OASIS: Weakening User Obligations for Security-critical Systems

Thein Than Tun* Amel Bennaceur* Bashar Nuseibeh†
*The Open University, UK *†Lero, Ireland
firstname.lastname@open.ac.uk

***Abstract*—Security-critical systems typically place some requirements on the behaviour of their users, obliging them to follow certain instructions when using those systems. Security vulnerabilities can arise when users do not fully satisfy their obligations. In this paper, we propose an approach that improves system security by ensuring that attack scenarios are mitigated even when the users deviate from their expected behaviour. The approach uses structured transition systems to present and reason about user obligations. The aim is to identify potential vulnerabilities by weakening the assumptions on how the user will behave. We present an algorithm that combines iterative abstraction and controller synthesis to produce a new software specification that maintains the satisfaction of security requirements while weakening user obligations. We demonstrate the feasibility of our approach through two examples from the e-voting and e-commerce domains.**

***Index Terms*—System security, user behaviour, e-voting**

I. Introduction

A recent survey by the UK Government shows that human behaviour such as staff not adhering to organisational policies contributes to 42% of security incidents [1]. For such reasons, users are often seen as the "weakest link" in the security chain. In this paper, we focus on security vulnerabilities that emerge when users fail to satisfy their obligations fully. We argue that in many cases, there are alternative designs for the software in which the system satisfies its security requirements even when a user deviates from the expected behaviour, and we propose a systematic approach to achieve them.

Research in the field of usable security has argued that the deviation of user behaviour is often justified because many of these instructions are arbitrary (e.g., mixing of character types in passwords), unrealistic (e.g., requiring different passwords for each account [2]), ineffective (e.g., users having to change passwords every 90 days leads to weaker passwords [3]), or cumbersome (e.g., requiring users to confirm before every critical action). Security mechanisms can therefore cause friction in the way users want to interact with systems, and their usability is therefore critical for their acceptance by users (and ultimately their effectiveness). However, focusing on user behaviour alone is insufficient and equal, if not bigger, importance should be given to the design of the associated software systems [4].

In this paper, we propose an approach, called OASIS (Obligations, Attack scenarios, SpecIfication abstraction, and Synthesis) that relaxes some of the obligations of the users while satisfying security requirements by generating a revised specification of the associated software. The contribution of this paper is threefold.

1) *A modelling process* that makes explicit the components of systems, their interaction, and users obligations. We decompose the environment into a set of interacting components in a way similar to problem frames [5]. To support automated reasoning, we use existing formalisation of protocol behaviour using finite state machines [6] for each component. The aim is to make precise user obligations and clarify the assumptions made for the design of the software, which in turn helps generate potential attack scenarios that arise from failures of the users to comply with those obligations.
2) *An algorithm to generate revised specification for weakened obligations.* Once an attack scenario is identified, we propose an algorithm that integrates abstraction and synthesis techniques in order to derive a revised specification that fixes the identified vulnerability and maintain the security requirements satisfied under weaker assumptions about user behaviour. By abstracting the specification first, the controller is able to change the sequencing of actions rather than only blocking or allowing some actions in an adversarial environment.
3) *A demonstrator* that shows that the revised software specification satisfies the security requirements even though the users deviate from theirs obligations. We implemented the OASIS approach on top of an existing model checker, LTSA [7], and evaluated it with two examples from the e-voting and e-commerce domains. For example, we automatically revise the behaviour of an e-voting application to avoid vote flipping even if users forget to confirm their vote.

The remainder of this paper is structured as follows. Section II introduces the e-voting illustrative example. Section III gives an overview of the approach. Section IV presents the formalism we use to model socio-technical systems. Section V details the OASIS approach and Section VI reports on the implementation and the experiments conducted to validate our approach. Section VII examines related work. Finally, Section VIII concludes the paper and discusses future work.

II. Motivating Example: Electronic Voting

In 2010 in Kentucky, eight co-conspirators were sentenced to a total of more than 156 years in prison for electoral fraud [8]. One of the methods used to steal votes involved the touch-screen voting machine called iVotronic manufactured by ES&S. When a voter enters the booth, the voter's interaction with the voting machine can be described as follows.

Step 1. Key in the voter ID & personal pin, and move to the next step (*password*)
Step 2. Select candidate and move to the next step (*select*)
Step 3. Vote selected candidate and move to the next step (*vote*), or return to Step 2 (*back*)
Step 4. Confirm the vote (*confirm*), or return to Step 3 (*back*)

In the above e-voting system, an important obligation of the voter is that they complete the entire sequence until the vote is confirmed in Step 4. The use of *confirm* action at the end is a common heuristic for user interface design to ensure that mistakes are prevented before a user performs a critical action (this is known as the "error prevention" heuristic [9]).

Vulnerability and Attack. When voters used this system, there were incidents where some voters mistakenly thought that their vote had been counted after the *vote* action, and exited the voting booth. The corrupt election officials, who were allowed to go inside the booth after the voter had exited the booth, went inside, tapped '*back*' twice, selected their preferred candidate, before confirming the vote. This attack is called "vote flipping".

Fixing the vulnerability. There are a number of potential ways in which this vulnerability can be fixed including:

(i) Make recruitment and monitoring of election officials more stringent.
(ii) Install booth doors preventing the voter from leaving before confirming, or authenticate every person entering the booth. This however makes voting booths less portable.
(iii) Give more effective instructions to voters so that they know not to leave before confirming their vote. For example, this may mean providing clearer signs on the display indicating the progress of their interactions with the system. However, interaction design techniques alone are often insufficient to build secure systems and need to be combined with software and security engineering methods [10].
(iv) Modify the behaviour of the e-voting software so that a corrupt official cannot easily change the vote after the voter has left the booth, whether the voter has confirmed their vote or not. This is a software engineering challenge, and our approach will focus on finding such a solution.

