are analyzed in a "*black-box* fashion" in order to check whether there are weaknesses in the protocol implementation or misbehaviors in the execution. Recall that *white-box* (resp. *black-box*) fashion refers to the fact that the internals and model of the system are (resp. are not) known.

In the following, we specify the parties involved in an e-exam. Then, we define the events related to an e-exam run.

3.1 Overview of an E-exam Protocol

An exam involves at least two roles: *candidate* and *exam authority*. The exam authority can have several sub-roles: *registrar* which registers candidates; *question committee* which prepares the questions; *invigilator* which supervises the exam, collects the answers, and dispatches them for marking; *examiner* which corrects the answers and evaluates them; and *notification committee* which delivers the marking. Generally, exams run in phases, where each phase ends before the next one begins. We note that our model is independent from the number of phases and their organization, as long as, the necessary events defined below in Section 3.2 are correctly recorder.

3.2 Events Involved in an E-exam

Events flag important steps in the execution of the exam. We define the following events that are assumed to be recorded during the exam or built from data logs. We consider *para-*

metric events of the form $e(p_1, \ldots, p_n)$, where e is event name, and $p_1, \ldots, p_n$ is a list of symbolic parameters which take some data values at runtime.

- Event $register(i)$ occurs when candidate i registers to the exam.
- Event $get(i, q)$ occurs when candidate i gets question q.
- Event $change(i, q, a)$ occurs when candidate i sets his answer to question q to a.
- Event $submit(i, q, a)$ occurs when candidate i sends a as an answer to question q.
- Event $accept(i, q, a)$ occurs when exam authority receives and accepts answer a to question q from candidate i.
- Event $corrAns(q, a, b)$ occurs when exam authority specifies a as a correct answer to question q, and provides score b for this answer. Note that more than one answer (possibly with different scores) can be correct for a question.
- Event $marked(i, q, a, b)$ occurs when answer a from candidate i to question q is scored with b. Note, we can omit candidate identity i to capture anonymous marking.
- Event $assign(i, m)$ occurs when mark m is assigned to candidate i.
- Event $start(t)$ occurs when the examination phase starts, at time t.
- Event $finish(t)$ occurs when the examination phase ends, at time t.

In the monitoring approach, all events are time stamped, however we parameterize them with time only when it is relevant for the considered property. In the model checking approach, we consider ordered events, which amounts to considering a discrete time.

In the rest of the paper, we may omit some parameters from the events when they are not relevant. For instance, we may use $submit(i)$ when candidate i submits an answer regardless of his answer.

4 E-exam Properties

Different sorts of properties are desired for e-exam systems. In the following, we identify and define several properties which aim at ensuring e-exams correctness. We split the requirements of a correct e-exam in several properties in order:

- to make our properties as simple as possible, so that they can be expressed using various formalisms;
- to enable the verification of as many requirements as possible depending on the available events;
- to narrow the sources of potential failures, and thus to facilitate their identification.

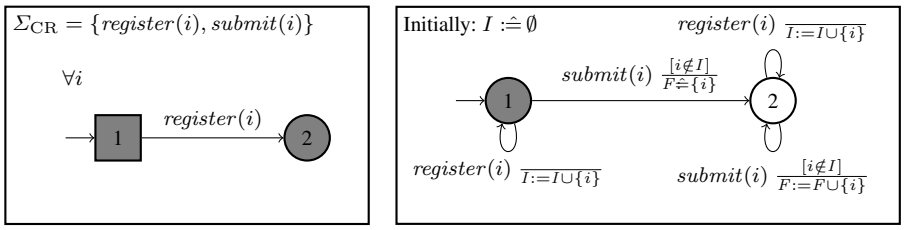
For each property, we provide an abstract definition which relies on the event-based model. Moreover, we provide two instantiations of our properties expressed as ProVerif queries and QEAs. We note that for QEAs, we use a skipping state rather than a close-to-failure state, when both states serve the purpose.

Candidate Registration. This property states that no candidate submits an answer without being registered to the exam. The intuition is that an unregistered candidate could try to fake the system by submitting an answer.

Definition 3 (Candidate Registration). An exam run satisfies Candidate Registration if every event $submit(i)$ is preceded by an event $register(i)$.

Candidate Registration property can be directly translated into a ProVerif query as follows:

query i:ID; event(submit(i)) ==> event(register(i))



(a) A QEA for Candidate Registration. (b) A QEA for Candidate Registration with Auditing.

Fig. 3: Candidate Registration expressed as QEA.

To verify individual exam runs, we express Candidate Registration as the QEA depicted in Fig. 3a. State (1) is a close to failure state, so an event $submit(i)$ that is not preceded by event $register(i)$ leads to a failure. For example, the run $submit(i).register(i)$ is rejected. An event $register(i)$ in state (1) leads to state (2) which is a skipping (circular) state. Henceforth, given a candidate i, any run starting with event $register(i)$ is accepted.

The QEA of Fig. 3a allows us to verify Candidate Registration, however it does not identify all the candidates that violate its requirements. Figure 3b depicts another QEA that, additionally, reports all candidates that violate Candidate Registration. The latter QEA, instead of quantifying on i, it collects in a set I all candidates that register to the exam, so that any occurrence of event $submit(i)$ at state (1) with $i \notin I$ fires a transition to the failure state (2). Such a transition results in the failure of the property since all transitions from state (2) are self-loops. In state (2), a set F is used to collect all unregistered candidates that submit answers. For example, the run $submit(i).register(i)$ results in set $F = \{i\}$.

We refer to QEAs that are similar (in concept) to that of Fig. 3a as *verification QEAs*, and to QEAs that are similar to that of Fig. 3b as *auditing QEAs*. For the rest of the properties in this section, we only provide verification QEAs, while auditing QEAs are provided in Appendix A (except for Cheater Detection as it is an auditing property by itself).

Candidate Eligibility. In a correct exam run, the exam authority should accept answers only from registered candidates. This is ensured by Candidate Eligibility. The definition of Candidate Eligibility is similar to that of Candidate Registration except that the event $submit(i)$ is replaced by the event $accept(i)$.

Definition 4 (Candidate Eligibility). An exam run satisfies Candidate Eligibility if every event $accept(i)$ is preceded by an event $register(i)$.

Candidate Eligibility can be expressed in ProVerif using a basic correspondence as follows:

query i:ID; event(accept(i)) ==> event(register(i)).

A verification QEA that expresses Candidate Eligibility is depicted in Fig. 4. As an example, the run $accept(i, q, a).register(i)$ is rejected by this QEA.

Answer Authentication. This property states that all accepted answers are actually submitted by candidates. Answer Authentication is useful to detect answers that may be added, for instance, by the exam authority.

Definition 5 (Answer Authentication). An exam run satisfies Answer Authentication if every occurrence of event $accept(i, q, a)$ is preceded by a distinct occurrence of event $submit(i, q, a)$.

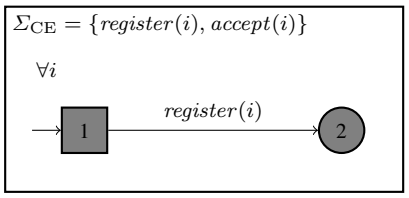


Fig. 4: A QEA for Candidate Eligibility.

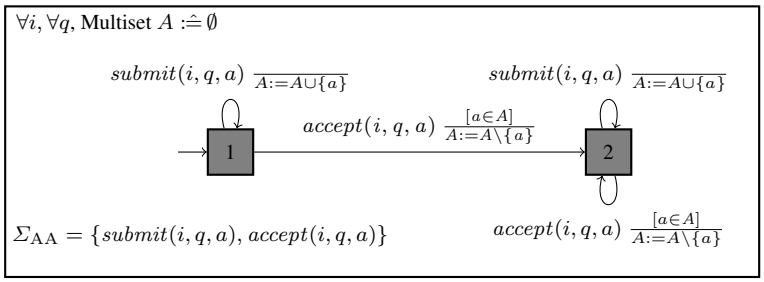


Fig. 5: A QEA for Answer Authentication.

Note that the word "distinct" in Definition 5 means that an answer can be accepted twice only if it is submitted twice. For example, the run $submit(i,q,a).accept(i,q,a).accept(i,q,a)$ is rejected, while the run $submit(i,q,a).accept(i,q,a).submit(i,q,a).accept(i,q,a)$ is accepted.

Answer Authentication can be expressed in ProVerif using an injective correspondence as follows:

query i:ID, q:bitstring, a:bitstring; inj−event(accept(i,q,a)) ==> inj−event(submit(i,q,a))

A verification QEA that formalizes Answer Authentication is depicted in Fig. 5. This QEA has no failure state, however it goes to failure from both states once an unsubmitted answer (*i.e.,* $a \notin A$) is accepted.

The previous three properties, Candidate Registration, Candidate Eligibility, and Answer Authentication ensure together that all the accepted answers are actually submitted by previously registered candidates. That is, they ensure that the following order of the events $register(i)$, $submit(i,q,a)$, and $accept(i,q,a)$ is respected.

Answer Singularity. Property Answer Singularity states that, for each question, at most one answer should be accepted from each candidate.

Definition 6 (Answer Singularity). An exam run satisfies Answer Singularity if for any i, q, there is only a single occurrence of event $accept(i,q,a)$ for some answer a.

As we mentioned in Section 2.1, in ProVerif an action may be repeated an unbounded number of times. Therefore, ProVerif does not provide a query to verify that a certain event occurs only once. Thus, we cannot verify Answer Singularity using ProVerif. However, we can verify a slightly-modified variant of it, which allows multiple copies of the same answer to the same question from the same candidate. We refer to this modified variant as Answer Singularity*.

Answer Singularity* can be expressed in ProVerif as "*unreachability of a bad event*". More precisely, we can define a process Test that receives from the exam authority, on a

```
1  let Test(chTest:channel) =
2  in(chTest, (=accept, i:ID, q:bitstring, a:bitstring));
3  in(chTest, (=accept, i':ID, q':bitstring, a':bitstring));
4
5  if i'=i && q'=q && a'<>a then
6  event Bad.
```

Listing 2: Test for Answer Singularity* in ProVerif.