There is a long line of work on formal verification of security properties in electronic voting systems at different levels of granularity: from cryptographic primitives, to high-level protocols, to software, to socio-technical systems (see Fig. 1). We give examples of approaches dedicated to each level of granularity on the left hand-side and illustrate their meaning with the e-voting example on the right hand-side. The OASIS approach focuses on the system level and is concerned with identifying security vulnerabilities and resolving them by controlling the software behaviour, without necessarily changing, constraining, or controlling the user behaviour as described in the following section.

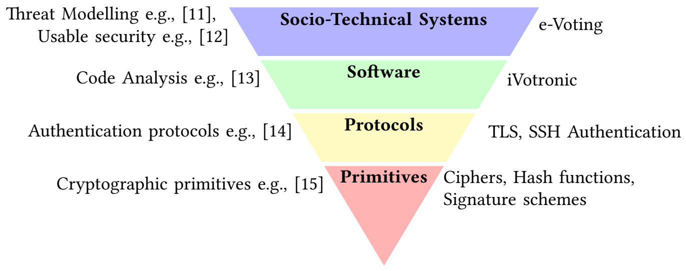

Threat Modelling e.g., [11],
Usable security e.g., [12]
Socio-Technical Systems
e-Voting
Code Analysis e.g., [13]
Software
iVotronic
Authentication protocols e.g., [14]
Protocols
TLS, SSH Authentication
Cryptographic primitives e.g., [15]
Primitives
Ciphers, Hash functions, Signature schemes

Fig. 1: Security: Layers of abstraction

III. Overview of the OASIS Approach

To describe our approach more precisely, we formalise it using Jackson and Zave's framework for requirements engineering [16]. This framework makes explicit the relationship between requirements, specifications, and environment properties. It allows us to describe our approach precisely without prescribing a specification or verification technique.

The OASIS (Obligations, Attack scenario, SpecIfication abstraction, and Synthesis) approach aims to revise the software design to maintain the satisfaction of requirements while weakening the expected behaviour of the users. To do so, OASIS starts by modelling the socio-technical system in order to identify the interactions between the software and the environment generally, and the users in particular. This steps constructs an argument showing how the security requirement R_s is satisfied by the software specification M_1 in the environment E_1 that places strong obligations on the users (i.e. $M_1, E_1 \models R_s$) as depicted in Fig. 2-❶. The software specification is described using Finite State Machines (FSM) [6]. The environment is made up of interacting components or agents, each of which represented using an FSM. Requirements are represented using Linear Temporal Logic (LTL) [17]. The entailment $M, E \models R$ holds when the software specification (machine) M satisfies the requirement R in the environment E. Modelling is explained in Section IV.

OASIS uses then the model of the environment (E_1) and the successive weakening of the user behaviour to generate attack scenarios, producing the model of an adversarial environment (E_2) where the security requirement is violated ($M_1, E_2 \not\models R_s$) as depicted in Fig. 2-❷.

Once an attack scenario is identified, the behaviour of the implemented software system, together with the model of the adversarial environment and the security requirements are used to identify a revised specification: that is, we seek M_2 such that $M_2, E_2 \models R_s$. The revision stage (see Fig. 2-❸) generates successively more abstract model M of the existing specification M_1, which is then used to synthesise a controller C that ensures the satisfaction of the security requirement in the adversarial environment. The revised specification M_2 is the result of controlling the abstracted specification and removes the security vulnerability without changing the behaviour of E_2. The revision is minimal with respect to M_1, that is it involves modification of fewest alphabets in the behavioural model of M_1. We acknowledge that minimality

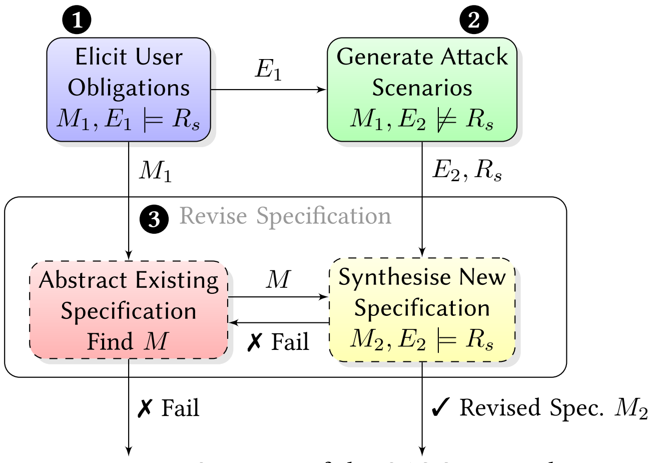


Fig. 2: Overview of the OASIS approach

may not be the sole criterion for selecting the appropriate revision. For example, sub-minimal revision may offer better usability or performance. Indeed, while we focus on the security requirements R_s for simplicity, other requirements can be specified without loss of generality. We will explain the modelling process in Section IV before detailing each step of the approach in Section V.

IV. Modelling System Behaviour

The behaviour of a component specifies its interaction with the environment and models how the actions of its alphabet are coordinated in order to achieve the specified functionality. We build upon state-of-the-art approaches to formalise behaviour using Finite State Machines (FSM) [6].

*Definition 1 (**Finite State Machine (FSM)**):* An FSM is a quintuple $\langle Q, \Sigma, \delta, q_0, F\rangle$ where

- Q is the set of states,
- Σ is the set of actions denoting the alphabet of the FSM,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- q_0 is the initial state, and
- $F \subseteq Q$ is the set of accepting states.