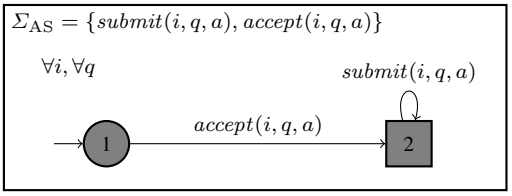


Fig. 6: A QEA for Answer Singularity.

private channel, the triplet (i, q, a) whenever an event $accept(i, q, a)$ is emitted. The process Test then checks whether there are two different answers (a'<>a) to the same question from same candidate. If it is the case, Test emits event Bad. Listing 2 depicts the process that takes parameters for two events $accept$, and then checks them. By having unbounded instances of process Test in parallel, thanks to the replication operator "|", and as the message sent on private channels are repeated any number of times, ProVerif can pairwisely compare the parameters of all events $accept$.

The situation is different with QEAs as it allows us to express the requirements of Answer Singularity. We can express Answer Singularity as the QEA depicted in Fig. 6. As examples, the runs $accept(i_1, q_1, a_1)$ and $submit(i_1, q_1, a_1).accept(i_1, q_1, a_1)$ are accepted. While, the run $submit(i_1, q_1, a_1).accept(i_1, q_1, a_1).submit(i_1, q_1, a_2).accept(i_1, q_1, a_2)$ is rejected since two answers are accepted from the same candidate to the same question.

Acceptance Assurance. An exam has to be fair to the candidates in the following sense. For a given question, the exam authority has to accept an answer from a candidate who has submitted at least one answer to that question. This is ensured by Acceptance Assurance.

Definition 7 (Acceptance Assurance). An exam run satisfies Acceptance Assurance if the first occurrence of event $submit(i, q)$ is followed by an event $accept(i, q)$.

In Definition 7, we say "the first occurrence of the event $submit(i, q)$" as not every event $submit(i, q)$ has to be followed by an event $accept(i, q)$. However, at least one event $submit(i, q)$ is followed by an event $accept(i, q)$ for the property to hold. And clearly it is the case for the first such $submit(i, q)$ event when the property holds. For example, the trace $submit(i, q, a_1).accept(i, q, a_1).submit(i, q, a_2)$ satisfies Acceptance Assurance.

Property Acceptance Assurance cannot be checked using ProVerif (cf. Section 2.1). However, we can express Acceptance Assurance using the QEA depicted in Fig. 7. Note that this QEA succeeds even if the accepted answer is not one of the submitted answers. For example the run $submit(i, q, a).accept\ (i, q, a')$ with $a \neq a'$ is accepted.

Questions Ordering. The previous five properties define the main requirements that are usually needed regarding the answer submission and acceptance. However, depending on the

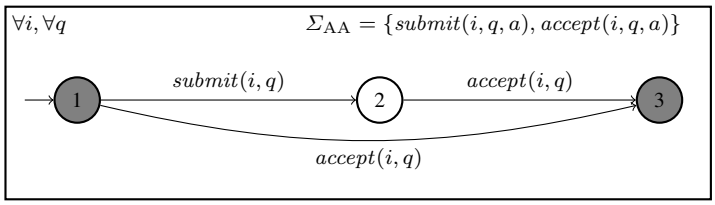


Fig. 7: A QEA for Acceptance Assurance.

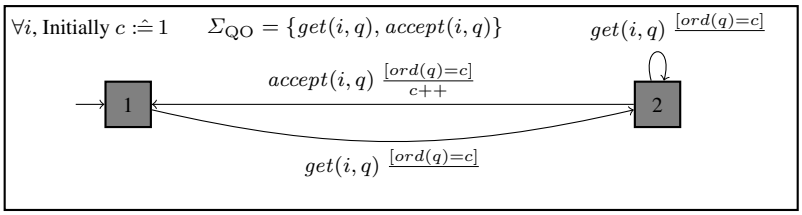


Fig. 8: A QEA for Questions Ordering.

exam, additional requirements might be needed such as: candidates have to answer the exam questions in a certain order. To capture the latter requirement, we define property Question Ordering which states that a candidate cannot get the next question before his answer to the current question is accepted by the exam authority.

Definition 8 (Questions Ordering). Let $q_1, \ldots, q_n$ denote the exam questions, which are ordered according to their indices. An exam run satisfies Question Ordering if, for $k = 2, \ldots, n$, event $get(i, q_k)$ is preceded by event $accept(i, q_{k-1})$ which in turn is preceded by event $get(i, q_{k-1})$.

Question Ordering can be expressed in ProVerif using the following nested correspondence:

```
query i:ID, q:bitstring, a:bitstring; event(get(i,q)) ==>
                    (event(accept(i,previous(q),a)) ==> event(get(i,previous(q))))
```

To relate the questions to a certain order in ProVerif, we define function previous with the following rewriting rule: q_k =previous(q_{k+1}) . Note that we have to i) consider a dummy question q_0 = such that q_0 =previous(q_1) and ii) emit the event accept(i,q_0,a) since, otherwise, the query would be violated by the event get(i,q_1) as no previous question was accepted before it.

Note also that the previous query does not check whether further answers to previous questions are accepted in the future. Actually, this cannot be verified using ProVerif. However, QEAs allow us to monitor this issue by determining whether more than one answer is accepted from the same candidate to the same question. Fig. 8 depicts a QEA that expresses the latter in addition to the main requirement of Question Ordering. This QEA makes use of a counter c (*per* candidate), which is initialized to 1 and incremented when an answer to the current question is accepted. Then after, the corresponding candidate can get the next question. The latter is captured by the condition $ord(q) = c$ where $ord(q)$ is the order of question q.

Exam Availability. This property states that answers can be accepted only during the examination time.

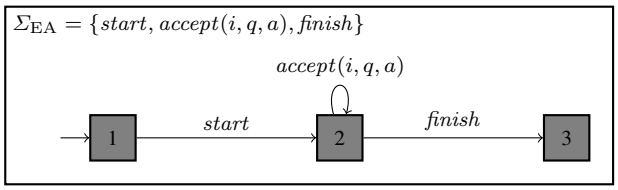


Fig. 9: A QEA for Exam Availability.

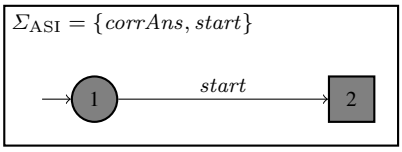


Fig. 10: QEA for Answer-Score Integrity.

Definition 9 (Exam Availability). An exam run satisfies Exam Availability if every occurrence of event $accept$ is preceded by the event $start$ and followed by the event $finish$.

We note that, a correct exam run should contain only one occurrence of each of the events $start$ and $finish$ (with $start$ occurring first). We take this into account when we instantiate Exam Availability as a QEA. Concerning ProVerif, again it does not allow us to check whether $start$ and $finish$ occurs only once each due to the action repetition issue. However, we can use it to verify that event $start$ occurs before event $finish$, using the query:

query event(finish) ==> event(start)

The first part of Exam Availability, *i.e.,* "$accept$ is preceded by $start$", can be expressed in ProVerif using the query: query event(accept) ==> event(start). However the second part, "$accept$ is followed by $finish$", cannot be checked using ProVerif.

A verification QEA that expresses Exam Availability is depicted in Fig. 9. Note that event $submit$ is not a member of Σ_{EA}, thus the submission of an answer outside the exam time is not considered as an irregularity by Exam Availability. However, one can consider event $submit$ or any other event. In such a case, the QEA in Fig. 9 has to be edited by looping over state (2) with any added event.

Another variant of Exam Availability: Exam Availability with Flexibility, which supports flexibility in exam duration and its starting time is defined in Appendix B.

Answer-score integrity. The exam authority delivers, for each question, the correct answers with the related scores. Answer-Score Integrity ensures that neither the correct answers nor the related scores of any question are modified after the examination phase begins. The intuition is that the exam authority could modify some correct answers to match a certain candidate's answers, or favors a candidate over the others by changing the scores after the examination phase.

Definition 10 (Answer-Score Integrity). An exam run satisfies Answer-Score Integrity if there is no event $corrAns$ that appears after the event $start$.

Answer-Score Integrity cannot be analyzed using ProVerif. However, we can express it using the verification QEA depicted in Fig. 10. Note that, similarly to Exam Availability, the QEA fails if event $start$ appears more than once in the run. The idea is that, after the event $start$

```
let Test(ch11:channel, ch12:channel, ch13:channel, ch21:channel, ch22:channel,
ch23:channel, idC1:ID, idC2:ID, ques1:bitstring, ques2:bitstring, ques3:bitstring) =

in(ch11,(=idC1, =ques1, ans11:bitstring));
in(ch12,(=idC1, =ques2, ans12:bitstring));
in(ch13,(=idC1, =ques3, ans13:bitstring));

in(ch21,(=idC2, =ques1, ans21:bitstring));
in(ch22,(=idC2, =ques2, ans22:bitstring));
in(ch23,(=idC2, =ques3, ans23:bitstring));

if (ans11 = ans21 && ans12 = ans22 && ans13 = ans23)
|| (ans11 = ans21 && ans12 = ans22)
|| (ans11 = ans21 && ans13 = ans23)
|| (ans12 = ans22 && ans13 = ans23)
then
event Bad
else
event Good.
```

Listing 3: Cheater Detection test for two candidates, three questions, and threshold $d = 1$.

is encountered (in state (1)), a transition to state (2), which is a close to failure state, occurs. In state (2), any occurrence of either event $corrAns$ or event $start$ fires a transition to an implicit failure state. As an example, the run $corrAns.start$ is accepted by this QEA, while the run $start.corrAns$ is rejected.

Cheater Detection. In an exam run, a candidate may copy his answers from another candidate. Cheater Detection helps detecting such cases using a notion of *distance* between candidates' answers. The distance between two candidates could be simply the number of their different answers. A more sophisticated form of distance is one that considers answers' acceptance time and/or the physical location of the candidates in the exam room.

Definition 11 (Cheater Detection). An exam run satisfies Cheater Detection for some distance `D` and a threshold d if for any two candidates, the distance between their answers is greater than d.

Note that, Cheater Detection can just help to detect potential fraud, since we cannot avoid detecting as cheaters two excellent candidates that have provided the right answers to all questions in a regular manner.

Cheater Detection is essential mainly when individual runs are considered. Actually when all possible runs are considered, Cheater Detection always fails as a run where $\texttt{D} < d$, for some candidates, is always possible. The goal is to detect whether it is the case for a given run.

Regardless of its significance, we can express Cheater Detection using ProVerif for the simple distance (*i.e.,* number of different answers) as "*unreachability of bad event*". More precisely, we can define a process Test which takes on private channels the triplets (i, q, a) whenever an event $accept(i, q, a)$ is emitted. Test then compares candidates' answers and emits event a bad event if *Distance* $\leqslant d$. Listing 3 depicts such a test process for the case of two different candidates, three questions, and threshold $d = 1$.

For runtime monitoring, we use a more fair form of distance `D` which considers, in addition to the answer values, their acceptance time. Formally, for two candidates i and j,

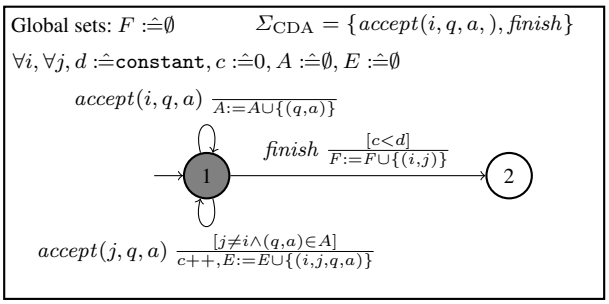


Fig. 11: QEA for Cheater Detection.

the distance of j's answers from i's answers is $D_{ij} = \sum_{k=1}^{k=N} d_{ij}(q_k)$ with

$$d_{ij}(q_k) = \begin{cases} 1 \text{ if } a_k^i \neq a_k^j \text{ or } time(a_k^j) \leqslant time(a_k^i) \\ 0 \text{ otherwise} \end{cases}$$

where N is the total number of questions, q_k is the k^{th} question (according to a fixed order), a_k^i and a_k^j are respectively i's answer j's answer to the question q_k, and $time(a)$ is the acceptance time for the answer a. The definition of distance D_{ij} considers identical answers as far answers if the answer from j is accepted before the answer from i. Note that $D_{ij} \neq D_{ji}$ in general. This allows us to recognize which candidate was copying from the another, that is if $D_{ij} \leqslant d$ then it is likely that candidate j was copying from candidate i.

A QEA that expresses Cheater Detection is depicted in Fig. 11. For candidates i and j, the QEA collects in a set E the identical answers if the acceptance time for j is greater than that of i. Note that, any answer accepted from j is ignored if no answer for the same question is yet accepted from i.

Marking Correctness. The last two properties concern marking. Marking Correctness ensures that all the answers are marked correctly.

Definition 12 (Marking Correctness). An exam run satisfies Marking Correctness if, every event $marked(q,a,b)$ is preceded by an event $corrAns(q,a,b)$ if and only if $b \neq 0$.

Using ProVerif, we can model Marking Correctness as "*unreachability of a bad event*". Note that in this case, instead of considering the event $corrAns$, we assume that a process Test has access to a marking algorithm as black-box or it takes the marking scheme as an argument. The process Test takes from the examiner on a private channel the parameters (q,a,b) whenever the event marked(q,a,b) is emitted. Test then checks whether the answer a and the mark m are consistent, and emits the event Bad if it is not the case. Listing 4 depicts process Test for the case where there are three questions with one correct answer each, and two possible score values mOK and mKO.

A verification QEA that expresses Marking Correctness is depicted in Fig. 12. The correct answers for the considered question are collected in a set A (self loop over state (1)). In state (1), once an answer to the considered question is marked correctly, a transition to state (3) is fired. Otherwise, if an answer is marked in a wrong way a transition to an implicit failure state occurs. In state (3), the property fails either if an answer is marked in a wrong way, or if an event $corrAns(q,a,b)$ is encountered as this means that certain answers are marked before all the correct answers are set. For example, run $corrAns(q,a,b).marked(q,a,b)$ is accepted, while run $marked(q,a,b).corrAns(q,a,b)$ is rejected.

```
let Test(marking scheme) =
in(chTest, (ques:bitstring, ans:bitstring, m:bitstring));
if (ques=q1 || ques=q2 || ques=q3) then

if (ques=q1 && ans=cAns1 && m = mOK)
|| (ques=q2 && ans=cAns2 && m = mOK)
|| (ques=q3 && ans=cAns3 && m = mOK)
|| (ques=q1 && ans<>cAns1 && m = mKO)
|| (ques=q2 && ans<>cAns2 && m = mKO)
|| (ques=q3 && ans<>cAns3 && m = mKO)
then
event Good
else
event Bad.
```

Listing 4: Marking Correctness test for three questions with one correct answer each, and two possible score values mOK and mKO.

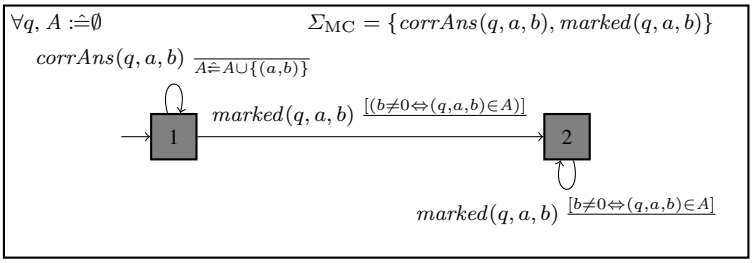


Fig. 12: A QEA for Marking Correctness.

Mark Integrity. The last property, Mark Integrity, states that each candidate should be assigned with his right mark, that is the one delivered on his accepted answers regardless of its correctness. Mark Integrity together with Marking Correctness, guarantee that each candidate participating in the exam gets the correct mark corresponding to his answers.

Definition 13 (Mark Integrity). An exam run satisfies Mark Integrity if every event $assign(i, m)$ is preceded by events $marked(i, q_1, a_1, b_1), \ldots, marked(i, q_n, a_n, b_n)$ with $n \in \mathbb{N}^*$ and mark m is equal to $markAlg(b_1, \ldots, b_n)$, where $markAlg$ computes the total mark from the individual scores.

We model Mark Integrity in ProVerif using the following correspondence:

```
query i:ID, a1:bitstring, ..., aN:bitstring, m1:bitstring, ..., mN:bitstring;
    event(assign(i, sum(m1, ... , mN))
          ==> event(marked(i,q1,a1,m1)) && ... && event(marked(i,qN,aN,mN)).
```

This above query succeeds if event(assign(i, sum(m1,..., mN)) is preceded by the events event(marked(i,q1,a1,m1)) ... event(marked(i,qN,aN,mN)).

A verification QEA that expresses Mark Integrity is depicted in Fig. 13. It accepts a run, if and only if, all the accepted answers are marked, and the total score (accumulated using variable s) is assigned to the corresponding candidate. For example, the run $accept(i, q, a).marked(q, a, b).\ assign(i, b)$ is accepted by Mark Integrity.

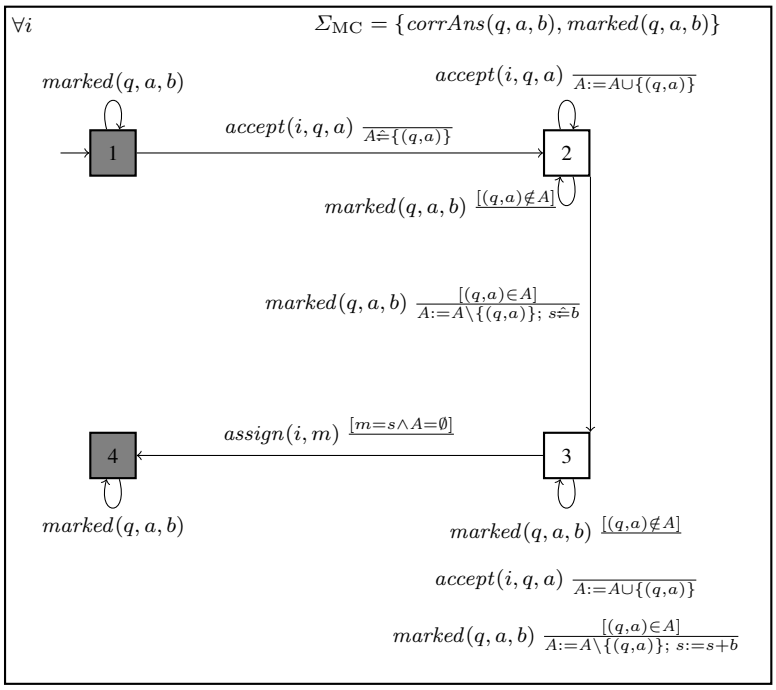


Fig. 13: A QEA for Mark Integrity.

5 Formal Verification of UGA Exam using ProVerif

We formally verify UGA exam protocol using ProVerif. The goal is to demonstrate the (in)security of UGA exam protocol for some properties. For security properties that are satisfied, we claim that monitoring them increases the trust in the implementation of the protocol. For properties that are not satisfied, attacks given by ProVerif should have been fixed before deployment of the protocol. Nevertheless, the attacks generated by the tool are precious information that should be taken into account by the monitors in order to detect such attacks that still exist in the running protocols. In all cases, such formal analysis is complementary to the monitoring approach (reported in Section 6).

We start by explaining UGA exam, then we present our modeling of UGA exam in ProVerif and the security results we obtained. We note that in this section, when we say that a property is satisfied, we mean that it holds for every possible run.

5.1 Exam Description

UGA exam consists of the following four phases.

Registration: Candidates have to register two weeks before the examination time. At registration, each candidate receives a username/password used for authentication at the examination time.

Examination: The exam takes place in a supervised room. Each candidate (a university student) is given a previously-calibrated tablet to pass the exam. The Internet access

is controlled: only IP addresses within a certain range are allowed to access the exam server. A candidate starts by logging in using his username/password. Then, he chooses one of the available exams by entering the exam code, which is provided at the examination time by the invigilator supervising the room. Once the correct code is entered, the exam starts and the first question is displayed. The pedagogical exam conditions mention that the candidates have to answer the questions in a fixed order and cannot get to the next question before answering the current one. A candidate can change the answer as many times as he wants before validating, but once he validates it, then he cannot go back and change any previously-validated answer. Note that all candidates have to answer the same questions in the same order. A question might be a one-choice question, multiple-choice question, open short-text question, or script-concordance question.

Marking: After the end of the examination phase, the grading process starts. For each question, all the answers provided by the candidates are collected. Then, each answer is evaluated anonymously by an examiner based on the correct answers provided by the exam authority for the corresponding question. After evaluating all the provided answers for all questions, the total mark for each candidate is calculated as the summation of all the scores attributed to his answers.

Notification: The marks are notified to the candidates. A candidate can consult his submission, obtain the correct answer and his score for each question.