Example. The voting software behaviour can be modelled as shown in Fig. 3. Generally, we will abstract away a sequence of software-controlled action immediately followed by the user-controlled action into one action. For example the action *password* could be described as two actions, the software asking for password and the voter entering the correct password.

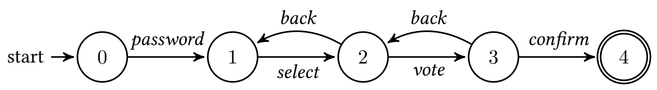


Fig. 3: Behaviour of the voting machine

Modelling socio-technical systems using a single FSM has many limitations:

1) Software is not the system, and the vulnerability is not in software. In the attack scenario discussed above, corrupt officials exploit the system behaviour not covered by the model in Fig. 3. For example, the model only says how the software and voters interact but does not say anything about who can enter and exit the booth at what points. There is a need to model the behaviour of the voter, attacker, the booth as well as voting software, individually and together in order to identify and fix the vulnerability.
2) The software cannot observe every action in the system. For example, the software can observe the candidate selected but not whether someone has entered or exited the booth. Once authenticated, the software does not know who it is interacting with. The assumption that actions subsequent to user authentication are always performed by the authenticated user is unwarranted and is a security risk.
3) Actions such as *password*, are special because we can assume that only the voters can perform the *password* action. Both voter and attacker can perform all other actions.

Therefore, analysing the structure and behavioural properties of the components involved in the system is central to identifying and fixing certain security vulnerabilities. The OASIS approach uses Structured FSM (SFSM) to model socio-technical systems. Similarly to Problem Frames [5] that decompose systems into multiple domains, SFSM structures the socio-technical system into multiple interacting components, each of which described as an FSM.

*Definition 2 (Structured FSM (**SFSM**)):* An SFSM is a tuple $\langle \mathcal{B}, m, controls\rangle$ where

- $\mathcal{B}$ is a set of finite state machines where αF denotes the alphabets of the FSM $F \in \mathcal{B}$ associated with one component,
- $m \in \mathcal{B}$ is the FSM associated with the software (machine), and
- $controls : \mathcal{B} \times \mathcal{B} \to 2^{\Sigma}$ where $\Sigma = \bigcup_{F \in \mathcal{B}} \alpha F$ is a function that specifies that the shared alphabet $a = controls(f_1, f_2)$ is controlled by the FSM f_1 and observed by another f_2.

Note that $controls(f_1, f_1)$ designates the *hidden* actions that are internal to the component $f_1 \in \mathcal{B}$. We use the shorthand $controls(f_1) = \bigcup_{f \in \mathcal{B}} controls(f_1, f)$ to identify all externally visible alphabets .

We assume the following syntactic rules to ensure SFMS models are well-formed.

- Every alphabet is controlled by a component (either shared or hidden).
- Every component controls or observes some shared alphabet (no component with hidden alphabets only).
- No alphabet is both hidden and shared at the same time.

The security properties are described using the alphabet Σ of the SFSM while the alphabet that the machine can observe and control is described using Σ_m, called *specification phenomena.*

Definition 3 (Specification phenomena): Specification phenomena Σ_m of a system described by an SFSM is:

$$\Sigma_m = \bigcup_{f \in \mathcal{B}} controls(m, f) \cup controls(f, m)$$

To avoid unwanted synchronisation between the FSM of different components, an alphabet controlled by multiple components in different state machines are relabelled by prefixing them with their respective components so that they are distinguishable in the composed model. For example, both the alphabets of the voter and the election official include the *enter* and *exist* actions in their interaction with the booth. In order to avoid the FSM of the voter and election official from synchronising they are both suffixed: $controls(\textit{Voter}, \textit{Voting Booth}) = \{\textit{v.enter}, \textit{v.exit}\}$ and $controls(\textit{Election Official}, \textit{Voting Booth}) = \{\textit{eo.enter}, \textit{eo.exit}\}$. The behaviour of the voting booth is a synchronisation of the voter and election official behaviour. Fig. 4 shows that the voter and the election official cannot be in the booth at the same time. For now, we consider one voter and one voting official in the model, but we consider a multitude of them in Section VI-C.

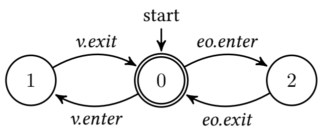


Fig. 4: Voting booth behaviour

System modelling using SFSM addresses the three aforementioned limitations:

1) We can distinguish between three kinds of actions [18]: (i) machine-controlled actions the environment observes ($\bigcup_{f \in \mathcal{B}} controls(m, v)$), (ii) environment-controlled actions the machine observes ($\bigcup_{f \in \mathcal{B}} controls(f, m)$), and (iii) environment-controlled actions the machine cannot control or observe ($\Sigma - \Sigma_m$). By explicitly modelling the behaviours of the software (machine) and environmental components, vulnerabilities due to interactions between components can be analysed. For example, we can express who can enter the voting booth and what is the state of the voting machine when they enter.
2) By including actions of the environment which the machine cannot observe in the analysis but restricting the alphabets of the machine to the specification phenomena, we can surface obligations placed on the environment that may turn out to be too strong. In other words, the SFSM makes explicit the interaction between the different components, in particular between the machine and the users as well as between the users and the environment.
3) By allowing that environment-controlled actions may be controlled by a number of components, we can model issues related to identity and authentication.

Using SFSM to describe both the structure and behaviour of the software system allows us to elicit and reason about user obligations more precisely, which we will explain in the following section.