5.2 Modeling of UGA exam in ProVerif

We model different (honest) exam parties as processes in ProVerif. We annotate these processes using exam events, which are defined in Section 2.1. For the attacker model, we consider the well-known Dolev-Yao attacker [DY83]. Moreover, we consider *corrupted parties* as threats may also come from parties other than the attacker, *e.g.,* a candidate bribes an examiner. Corrupted parties cooperate with the attacker, by revealing their secret data (*e.g.,* secret keys) to him, or taking orders from him (*e.g.,* how to answer a question). A corrupted party is modeled as a process similar to the honest party process, but the corrupted process outputs all the secret data on a public channel and also inputs the fresh data (*e.g.,* an answer to a question) from the attacker.

In symbolic verification, dishonest parties are usually not modeled and are assumed to be totally subsumed under the attacker, who owns their secrets and plays their roles. However in our case, considering dishonest parties allows the attacker to trivially falsify some properties by not emitting some related events, even if they occur. A corrupted party as defined above provides the attacker with the same additional capabilities as those provided by a dishonest party, except that in the case of a corrupted party the attacker cannot prevent the emission of some events.

To support secured communications, we use the well-known rewriting rule of deterministic symmetric encryption: $sdec(senc(m, k), k) = m$, which is presented in Section 2.1. In addition, we use the constructor $sum(b_1, \ldots, b_n)$ to represent the summation of the scores $b_1, \ldots, b_n$, where n is the number of exam questions. Note that, we do not need a rewriting rule that captures all algebraic properties of the arithmetic sum operator, as we only need a function which computes a candidate's total mark from his partial scores.

At registration, we use tables in ProVerif to support the login (username/password) assignment by the exam authority to a candidate. Finally, we assume secure communications (*i.e.,* messages are encrypted using $senc$ and a shared key) between the different parties involved in the exam process. In practice, this is achieved using a secure network.

5.3 Analysis using ProVerif

In this section, we discuss our analysis of UGA exam using ProVerif. The obtained results are summarized in Table 1. They are obtained using a standard PC (AMD A10-5745MQuad-Core 2.1 GHz, 8 GB RAM).

Note that, as we already mentioned in Section 4, properties Acceptance Assurance and Answer-Score Integrity cannot be expressed in ProVerif, and thus cannot be analyzed. Actually, ProVerif only supports the verification of reachability, correspondance and observational equivalence properties. Such a limitation emphasizes the necessity of runtime monitoring in order to consider properties where some events occur in the future.

Candidate Registration. For Candidate Registration, we considered an honest exam authority, honest examiners, and corrupted candidates. ProVerif shows that Candidate Registration is satisfied for unbounded number of candidates and unbounded number of questions.

Actually, Candidate Registration does not reveal a weakness in an exam design when it fails. However, it allows us to detect if a candidate tries to fake the system (*spoofing attacks*). Such an attack cannot be detected by ProVerif because, by default, the attacker cannot emit events. Note that, if we provide the attacker with the capability of emitting event submit, then Candidate Registration fails. The latter can be achieved in ProVerif using a dedicated process that takes some i, q, and a from the attacker, and then emits the event $submit(i, q, a)$. ProVerif confirms the previous statement.

Candidate Eligibility. We analyzed Candidate Eligibility in the presence of a honest exam authority, honest examiners and corrupted candidates. ProVerif shows that it is satisfied for the case of unbounded number of candidates and unbounded number of questions.

Answer Authentication. ProVerif found an attack against Answer Authentication when a corrupted exam authority is considered (with honest examiners and honest candidates). Corrupted exam authority discloses a candidate login, an exam question, as well as, the shared symmetric key to the attacker who then submits an answer so that the event accept is emitted without being preceded by an event submit. Note that Answer Authentication is satisfied if honest exam authority is considered and the protocol respects the types of the terms.

Acceptance Assurance. As we mentioned in Section 4, Acceptance Assurance cannot be verified using ProVerif.

Answer Singularity. ProVerif shows that Answer Singularity is violated when a candidate and exam authority are corrupted. A corrupted candidate can send two different answers to the same question, which will be accepted by a corrupted exam authority.

Questions Ordering. According to UGA exam, candidates can get all questions at the beginning of the exam, but, they have to validate them in order. Thus, Question Ordering is not relevant to UGA exam as it is obviously unsatisfied. However, we can verify a modified variant of Question Ordering that is: a candidate cannot set an answer to the next question (event change) before answering the current question (event accept). The intuition is that if a candidate changes the answer field of a question, he must have received the question previously. We denote this modified variant by Question Ordering*.

Question Ordering* can be expressed using a query similar to that of Question Ordering (cf. Section 4), but with replacing event get(i,a) with event change(i,q,a). ProVerif shows that Question Ordering* is satisfied by UGA exam.

Table 1: Results of the formal analysis of UGA exam using ProVerif, where ✓ means that the property is satisfied, ✗ means that there is an attack, and - means that we did not verify it with ProVerif.

Property	Result	Time (ms)
Candidate Registration	✓	34
Candidate Eligibility	✓	33
Answer Authentication	✗	18
Acceptance Assurance	-	-
Answer Singularity	✗	28
Question Ordering*	✓	58
Exam Availability (sub-queries)	✓	33
Answer-Score Integrity	-	-
Cheater Detection	✗	44
Marking Correctness	✓	66
Mark Integrity	✓	48

Exam Availability. We verified the two sub-queries related to Exam Availability. ProVerif shows that the two sub-queries are satisfied for any number of questions while considering corrupted candidates, honest examiners, and honest exam authority.

Answer-Score Integrity. This property cannot be verified using ProVerif (cf. Section 4).

Cheater Detection. We verified Cheater Detection for an exam with two candidates and three questions with threshold $d = 1$ for the distance. Note that, as mentioned in Section 4, we consider a distance function that computes the number of different answers between candidates. ProVerif shows, as expected, the existence of a run that violates Cheater Detection.

Marking Correctness. ProVerif shows that Marking Correctness is satisfied for any number of candidates and three questions when all parties are honest.

Note that, if a corrupted examiner is considered, the property can be easily falsified by him, *e.g.,* scoring an answer with a false mark. However in UGA exam, a candidate can consult how his answers are marked, check the marking, and object if there is something wrong. In other words, if the examiner is corrupted he can deliver a wrong mark, but this can be detected by the related candidate.

Mark Integrity. ProVerif shows that Mark Integrity is satisfied by UGA exam for one candidate and two questions.

6 Monitoring of the UGA Exam

THEIA company, which is specialized in e-formation platforms, has implemented UGA exam described in Section 5.1 under a project leaded by *University Grenoble Alpes*. In June 2014, the pharmacy faculty at UGA organized a first e-exam using the developed software, as a part of *Epreuves Classantes Nationales informatisées*.

We validated our framework by verifying two real e-exam runs passed with this system:

– Run 1 involves 233 candidates and contains 42 questions for a duration of 1h35.

– Run 2 involves 90 candidates and contains 36 questions for a duration of 5h20.

The resulting traces for these two runs are respectively of sizes 1.85 MB and 215 KB, and contain 40,875 and 4,641 events. All the logs received from the e-exam organizer are pseudonymised. Nevertheless for confidentiality reasons, we are not authorized to provide precise contextual information about the considered runs. Note that, these results cannot be generalized to any exam run, as they are only correct for the two considered runs.

We use MarQ[4] [RCR15] to model the QEAs and perform the verification. We first analyzed the properties defined in Section 4 (cf. Section 6.1). Then, after evaluating the results and discussing them with the exam developers, we conducted a further analysis specific for UGA exam (cf. Section 6.2). The result of our analysis together with the time required for MarQ to conclude using a standard PC (AMD A10-5745M–Quad-Core 2.1 GHz, 8 GB RAM), are summed up in Tables 2 and 3.

Note that, UGA exam developers did not log some data required for monitoring. Mainly, they did not log anything concerning the marking and the notification phase. Thus, we were not able to verify the properties: Answer-Score Integrity, Cheater Detection, Marking Correctness, and Mark Integrity. Nevertheless, we implemented these properties in MarQ and also validated them on toy runs as we expect to obtain the actual runs of the marking phase in the future.

6.1 First Analysis

Using MarQ tool, we performed off-line monitoring of the two exam runs. The results are summed up in Table 2. We found out that the first four properties: Candidate Registration, Candidate Eligibility, Answer Authentication, and Acceptance Assurance are all satisfied by both of the considered exam runs.

Concerning Answer Singularity, we found that it is violated by Run 1, but satisfied by Run 2.

For Question Ordering, as we reported in Section 4, it is not relevant to UGA exam. Thus, we monitored its modified variant: Question Ordering*, where we replaced event $get(i, q)$ by event $change(i, q, a)$ in the QEA depicted in Fig. 8. The analysis showed that Question Ordering* is violated by both runs.

Finally, MarQ proved that Exam Availability is satisfied by Run 1. However, Exam Availability is violated by Run 2: a candidate was able to change and submit an answer, which is accepted, after the end of the exam duration.

6.2 Post-Discussion Analysis

We have reported to the exam developers the three violations found in the first analysis concerning Answer Singularity, Question Ordering*, and Exam Availability. In the following, we present their answers and the results of the post-discussion analysis. The result of our additional analysis is summed up in Table 3.

[4] `https://github.com/selig/qea`

Table 2: Results of the initial off-line monitoring of two UGA exam runs, where ✓ means that the property is satisfied, ✗ means that there is an attack, the number between ( ) indicates the number of candidates that violate the property, and - means that we could not verify it with MarQ.

	Run 1		Run 2	
Property	Result	Time (ms)	Result	Time (ms)
Candidate Registration	✓	538	✓	230
Candidate Eligibility	✓	517	✓	214
Answer Authentication	✓	301	✓	255
Acceptance Assurance	✓	326	✓	309
Answer Singularity	✗ (1)	312	✓	293
Question Ordering*	✗ (2)	757	✗ (1)	389
Exam Availability	✓	518	✗(1)	237
Answer-Score Integrity	-	-	-	-
Cheater Detection	-	-	-	-
Marking Correctness	-	-	-	-
Mark Integrity	-	-	-	-

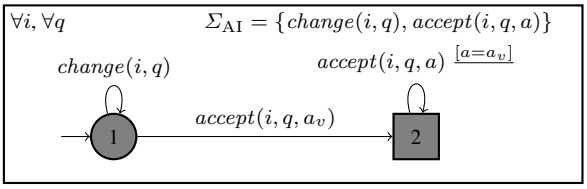


Fig. 14: A QEA for Answer Integrity.

Answer Singularity. The violation of Answer Singularity actually revealed a discrepancy between the initial specification and the current features of the e-exam software: a candidate can submit the *same answer* several times and this answer remains accepted. Consequently, an event $accept$ can appear more than once but only for same answer.

To check whether the failure of Answer Singularity is only due to the acceptance of same answer more than once, we updated the property Answer Singularity and its QEA presented in Fig. 6 by storing the accepted answer in a variable a_v, and adding a self loop transition on state (2) labeled by $accept(i,q,a)$ $\frac{[a=a_v]}{}$. We refer to this new (weaker) property as Answer Singularity*, which differs from Answer Singularity by allowing the acceptance of the same answer again; but it still forbids the acceptance of a different answer. We found out that Answer Singularity* is satisfied, which confirms that there is no critical issue of the system.

Furthermore, the UGA exam has a requirement stating that after validating an answer to a certain question, the writing field is "blocked" and the candidate cannot change it anymore. To verify this additional requirement, we defined property Answer Integrity which states that a candidate cannot change the answer after acceptance. Answer Integrity is expressed by the QEA depicted in Fig. 14. Note that, we allowed the acceptance of the same answer to avoid the bug found by Answer Singularity. Our analysis showed that Answer Integrity is satisfied by Run 1, but, it is violated in Run 2: at least one student was able to change the content of the writing field after having his answer accepted.

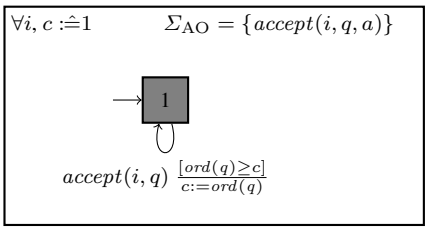


Fig. 15: A QEA for Acceptance Order.

Question Ordering*. Exam developers said that it should not be possible for a candidate to set an answer to a future question before validating the current question. Thus, the violation found against Question Ordering* represents a real intrusion into the system. After a manual check, we found that some candidates were able to skip certain questions without validating them, and then answer and validate the following questions. Note that, this violation represents a discrepancy between the implementation and the original specification. The specification states that a candidate cannot be able to get the next question before answering the current question, which appears to be possible in the implementation. In the previous section, we used ProVerif to analyze the original specification. This is why we did not find an attack against Question Ordering* (cf. Table 1). It confirms that monitoring is useful to detect "bug" or misbehaviors in real implementations.

To check whether the global order of the questions is still respected by all candidates (possibly with some question skipped), we defined Acceptance Order. A QEA that expresses Acceptance Order is depicted in Fig. 15. It fails if an event $accept(i, q_2, a_2)$ is followed by an event $accept(i, q_1, a_1)$ such that the order of question q_2 is greater than that of question q_1. We found that Acceptance Order is satisfied by both exam runs.

Exam Availability. Exam developers said that technically the software does not prevent candidates from validating answers, which remains accepted, after the end of exam time. Actually in practice, an exam organizer asks the candidates to logout and stop answering questions when the time ends, similarly to pencil-and-paper exams. Again, this represents a discrepancy between the implementation and the original exam specification. The specification states that a candidate is not allowed to answer any question after the exam time. This is why no attack was found by ProVerif against Exam Availability.

Table 3: Results of the post-discussion off-line monitoring of two UGA exam runs, where ✓ means that the property is satisfied, and ✗ means that there is an attack.

	Run 1		Run 2	
Property	Result	Time (ms)	Result	Time (ms)
Answer Singularity*	✓	751	✓	237
Answer Integrity	✓	641	✗	218
Acceptance Order	✓	697	✓	294

7 Related Work and Discussion

In this paper, we provide declarative definitions for e-exam properties, which allow both model-checking of exam specifications and monitoring of the real exam runs. Thus, we provide means to verify exam protocols from their design to their execution. Some of the results presented here have been published in our previous paper [KFL15]. This paper extends our results by providing additional properties, general abstract definitions, and further analysis of our case study (UGA exam) using ProVerif. To the best of our knowledge, our work is the first in addressing the verification of e-exams at runtime and security properties using an automatic cryptographic protocol verification tool.

In this section, we present related work on the verification of the security of e-exams and related domains. We classify the different approaches in two groups:

- Formal verification of protocol specifications (cf. [Bla13, CK16, ABC09, Ray10]); and
- Runtime verification and monitoring (cf. [HG05, LS09, FHR13]).

Formal verification of exam protocol specifications. Foley *et al.* [FJ95] have proposed a formalization for functionality and confidentiality requirements of Computer Supported Collaborative Working (CSCW) applications. They have illustrated their formalization using an exam example as a case study. Arapinis *et al.* [ABR12] have proposed a cloud-based protocol for conference management system that supports applications, evaluations, and decisions. They identify a few privacy properties (secrecy and unlinkability) that should hold in spite of a *malicious-but-curious* cloud, and they prove, using ProVerif, that their protocol satisfies them.

A formal framework, based on π-calculus, to analyze security properties such as authentication and privacy has been proposed in [DGK$^+$14a, DGK$^+$14b]. Our abstract definitions are inspired by their definitions of authentication properties using correspondence between events. However, they consider only five events: $register$, $submit$, $accept$, $assign$, and $marked$ (plus an event $distribute$ which is emitted when the authority distributes an exam-copy for marking, and which aims at supporting anonymous marking). Note that, they dealt with the exam questions as one entity and thus do not consider the notion of individual questions and scores. Such a limitation does not allow them to define properties such as Question Ordering, and only captures a weaker variant of properties such as Cheater Detection as two candidates answers will be either identical or not when they are represented using one entity as a whole. Moreover, all their definitions have a similar form: "event e_1 should be preceded by event e_2". In contrast, in our definitions, we use various relations between events which allows us to capture more complex requirements. For example, we make use of the relation: "event e_1 should be followed by the event e_2", as well as, composite relations built of the former and the latter one. We note also that, some of the properties defined in this paper, *e.g.,* Candidate Eligibility, resemble those proposed in [DGK$^+$14a, DGK$^+$14b]. Yet, we considered new properties such as Exam Availability and Cheater Detection. Furthermore, the approaches in [DGK$^+$14a, DGK$^+$14b] are limited to the formal analysis of exam specifications, and do not allow to monitor e-exam executions at runtime.

Bella *et al.* [BGLR15] proposed an e-exam protocol that does not require trusted third party (TTP), and analyzed it using ProVerif based on the framework proposed in [DGK$^+$14a] with some additional properties. System verification is also addressed in some other related domains *e.g.,* in e-auctions [DJL13], e-voting [KRS10, BHM08], e-reputation [KLL14], e-cash [DKL15b, DKL15a] and e-payment [Kat09]. For example, Katsaros [Kat09] proposed a methodology for modeling and validating protocols aiming at providing guarantees for

payment transaction *via* model checking. The author uses Colored Petri Nets for analyzing protocols while considering all execution scenarios and all players perspectives.

Dreier *et al.* [DGK+15] studied verifiability in exam protocols ensuring that, a protocol provides some "tests" that allow the different parties to check some statements about the exam. An example of verifiability property consists in verifying whether an exam provides a "test" that allows a candidate to check himself whether his mark is computed correctly. The authors defined several verifiability properties, and validated them by analyzing two exams using ProVerif. They relied on an event-based model similar to the one used in [DGK+14a] with the difference that they use sets (and $marking$ function) which correspond to events ($i \in R$ is equivalent to the emission of event $register(i)$) to define their properties.

Several tools for checking security protocol models have been proposed such as AVISPA [ABB+05], ProVerif [Bla01], Scyther [Cre08a,Cre08b], Tamarin [SMCB12,MSCB13], and AKISS [CcCK12]. Most of these tools aim at verifying secrecy (and strong secrecy) and authentication, and assume attacker models based on deduction rules similar to those introduced in the Dolev-Yao model. Basagiannis *et al.* [BKP07, BKP11] have proposed an attacker model that provides a base for the integration of attack tactics. The proposed approach allows them to combine the attack tactics to compose complex actions such as a Denial of Service attack. Thus, it allows the analyst to express requirements that are not restricted to the absence of secrecy and authentication. For our analysis, we use ProVerif tool, one of the most efficient cryptographic protocol verification tools [PLV09, CLN09, LP15]. We also consider Dolev-Yao attacker (with corrupted parties) which is suitable for our case study. ProVerif is convenient in our case as it supports events and allows us to check correspondences between them, which are respectively needed to express our event-based model and to verify our properties. However, the definitions of our properties are independent from the attacker model, and thus one can use any attacker model while verifying these properties, depending on the case study. Moreover, our properties have been designed in order to be instantiated and analyzed using other cryptographic protocol verification tools as long as they support their verification.

Runtime verification and monitoring. Offline monitoring of user-provided specifications over logs has been addressed in the context of several tools in the runtime verification community (cf. [BBF14,FNRT15,RHF16,BFB+17] for descriptions of some of the main tools): Breach [Don10] for Signal Temporal Logic, RiTHM [NJW+13] for (variant of) Linear Temporal Logic, LogFire [Hav15] for rule-based systems, and JavaMOP [JMLR12] for various specification formalisms provided as plugins. MarQ [RCR15] is a tool for monitoring Quantified Event Automata [BFH+12, Reg14]. Moreover, offline monitoring was successfully applied to several industrial case studies, *e.g.,* for monitoring financial transactions with LARVA [CP13], and monitoring IT logs with MonPoly [BCE+14].

Our choice of using QEAs stems from two reasons. First, QEAs is one of the most expressive specification formalism to express monitors. The second reason stems from our interviews of the engineers who were collaborating with us and responsible for the development of the e-exam system at UGA. To validate our formalization of the protocol and the desired properties for e-exams, we presented the existing alternative specification languages. QEA turned out to be the specification language that was most accepted and understood by the engineers. Moreover, MarQ came top in the 1^{st} international competition on Runtime Verification, showing that MarQ is one of the most efficient existing monitoring tools for

both off-line and on-line monitoring, as seen in the last editions of the competition on Runtime Verification [BBF14, FNRT15, RHF16][5].

8 Conclusions, Lessons Learned, and Future Work

Conclusions. We define an event-based model of e-exams, and identify several fundamental exam properties based on events. We define these properties, and then express them as ProVerif queries and Quantified Event Automata (QEA). We introduce Auditing QEAs, as extension of QEAs, which additionally report all entities that violated the property. Auditing QEAs are given in the Appendix A. We validate our properties by analyzing real e-exams at UGA. First, we model and verify the specification of UGA exam using ProVerif. We find counterexamples against the specifications. Because of ProVerif limitation, we were not able to analyze two out of eleven properties Second, we perform off-line verification of certain exam runs using the MarQ tool. Analyzing logs of real e-exams requires less than a second on a regular computer. As a result we find violations in the exam executions, and discrepancies between the specification and the implementation.