V. The OASIS Approach

A. Eliciting User Obligations

In order to describe the user obligations assumed by an implemented system, we model the system behaviour including the behaviour of the machine and environment. When modelling the machine, the focus is on the interactions with the user rather than actions internal to the machine. The first step is to decompose the behavioural model according to its different components, which involves developing an FSM for each component. We then extend the model of user behaviour with actions users control that machine cannot observe. This may also involve adding new states as well as new transitions.

Example. In the e-voting system, the assumed behaviour of the voter can be developed by expanding the model in Fig. 3 to include the *enter* and *exit* actions, which voter can perform but the machine cannot observe as shown in Fig. 5. The model emphasises that the voter enters the booth before initiating a voting session and that the voter exits the booth after confirming the vote, hence the booth hides these actions from the machine. However, the voter can actually exit the booth at any time/state without the machine/software being able to detect this action. Since there are common alphabets controlled by both voter and election official, they are relabelled using action suffixes.

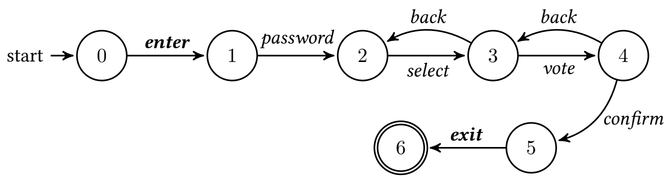


Fig. 5: Assumed voter behaviour

Specifying security requirements Besides modelling the system behaviour, we must also represent the security requirement (R_s) and show that the composed system behaviour satisfies the security requirement.

Example. For the e-voting system, let us consider some possible formulations of the security requirement "No Vote Flipping" and they include the following (a voting session is a trace of the behavioural model in Fig. 5):

R1 the confirmed vote in every voting session is for the candidate selected by the voter in the session
R2 the person who confirms the vote must be the voter of the session
R3 in every session, it must be the voter who chooses the candidate, confirms the vote.
R4 election officials can never select a candidate after the voter has entered the password

Although the requirement "No Vote Flipping" in Fig. 3 is the relationship between the candidates each voter has selected and the final vote tally, the behavioural model does not say anything about the vote directly. As a result, we cannot talk about the vote count in R1 directly using LTL. We will restrict our requirements to properties we can express in LTL, namely, safety and progress properties. This often means rewriting the requirements [19]. We choose the stronger formulation of the requirement for "No Vote Flipping", i.e. R4, which can be expressed in LTL as $\square(\textit{v.vote} \rightarrow \square(\neg \textit{eo.select}))$.

Since the machine cannot observe who performs the *confirm* action, the specification cannot rely on the occurrence of *confirm* action to satisfy the security requirement. The voter in Fig. 5 is obliged to confirm the vote before leaving the voting booth. It is easy to see that if the user fulfils the obligations, we can verify that the security property "No Vote Flipping" is satisfied by the system through the entailment $M_1, E_1 \models R_s$, where E_1 is the environment where the voter complies with their obligations.

B. Generating Attack Scenarios

Once user obligations have been identified, this step aims to relax those obligations, creating a weaker, and more realistic, environment E_2 and identifying possible violations to the security requirements with the existing machine specification, i.e. counterexamples to the entailment $M_1, E_2 \models R_s$. In the following, we describe each of these two steps and illustrate them using the e-voting example.

Relaxing user obligations. The aim is to weaken the behaviour of the users so that 1) they are allowed to perform actions in any order they like and 2) the machine can constrain user behaviour only by defining valid action sequences the user controls and the machine observes. In effect, the user behaviour is a single state with all transitions returning to it.

Example. Fig. 6 shows the weakened behaviours of the voter and the election official, where they are allowed to perform actions in any order after they have entered the voting booth. This can be compared with the initial assumed behaviour of, for example, the voter described in Fig. 5. In other words, the machine must control the behaviour allowed by the voter rather than obliging the voter to follow a specific behaviour. Note also that apart from the *password* action, the machine cannot differentiate between the actions performed by the voter and a legitimate or malicious official.

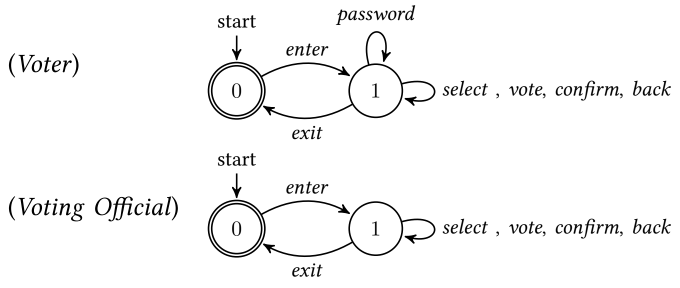


Fig. 6: Weakened behaviours of the voter and the voting official

Identifying requirements violation. Once the user obligations are weakened, we can define a more realistic behavioural model of the environment by composing the behavioural models of the component and reverify the requirement entailment. Counterexamples are possible attack scenarios. The behaviour of the system is the composition of all components in the system. We can then use model checking to verify that $M_1, E_2 \not\models R_s$. Notice that E_2 refines E_1 by weakening user behaviours and extending user alphabets, which may lead to the violation of security requirements.

Example. The behaviours of the voter, election official, voting booth and the machine are then composed to give the revised system behaviour as shown in Fig. 7. This is the behavioural model of the environment in which the e-voting system will be used. Given the model, several counterexamples to the security requirement that the election official cannot select the vote can be generated. The attack scenario discussed in Section II is highlighted in red dotted lines, but the model highlights several other attack scenarios as well. In short, once the system has reached one of the red states (the voter has given the password and has left the voting booth without having confirmed the vote), the election official can confirm the vote and thus violating the security requirement.