Lessons learned. The analysis conducted in this paper allows us to derive the following lessons:

- The analysis confirms: 1) the necessity of verifying e-exam systems, and 2) the effectiveness of model checking and runtime monitoring in the verification of exam systems. The analysis confirms that, even though the protocol of the e-exam was specified, model checking it allowed to discover some attacks. These attacks 1) emphasize the significance of model-checking protocols before implementing them so that one could fix potential flaws beforehand, and 2) could be used as a reference in the monitors designing process. The analysis also confirmed that runtime monitoring is 1) effective in efficiently finding discrepancies between the implementation and the specifications with QEAs, and 2) needed as a complementary approach to model checking.
- QEAs was a suitable formalism as it met the requirements of our study. First, we were able to design security properties and make sure that the engineers who designed and developed the e-exam system understood them properly. Second, QEAs are supported by an efficient tool implementation letting us analyze the logs of real e-exam runs. We note that, because of the lack of logs about the marking and notification phases, we were not able to analyze all properties.
- The UGA exam case study clearly demonstrates that the exam developers and designers do not think to log these two phases during which there is less interaction with the candidates. However, we believe that verifying the marking phase is essential since a successful attempt from a bribed examiner or a cheating student can be very effective; and we thus encourage universities and educational institutions to incorporate logging features in their e-exam systems.

Future Work. Several avenues for future work are open by this paper. First, we intend to analyze more existing e-exams from other universities. We also expect to receive all required logs for a complete analysis in the near future. Moreover, we plan to perform on-line verification with our monitors during live e-exams. By doing so, we plan to study 1) how to decentralize monitors (cf. [BF12]) to have them running on the candidate devices, and 2)

[5] See also `http://rv2014.imag.fr/monitoring-competition/results`.

to what extent runtime enforcement (cf. [Fal10]) can be applied during a live e-exam run. This requires an access to the source code of the e-exams for instrumentation purposes. Finally, we plan to study more expressive and quantitative properties that can detect colluding students through similar answer patterns.

Acknowledgements The authors would like to thank François Géronimi from THEIA, Daniel Pagonis from TIMC-IMAG, and Olivier Palombi from LJK for providing us with a description of e-exam software system, for sharing with us the logs of some real french e-exams, and for validating and discussing the properties presented in this paper. The authors also thank Giles Reger for providing us with help on using MARQ. Finally, we thank Jannik Dreier from LORIA for his help with ProVerif.

This research was conducted with the support of the Digital trust Chair from the University of Auvergne Foundation.

This article is based upon work from COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology).

References

[AB05a] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *J. ACM*, 52(1):102–146, January 2005.

[AB05b] Xavier Allamigeon and Bruno Blanchet. Reconstruction of attacks against cryptographic protocols. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*, pages 140–154. IEEE Computer Society, 2005.

[ABB+05] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.

[ABC09] Martín Abadi, Bruno Blanchet, and Hubert Comon-Lundh. Models and proofs of protocol security: A progress report. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2009.

[ABR12] Myrto Arapinis, Sergiu Bursuc, and Mark Ryan. Privacy supporting cloud computing: Confichair, a case study. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, volume 7215 of *Lecture Notes in Computer Science*, pages 89–108. Springer, 2012.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 104–115. ACM, 2001.

[BBF14] Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone. First international competition on software for runtime verification. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2014.

[BCE+14] David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. Scalable offline monitoring. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 31–47, Cham, 2014. Springer International Publishing.

[BF12] Andreas Klaus Bauer and Yliès Falcone. Decentralised LTL monitoring. In Giannakopoulou and Méry [GM12], pages 85–100.

[BFB+17] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien

Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. First international competition on runtime verification: rules, benchmarks, tools, and final results of crv 2014. *International Journal on Software Tools for Technology Transfer*, pages 1–40, 2017.

[BFH+12] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In Giannakopoulou and Méry [GM12], pages 68–84.

[BGLR15] Giampaolo Bella, Rosario Giustolisi, Gabriele Lenzini, and Peter Y. A. Ryan. A secure exam protocol without trusted parties. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, volume 455 of *IFIP Advances in Information and Communication Technology*, pages 495–509. Springer, 2015.

[BHM08] Michael Backes, Catalin Hritcu, and Matteo Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF '08, pages 195–209, Washington, DC, USA, 2008. IEEE Computer Society.

[BKP07] Stylianos Basagiannis, Panagiotis Katsaros, and Andrew Pombortsis. Intrusion attack tactics for the model checking of e-commerce security guarantees. In Francesca Saglietti and Norbert Oster, editors, *Computer Safety, Reliability, and Security, 26th International Conference, SAFECOMP 2007, Nuremberg, Germany, September 18-21, 2007.*, volume 4680 of *Lecture Notes in Computer Science*, pages 238–251. Springer, 2007.

[BKP11] Stylianos Basagiannis, Panagiotis Katsaros, and Andrew Pombortsis. Synthesis of attack actions using model checking for the verification of security protocols. *Security and Communication Networks*, 4(2):147–161, 2011.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, CSFW '01, pages 82–, Washington, DC, USA, 2001. IEEE Computer Society.

[Bla02] Bruno Blanchet. From secrecy to authenticity in security protocols. In Manuel V. Hermenegildo and Germán Puebla, editors, *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, volume 2477 of *Lecture Notes in Computer Science*, pages 342–359. Springer, 2002.

[Bla13] Bruno Blanchet. Automatic verification of security protocols in the symbolic model: The verifier proverif. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 54–87. Springer, 2013.

[BM15a] Ezio Bartocci and Rupak Majumdar, editors. *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*. Springer, 2015.

[BM15b] Ezio Bartocci and Rupak Majumdar, editors. *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*. Springer, 2015.

[BSC16] Bruno Blanchet, Ben Smyth, and Vincent Cheval. *ProVerif 1.90: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*, 2016. Originally appeared as Bruno Blanchet and Ben Smyth (2011) ProVerif 1.85: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial.

[CcCK12] Rohit Chadha, Ştefan Ciobâcă, and Steve Kremer. Automated verification of equivalence properties of cryptographic protocols. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 108–127. Springer, 2012.

[CK16] Véronique Cortier and Steve Kremer. Formal models for analyzing security protocols: Some lecture notes. In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 33–58. IOS Press, 2016.

[CLN09] Cas J. F. Cremers, Pascal Lafourcade, and Philippe Nadeau. Comparing state spaces in automatic security protocol analysis. In Véronique Cortier, Claude Kirchner, Mitsuhiro Okada, and Hideki Sakurada, editors, *Formal to Practical Security - Papers Issued from the 2005-2008 French-Japanese Collaboration*, volume 5458 of *Lecture Notes in Computer Science*, pages 70–94. Springer, 2009.

[Cop13] Copeland, L. School cheating scandal shakes up atlanta. USA TODAY, April 2013. Available at `http://www.usatoday.com/story/news/nation/2013/04/13/atlanta-school-cheatring-race/2079327/`.

[CP13] Christian Colombo and Gordon J. Pace. Fast-forward runtime monitoring — an industrial case study. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification: Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, pages 214–228, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Cre08a] Cas J. F. Cremers. The scyther tool: Verification, falsification, and analysis of security protocols. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.

[Cre08b] Cas J. F. Cremers. Unbounded verification, falsification, and characterization of security protocols by pattern refinement. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 119–128. ACM, 2008.

[DGK+14a] Jannik Dreier, Rosario Giustolisi, Ali Kassem, Pascal Lafourcade, Gabriele Lenzini, and Peter Y. A. Ryan. Formal analysis of electronic exams. In Mohammad S. Obaidat, Andreas Holzinger, and Pierangela Samarati, editors, *SECRYPT 2014 - Proceedings of the 11th International Conference on Security and Cryptography, Vienna, Austria, 28-30 August, 2014*, pages 101–112. SciTePress, 2014.

[DGK+14b] Jannik Dreier, Rosario Giustolisi, Ali Kassem, Pascal Lafourcade, Gabriele Lenzini, and Peter Y. A. Ryan. Formal security analysis of traditional and electronic exams. In Mohammad S. Obaidat, Andreas Holzinger, and Joaquim Filipe, editors, *E-Business and Telecommunications - 11th International Joint Conference, ICETE 2014, Vienna, Austria, August 28-30, 2014, Revised Selected Papers*, volume 554 of *Communications in Computer and Information Science*, pages 294–318. Springer, 2014.

[DGK+15] Jannik Dreier, Rosario Giustolisi, Ali Kassem, Pascal Lafourcade, and Gabriele Lenzini. A framework for analyzing verifiability in traditional and electronic exams. In Javier Lopez and Yongdong Wu, editors, *Information Security Practice and Experience - 11th International Conference, ISPEC 2015, Beijing, China, May 5-8, 2015. Proceedings*, volume 9065 of *Lecture Notes in Computer Science*, pages 514–529. Springer, 2015.

[DJL13] Jannik Dreier, Hugo Jonker, and Pascal Lafourcade. Defining verifiability in e-auction protocols. In Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng, editors, *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, pages 547–552. ACM, 2013.

[DKL15a] Jannik Dreier, Ali Kassem, and Pascal Lafourcade. Automated verification of e-cash protocols. In *E-Business and Telecommunications - 12th International Joint Conference, ICETE 2015, Colmar, France, July 2022, 2015, Revised Selected Papers*, pages 223–244, 2015.

[DKL15b] Jannik Dreier, Ali Kassem, and Pascal Lafourcade. Formal analysis of e-cash protocols. In Mohammad S. Obaidat, Pascal Lorenz, and Pierangela Samarati, editors, *SECRYPT 2015 - Proceedings of the 12th International Conference on Security and Cryptography, Colmar, Alsace, France, 20-22 July, 2015.*, pages 65–75. SciTePress, 2015.

[Don10] Alexandre Donzé. Breach, A toolbox for verification and parameter synthesis of hybrid systems. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 167–170. Springer, 2010.

[DY83] D. Dolev and Andrew C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

[FAI15] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. On verifying hennessy-milner logic with recursion at runtime. In Bartocci and Majumdar [BM15a], pages 71–86.

[Fal10] Yliès Falcone. You should better enforce than verify. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2010.

[FFJ+12] Yliès Falcone, Jean-Claude Fernandez, Thierry Jéron, Hervé Marchand, and Laurent Mounier. More testable properties. *STTT*, 14(4):407–437, 2012.

[FFM09] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In Saddek Bensalem and Doron A. Peled, editors, *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*, volume 5779 of *Lecture Notes in Computer Science*, pages 40–59. Springer, 2009.

[FFM12] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[FHR13] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In Manfred Broy, Doron A. Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.

[Fig15] Figaro. Etudiants: les examens sur tablettes numériques appellés à se multiplier. Press release, January 2015. Available at `goo.gl/ahxQJD`.

[FJ95] Simon N. Foley and Jeremy Jacob. Specifying security for computer supported collaborative working. *Journal of Computer Security*, 3(4):233–254, 1995.

[FNRT15] Yliès Falcone, Dejan Nickovic, Giles Reger, and Daniel Thoma. Second international competition on runtime verification CRV 2015. In Bartocci and Majumdar [BM15a], pages 405–422.

[GM12] Dimitra Giannakopoulou and Dominique Méry, editors. *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*. Springer, 2012.

[Hav15] Klaus Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, 17(2):143–170, 2015.

[HG05] Klaus Havelund and Allen Goldberg. Verify your runs. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, volume 4171 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2005.

[JMLR12] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. Javamop: Efficient parametric runtime monitoring framework. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 1427–1430. IEEE, 2012.

[Kat09] Panagiotis Katsaros. A roadmap to electronic payment transaction guarantees and a colored petri net model checking approach. *Information & Software Technology*, 51(2):235–257, 2009.

[KFL15] Ali Kassem, Yliès Falcone, and Pascal Lafourcade. Monitoring electronic exams. In Bartocci and Majumdar [BM15b], pages 118–135.

[KKL+02] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Computational analysis of run-time monitoring - fundamentals of java-mac. *Electr. Notes Theor. Comput. Sci.*, 70(4):80–94, 2002.

[KLL14] Ali Kassem, Pascal Lafourcade, and Yassine Lakhnech. Formal verification of e-reputation protocols. In Frédéric Cuppens, Joaquín García-Alfaro, A. Nur Zincir-Heywood, and Philip W. L. Fong, editors, *Foundations and Practice of Security - 7th International Symposium, FPS 2014, Montreal, QC, Canada, November 3-5, 2014. Revised Selected Papers*, volume 8930 of *Lecture Notes in Computer Science*, pages 247–261. Springer, 2014.

[KRS10] Steve Kremer, Mark Ryan, and Ben Smyth. Election verifiability in electronic voting protocols. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *Computer Security – ESORICS 2010: 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, pages 389–404, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[LP15] Pascal Lafourcade and Maxime Puys. Performance evaluations of cryptographic protocols verification tools dealing with algebraic properties. In Joaquín García-Alfaro, Evangelos Kranakis, and Guillaume Bonfante, editors, *Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers*, volume 9482 of *Lecture Notes in Computer Science*, pages 137–155. Springer, 2015.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

[NJW+13] Samaneh Navabpour, Yogi Joshi, Chun Wah Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. Rithm: a tool for enabling time-triggered runtime verification for C programs. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 603–606. ACM, 2013.

[PLV09] Vanessa Terrade Pascal Lafourcade and Sylvain Vigier. Comparison of cryptographic verification tools dealing with algebraic properties. In Pierpaolo Degano Joshua Guttman, editor, *sixth International Workshop on Formal Aspects in Security and Trust, (FAST'09)*, Eindhoven, Netherlands, nov 2009.

[PZ06] Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.

[Ray10] Sandip Ray. *Scalable Techniques for Formal Verification*. Springer, 2010.

[RCR15] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. Marq: Monitoring at runtime with qea. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, pages 596–610, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[Reg14] Giles Reger. Automata Based Monitoring and Mining of Execution Traces. PhD thesis, University of Manchester, 2014.

[RHF16] Giles Reger, Sylvain Hallé, and Yliès Falcone. Third international competition on runtime verification - CRV 2016. In Yliès Falcone and César Sánchez, editors, *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, volume 10012 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 2016.

[SMCB12] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In Stephen Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 78–94. IEEE, 2012.

[Wat14] R. Watson. Student visa system fraud exposed in BBC investigation. `http://www.bbc.com/news/uk-26024375`, feb 2014.

Appendix A. Auditing QEAs

For each property, we provide an auditing QEA except for Candidate Registration (which is given in Section 4) and Cheater Detection (which is auditing by itself). An auditing QEA reports some data in case of failure.

Candidate Eligibility with Auditing. An auditing QEA that expresses Candidate Eligibility is depicted in Fig. 16. It collects in a set F all the candidates from which an answer is accepted without being registered to the exam.

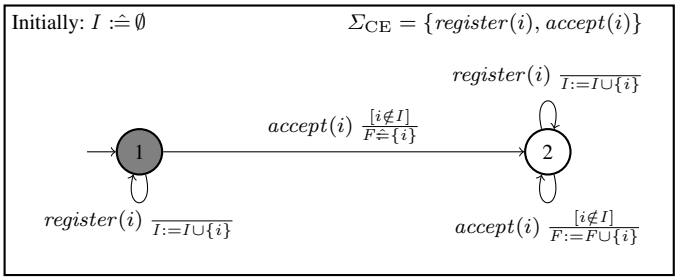


Fig. 16: A QEA for Candidate Eligibility with Auditing.

Answer Authentication with Auditing. An auditing QEA that expresses Answer Authentication is depicted in Fig. 17. It collects in a set F all the unsubmitted answers that are accepted together with the corresponding candidates and questions. Note that A in the QEA in Fig. 5 is a multi-set.

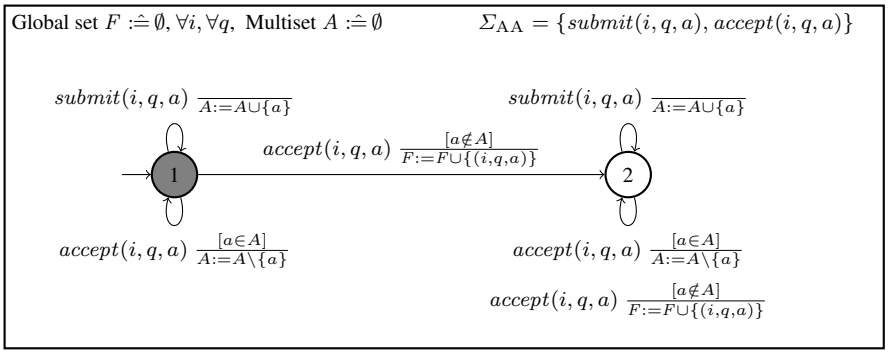


Fig. 17: A QEA for Answer Authentication with Auditing.

Answer Singularity with Auditing. An auditing QEA that expresses Answer Singularity is depicted in Fig. 18. It collects in a set F all further answers (after the first one) that are accepted to the same question from the same candidates.

Acceptance Assurance with Auditing. An auditing QEA that expresses Acceptance Assurance is depicted in Fig. 19. It collects in a set F all candidates who submit an answer to a question but no answer from them is accepted for this question.

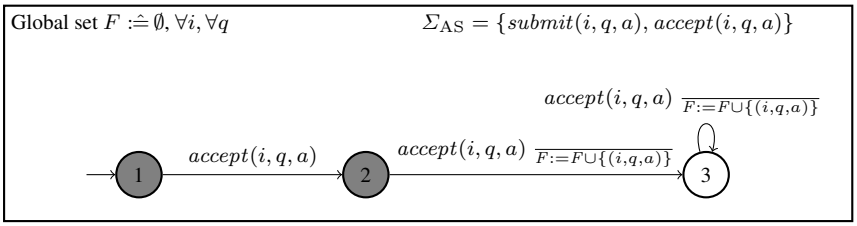


Fig. 18: A QEA for Answer Singularity with Auditing.

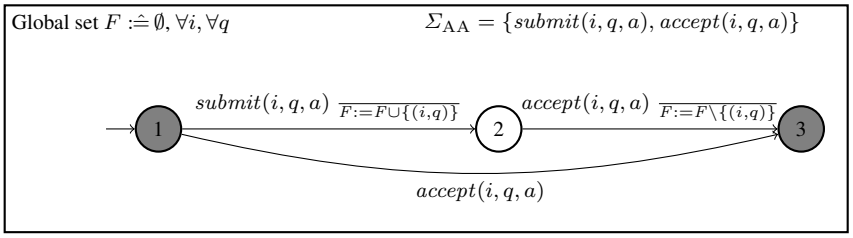


Fig. 19: A QEA for Acceptance Assurance with Auditing.

Questions Ordering with Auditing. An auditing QEA that expresses Question Ordering is depicted in Fig. 20. It collects in a set F all candidates who get a higher-order question before their answer to the current question is accepted.

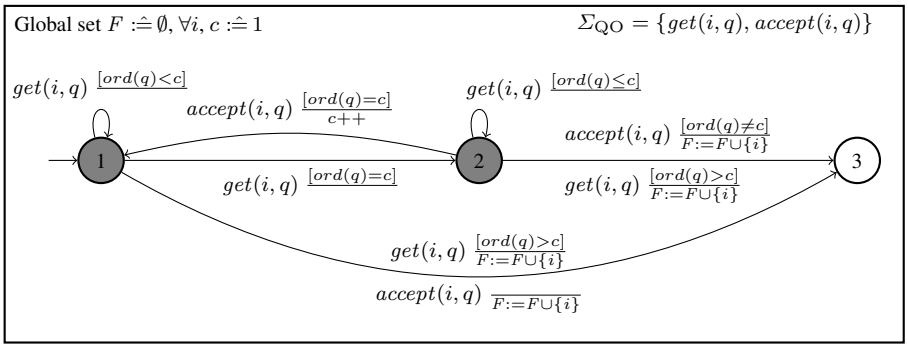


Fig. 20: A QEA for Questions Ordering with Auditing.

Exam Availability with Auditing. An auditing QEA that expresses Exam Availability is depicted in Fig. 21. It collects in a global set F all the answers (together with the corresponding questions and candidates) that are accepted before the event $start$ or after the event $finish$.

Answer-Score Integrity with Auditing. An auditing QEA that expresses Answer-Score Integrity is depicted in Fig. 22. It collects in a set F all the triplets (q, a, s) where $corrAns(q, a, s)$ comes after the event $start$.