The machine can control only the ordering of the actions *password*, *select*, *vote*, *confirm* and *back* . The machine cannot observe the actions *exit*, and *enter* of the voter and election official. It is difficult to define the bad states because it depends on who is controlling the transitions; even in Fig. 7, states such as 21 and 22 or 31 and 32 are indistinguishable for the machine; so are events such as *v.select* and *eo.select*. What is needed is a re-design of the behaviour of the software system in order to prevent election officials to alter a vote even when the voter exits without confirming.

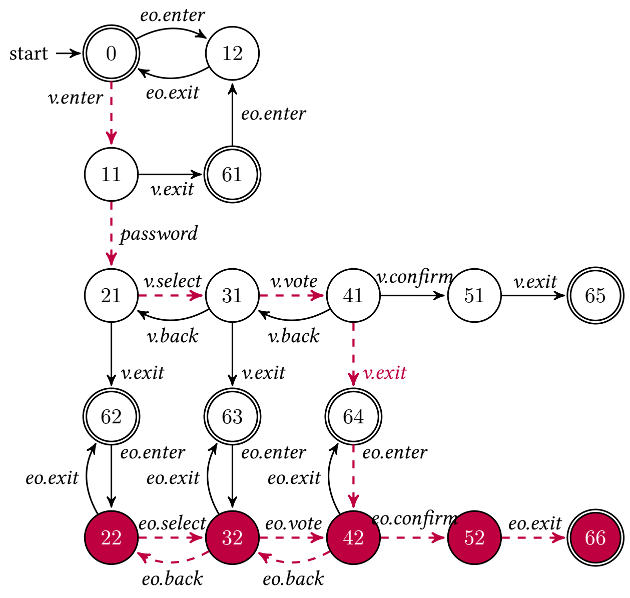


Fig. 7: Expanded E-voting system behaviour

C. Revising Specification

Once an attack scenario is identified, we then need to revise the specification M_1 into a specification M_2 that satisfies the requirements R_s in the weakened environment E_2 (E_2 refines E_1). There are three main cases for this revision depending on the relationship between M_1 and M_2.

Case 1: *M_1 refines M_2 and $\alpha M_2 \subseteq \alpha M_1$.* If the revised specification M_2 is a refinement, which implies that the alphabet and the transitions are subsets of those in the original specification, then finding δ such that $\delta, M_1, E_2 \models R_s$ is a simple controller synthesis problem. For example, controller synthesis approaches can find a fix where all *back* actions are removed from the specification as depicted in Fig. 8. This means that as long as the voter leaves the voting booth after selecting the candidate, the attacker cannot modify the vote.

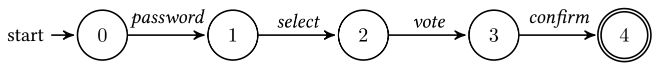


Fig. 8: A revised specification of the voting machine (the most constrained)

D'Ippolito *et al.* [20] propose a multi-tier framework whereby a stack of mediators are synthesised to satisfy stronger requirements when making stronger assumptions about the environment. For example, a two level stack would be as follows.

Synthesise M_1 such that $E_1, M_1 \models R_1$,
Synthesise M_2 such that $E_2, M_2 \models R_2$,
E_2 simulates E_1, and M_2 simulates M_1

where in the higher tier, some strong assumptions about the environment are made and strong guarantees provided while weaker assumptions (E_2 simulates E_1) are made in the lower first tier but also weaker guarantees are provided. Nevertheless, it is not always the case that the revised version refines the original one while satisfying the same requirements. For example, if we still need to avoid vote flipping while allowing voter to change their selection before confirming, then a different behavioural design of the machine is required.

Case 2: *M_2 does not refine M_1 and $\alpha M_2 \nsubseteq \alpha M_1$.* States and alphabets can be added to and removed from the description of the environment (which may also affect the specification phenomena). This amounts to modifying the environment so that the existing specification satisfies the requirement. For example, this could be the addition of new states and transitions to prevent the voter from leaving the booth before confirming or enable the machine to observe or control the *enter* and *exit* actions. As we have discussed in Section II, this is outside the scope of this work.

Case 3: *M_2 does not refine M_1 and $\alpha M_2 \subseteq \alpha M_1$.* An implemented design is a commitment to particular behaviour where alternatives may be possible. Since some of the user obligations have been removed, the users interacting with the system perform actions in different sequences. As a result, the machine can also modify its behaviour without changing the set of actions it executes. Therefore, we seek M_2 that is similar structurally to M_1 and satisfies the requirements $M_2, E_2 \models R_s$. The aim of this step is to revise some design decisions in order to find the implementation most similar to M_1 that maintains the satisfaction of the requirements and prevents the attack scenario identified.

The primary focus of this work is on the last case and we define a revision algorithm that works in two iterative steps: abstracting the existing specification and then synthesising a controller that uses this abstract specification to synthesise a new one that satisfies the given requirements.

Algorithm 1 for generating a revised specification takes as inputs the behaviour model of the machine component

Algorithm 1: Generate Revised Specification

```
input  : $M_1 = \langle Q, \Sigma_m, \delta, q_0 \rangle$: machine behavioural
         model
         $E_2$: Environment behavioural model
         $R_s$: Security requirements
output : $\{M_2, Fail\}$
1  $\mathcal{L} \leftarrow$ orderedPowerSet($\Sigma_m$);
2  while $\mathcal{L} \neq \emptyset$ do
3      $L \leftarrow$ nextMinimalElement($\mathcal{L}$);
4      $M \leftarrow$ minimize ($\langle Q, (\Sigma_m - L) \cup \{\tau\}, \delta_1, q_0 \rangle$)
         where $\forall q \in Q, \forall a \in \Sigma_m \cdot \delta(q,a) = \delta_1(q,\tau)$ if
         $a \in L$ and $\delta(q,a) = \delta_1(q,a)$ otherwise;
5      $N \leftarrow$ minimize ($\langle Q, L \cup \{\tau\}, \delta_2, q_0 \rangle$)
         where $\forall q \in Q, \forall a \in \Sigma_m \cdot \delta(q,a) = \delta_2(q,a)$ if
         $a \in L$ and $\delta(q,a) = \delta_2(q,\tau)$ otherwise;
6      $F \leftarrow$ M || N ;
7      synthesise $C$ such that $C, F, E_2 \models R_s$;
8      if $C \neq Null$ then
9          $M_2 \leftarrow$ C || M;
10         return $M_2$;
11     end
12 end
13 return Fail;
```

M_1, and that of the environment composed behavioural models of other components, and the security requirements R_s expressed as an LTL property. The algorithm produces either the behavioural model for the revised specification M_2 or the special symbol *Fail* as its output.

First the algorithm constructs the poset $\mathcal{L}$ from the alphabets in the input model M_1, where members of the power set are ordered by the subset relationship. In other words, $\mathcal{L} = (2^{\Sigma_m}, \subseteq)$. This produces a lattice where the smallest element is the empty set and the greatest element is M_1 (Line 1). This lattice is used to loop on possible abstractions of M_1 by selecting the minimal element L from the lattice and removing it together with the complement subset from the lattice, i.e. $\mathcal{L} \leftarrow \mathcal{L} - \{L, \Sigma_m - L\}$ (Line 3). Note that the first element in the lattice is the empty set ($\emptyset$) and therefore in the first iteration of the loop, M is assigned M_1.

For each minimal element L, the algorithm constructs an abstract state machine F that interleaves all the actions in L. In a process algebraic form, this is done by composing two state machines M and N. The state machine M is obtained by removing all actions in L from M_1. This is done by replacing all the transitions involving those actions by silent transitions ($\delta(q,a) = \delta_1(q,\tau)$ if $a \in L$) and then minimising the resulting state machine (Line 4).

The algorithm generates another state machine N by first hiding alphabets other than those from the set L, before minimising it (Line 5). F, the parallel composition of M and N (Line 6) is based on interleaving semantics where the two components synchronise on shared actions (note that $\alpha M \cap \alpha N = \emptyset$ by construction) and can progress by alternating at any rate the execution of their other actions. M_1 is one potential refinement of F in which a particular sequences of the actions in L is chosen.

The algorithm then attempts to synthesise the controller C such that it can allow or block the actions of F in order to satisfy the security requirements in the adversarial environment $E2$ (Line 7). In other words, the synthesised controller ensures that the composition of the behaviours of the different components is deadlock-free and reaches a state where the requirements are satisfied. There are many approaches to controller synthesis [21]–[27], and they differ in their assumptions (e.g., system behaviour is deterministic) and the expressiveness of the goals involved (e.g., dealing with safety, liveness, or general LTL properties). Rather than focusing on a specific approach for synthesis, we show how these techniques can be extended through abstraction.

If the synthesis succeeds, the revised specification M_2 is the parallel composition of C and M; the algorithm successfully terminates by returning M_2 (Line 10). If the synthesis fails, the algorithm chooses the next set of alphabets from the lattice and repeats the loop. If no controller C is found at the end of the loop, the algorithm returns the special symbol *Fail* (Line 13).

Proposition 1: M generated in every iteration of the loop in Algorithm 1 is an abstraction of M_1.
Intuitively, M is obtained by removing some of the actions of M_1 and then later on changing their positions in F to obtain different refinements. Refinement gives an intuitive notion of correctness (especially for safety properties), and it has been applied in the stepwise design and implementation of software systems, starting from their more abstract specification [28].

We prove the contrapositive that M_1 is a refinement of M. As noted, in the first iteration of the loop, M is M_1, and therefore M_1 refines M in that iteration. As M is obtained by hiding some of the alphabet L of M_1 while maintaining the same transition set which implies inclusion of the trace sets since. By construction $\forall q \in Q, \forall a \in \Sigma_m \cdot \delta(q,a) = \delta_1(q,\tau)$ if $a \in L$ and $\delta(q,a) = \delta_1(q,a)$ otherwise where L is the hidden alphabet in the given transition, δ and δ_1 the transition functions of M_1 and M respectively.

Example. In the e-voting example, if we choose a subset $L = \{$*password*$\}$, we obtain the abstract machine M, and the corresponding complementary N depicted in Fig. 9. Their parallel composition F means that the *password* action can be placed at any stage between the actions of M, i.e. $\{$*select*, *vote*, *confirm*, *back*$\}$. Placing it before the *select* action as in the original specification M_1 depicted in Fig. 3 is only one option or possible refinement of F. We can run the synthesis to generate a controller that controls the actions of F to make the security requirements satisfied. However, multiple possible revised specifications can be obtained as depicted in Fig. 10. When minimality is defined in terms of overlap of transitions with the specification M_1 then the first case (a) can be discarded. Alternatively, a partial specification of M_2 or some additional requirements can drive the choice between cases (b) and (c). For example, case (b) prevents the election officials changing the vote but not confirming it.