Marking Correctness with Auditing. An auditing QEA that expresses Marking Correctness is depicted in Fig. 23. It collects in a global set F all the answers that are marked incorrectly. We relax the constraint stating that, for a question q, no event $corrAns(q, a, b)$ can be emitted after the marking phase has started. More simply, an answer that is not declared as a correct answer yet is considered as a wrong answer.

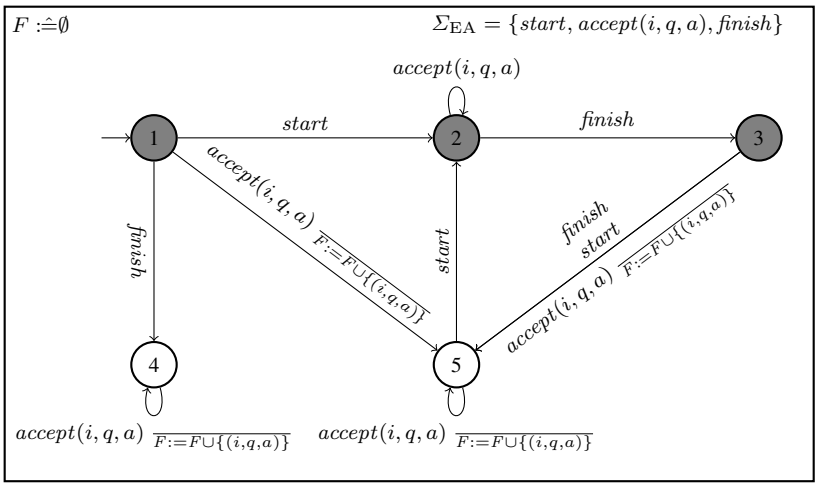


Fig. 21: A QEA for Exam Availability with Auditing.

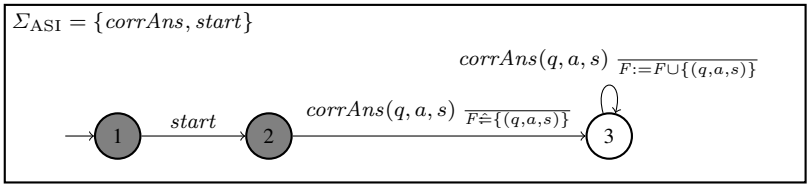


Fig. 22: QEA for Answer-Score Integrity with Auditing

Mark Integrity. An auditing QEA that expresses Mark Integrity is depicted in Fig. 24. It collects 1) in a set F_1 all the candidates who have their first assigned marked incorrectly, and 2) in a set F_2 all further marks assigned to the candidates regardless if they are correct or not.

Appendix B. Flexible Exam Duration

We define another variant of Exam Availability that supports exams with flexible starting and duration times, we call it Exam Availability with Flexibility. To define it, we extend the exam model defined in Section 3.2 with event $begin(i,t)$, which is emitted when candidate i begins his examination phase, at time t. We also define a function $dur(i)$ which specifies exam duration of candidate i and thus also his exam ending time.

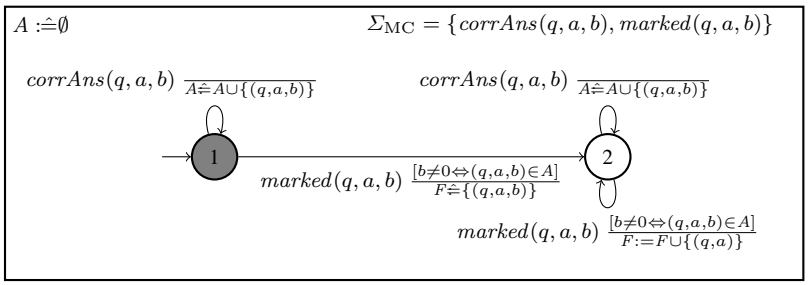


Fig. 23: A QEA for Marking Correctness with Auditing.

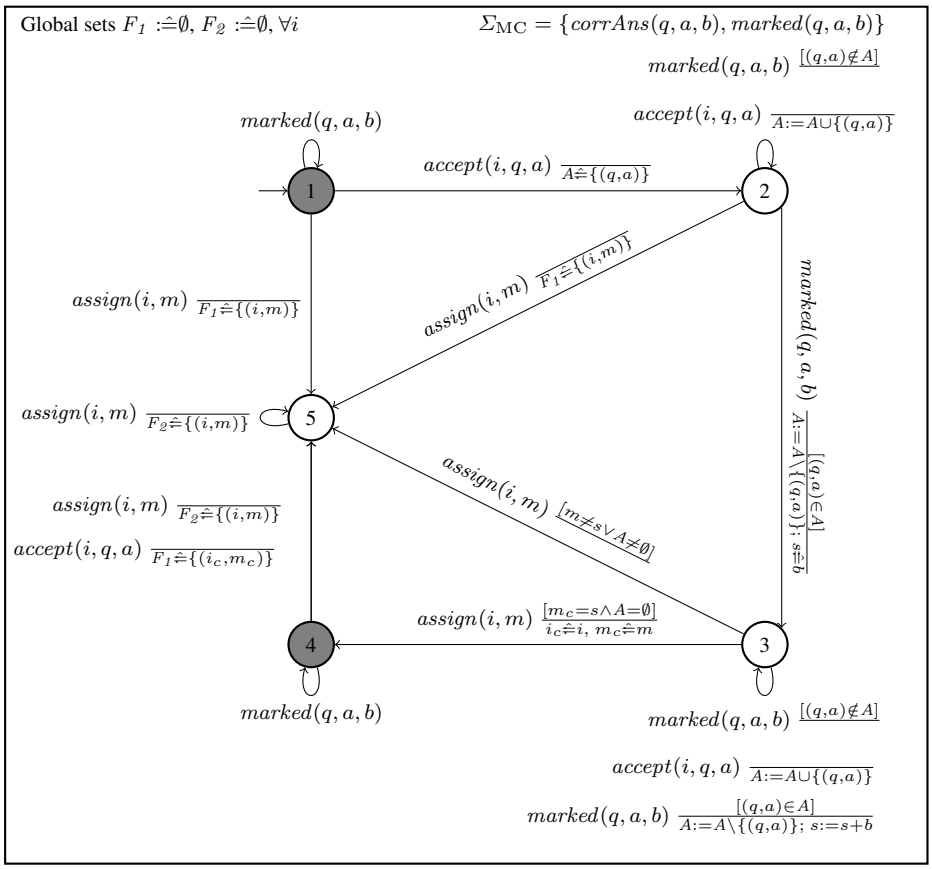


Fig. 24: A QEA for Mark Integrity with Auditing.

Definition 14 (Exam Availability with Flexibility). Let $dur(i)$ be the exam duration for candidate i. An exam run satisfies Exam Availability with Flexibility if

- event $begin(i, t_s)$ is preceded by event $start(t)$; and
- event $accept(i, t)$ is preceded by event $begin(i, t_b)$, and is followed by event $finish(t)$ where $t - t_b \leqslant dur(i)$.

A candidate can validate answers till the end of his allocated duration unless he exceeds the global ending time specified by event $finish(t)$. Similarly to event $start(t)$, event $begin(i)$ has to occur only once *per* candidate.

We express Exam Availability with Flexibility in ProVerif using events $begin(i, t)$ and $end(i, t)$. The ProVerif queries are parameterized now with the candidate identity as follows:

- `query i:ID; event(accept(i)) ==> event(begin(i)).`
- `query i:ID; event(end(i)) ==> event(begin(i)).`

A verification QEA and an auditing QEA that express Exam Availability with Flexibility are depicted in Fig. 25 and Fig. 26, respectively. The auditing QEA collects in a set F all the candidates from which an answer is accepted outside the allowed duration.

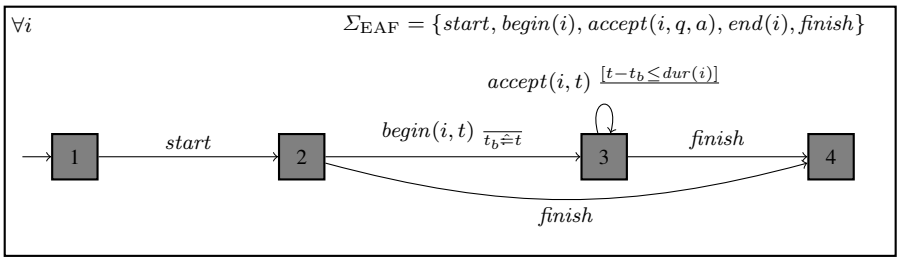


Fig. 25: A QEA for Exam Availability with Flexibility.

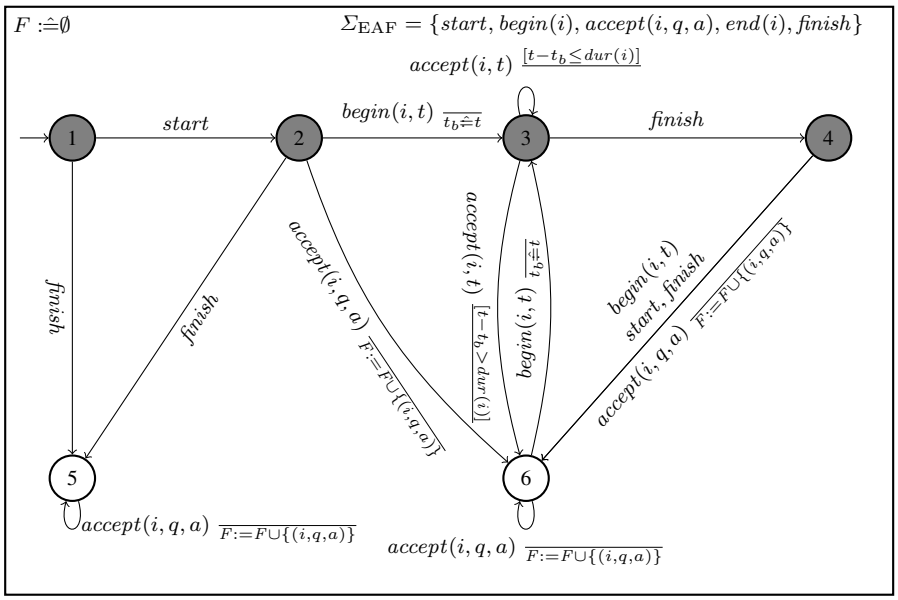


Fig. 26: A QEA for Exam Availability With Flexibility Auditing.