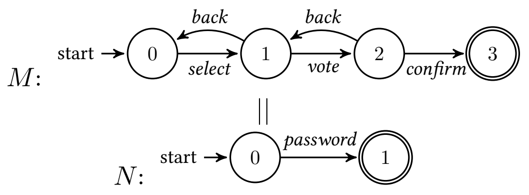


Fig. 9: Abstraction and composition of the voting machine

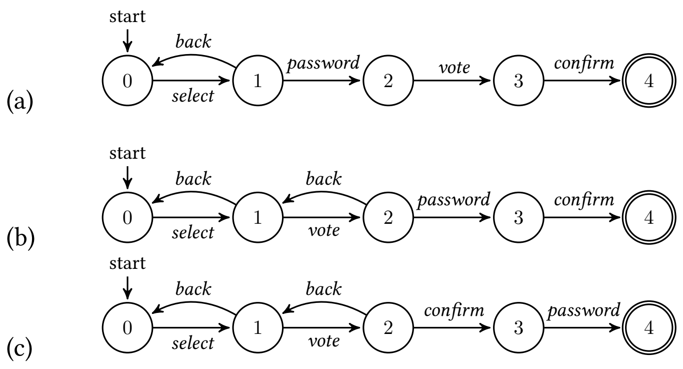


Fig. 10: Possible revisions of the e-voting machine

VI. Evaluation

This section discusses the implementation of Algorithm 1 together with theoretical and practical evaluation of the algorithm. The evaluation covers the following three properties of our approach: 1) *Complexity*: we examine the theoretical aspects of the OASIS approach and discuss its performance. 2) *Feasibility*: we describe the implementation of the approach on top of an existing model checker and show how it can be used to identify an attack and revise the behaviour in two scenarios. 3) *Scalability*: we measure the time and the size of the search space when dealing with an increasingly complex specification, which we obtain by varying the number of voters, voting officials, and voting booths. We show that, although theoretically complex, OASIS can be applied at runtime in practical cases (models with up to around 500 states and 4000 transitions). Finally, we discuss the limitations and possible enhancements of our approach.

A. Complexity

In the general case, controller synthesis is known to be computationally expensive [29]. Let us consider the synthesis of a controller C that ensures the requirement R_s assuming E_2, i.e. the controller satisfies the formula $\phi \equiv E_2 \Rightarrow R_s$. In other words, $C \models \phi$. When ϕ is expressed as an LTL formula, controller synthesis may reach complexity of double exponent in the size of ϕ [29]. Yet for safety formulas as well as subclasses of liveness formulas (e.g., GR(1) [30] or SGR(1) [26]), the synthesis problem can be solved in polynomial time. Our approach does not aim to improve the synthesis algorithm *per se*. Instead, we rely on the extensive work that has been developed in the area of controller synthesis and reduce the size of the models provided as input to the synthesis algorithm. In addition, the abstraction allows us to redesign behaviour by moving actions, which is not possible with existing synthesis approaches. The time complexity of performing minimisation with strong equivalence is $O(kn)$ for an FSM with k transitions and n states [31] in general and and $O(klogn)$ for more efficient algorithms [32]. However, the model checker we used, LTSA, implements a simpler algorithm proposed by Holzmann [33], which is less efficient theoretically but proves faster for practical uses [34].

The minimisation and synthesis are performed for a stack of abstract FSM starting from the original/existing specification and gradually abstracting it, until all possible partitions of the alphabet have been explored. In the worst case scenario, this would necessitate 2^{m-1} iterations for an existing machine specification with a size m alphabet. However, although the algorithm has an exponential complexity $O(2^m)$, one can make use of the partial specification S to bound the possible abstraction. In the e-voting example, constraints such as *select* must precede *vote*, which must precede *confirm* considerably reduce the search space. This is similar in principle to protocol projections [35] with the image protocol explored through possible partitions. A partial specification can guide the exploration and reduce the processing time.

B. Feasibility

In order to validate our approach and give evidence of its functional correctness, we have implemented the approach using the LTSA model checker [7]. LTSA is a free verification tool that can check safety and liveness properties of communicating processes. We built on the existing capabilities of LTSA for behavioural analysis and used composition to capture the specification of the controller. In the following, M_1 is the behavioural model given in Fig. 3. E_2 is the parallel composition of the models for voting booth (Fig. 4), the voter and the election official (Fig. 6). The attack scenario is generated by checking the following LTL property.

```
assert NoEOSelectAfterVPassword
    = [](v.vote -> [](!eo.select))
```

which produces the attack scenario in less than 1ms.

The complete specification and its explanation are available at https://github.com/amelBennaceur/oasis.

Case study: Session Hijacking via Cross-Site Scripting

To show applicability to other domains, we consider an e-commerce web application. When a user is authenticated, the application stores a session cookie (a random string) inside the client browser, which is then used in subsequent interactions between the client and the server. A class of attacks called *session hijacking* happen when an attacker manages to obtain the session cookies. This can happen when the user unwittingly executes a malicious script hidden inside a page or disguised as a link which reads the cookie and sends it to the attacker.

Step 1. The user provides username and password to log in.
Step 2. If credentials are valid, the e-commerce website stores a session cookie.
Steps 3 and 4. The user browses the catalogue and adds items to basket.
Step 5. The user chooses a delivery address.
Step 6. The user may make payment to complete the purchase.
Step 7. The user logs out from the site.

The session cookie generated and stored on the user's computer in Step 2 needs to be validated by the server in each interaction. The cookies are valid for the entire session until the user logs out in Step 6, or a period of inactivity has occurred. In such cases, the user is asked to log in again creating a new session cookie.

Vulnerability and Attack. A user may log in and keep the session cookie on the computer for a length of time for a number of reasons, such as taking a long time to make purchase decisions and forgetting to log off after deciding not to buy anything. An attacker may steal the cookie via Cross Site Scripting: a script that reads the cookie and sends it to the attacker's server may be executed when the user clicks on a link. This means that there are four states (states 2 to 5), when the system is vulnerable to session hijacking attacks.

An important obligation placed on the user is that after they have logged in, they make the purchase and log out, AND never click on a link containing a script that steals the cookie before logging out. The OASIS approach aims to weaken this obligation while keeping the system secure.

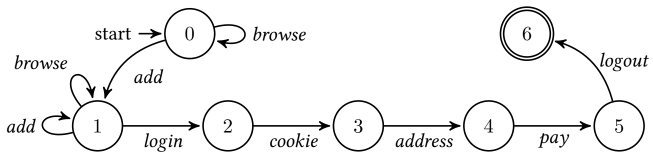


Fig. 11: Revised specification of a shopping session

One revised specification according to our approach is to move the login and cookie events to immediately before the address event so that the user only has to log in when they are about to pay (see Fig. 11). The system is now vulnerable to session hijacking attack in three states (states 3 to 5). Although the number of vulnerable states is reduced by one, the eliminated state tends to last longer, and therefore poses a significant security threat.

It is worth noting that this pattern of authenticating users (again) before escalating privilege can be observed in many popular online applications: banking applications tend to authenticate users when checking transactions, and again before making transfers.

Modern web browsers also have partial defences against session hijacking attacks: (i) HttpOnly cookies cannot be read by client-side scripts, (ii) when the attribute SameSite is set to strict, cookies cannot be sent to a different domain, and (iii) Content Security Policies can prevent execution of scripts that are not white-listed by the application. These techniques are complementary in that they are designed to restrict access to cookies inside the browser, while our focus is to reduce the number of states when cookies are stored.

C. Scalability

In order to evaluate the scalability of our algorithm using LTSA, we increase the size of the model by increasing the number of instances of Voter and Election Official. Starting with one voter and one election official, we increment them alternately. Fig. 12 shows how the state space of the model, time taken by the tool to synthesise the new specification, and the amount of memory required by the tool as the number of users increase. LTSA, written in Java, has a memory limitation of 1GB on 32-bit machines. After a combined total of 26 users, LTSA raises an out of memory error. The time required to generate the counter example is negligible in all cases. Both time and space required for the synthesis explodes quickly as the number of users approaches 26. A large part of the time inefficiency is due to the minimise operation. The largest model has a state space over 4×10^{11}, and is synthesised in about 5 mins using 240 MB of RAM on a 32-bit laptop.

Threats to Validity. There are both internal and external threats to validity in our evaluation. An internal threat is related to the use of LTL to specify the requirements, which could have limited expressivity especially in cases relating to complex data structures. We also relied on the operators available in LTSA (and associated process algebra FSP) to implement the algorithm. Other model checkers may have different operators or have built-in simplification that give better performance. Since the complexity of the revision is exponential, finding the right level of abstraction to analyse the behaviour of a system is paramount. For example, increasing the number of users adds to the complexity without necessarily uncovering different behaviours. With respect to external threats to validity, OASIS was evaluated in two cases for which we already knew about potential attacks. We plan to conduct more extensive evaluation to investigate whether the tool will identify false positive attacks which might have been handled by other means (e.g., firewalls or human agents) and how the revisions proposed are received by developers.

D. Discussion

Our initial evaluation demonstrated that OASIS can be used to revise the software specification to allow for weakened user obligations thus improving overall system security (mitigating more attacks). We made several simplifying assumptions in order to implement and empirically evaluate our approach. We discuss how some of these assumptions can be relaxed.

Minimality of revised specification. We have assumed that revisions of the software that involves fewer actions are relatively minimal to those that involve more actions. This may not be the case when the revision is translated into code. This issue needs to be investigated in future work.

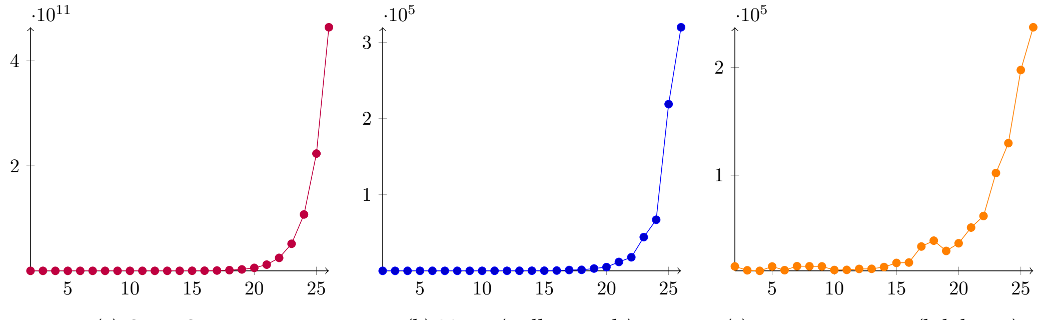
Generalisation. Being able to precisely determine the role of the users and to make the software resilient to deviation in their behaviour is an important issue. This also raises the question of how users interact when the machine itself is made up of multiple components that collaborate in order to satisfy security. We are applying the proposed approach to practical problems in a number of domains. In particular, many real-world examples will involve properties about data (such as collection of the vote results) and multiple processes and people. While this paper focuses on behavioural analysis, existing work on synthesis also considers data flow and associated control [36]. In addition, while the behaviour of individual components are usually small, if all the components in the environment are modelled the size of the composed model may grow quickly, increasing the complexity of the synthesis. Collaborative approaches to synthesis [37] provides a way to manage this complexity in domains such as the IoT.

Applicability. We intend to conduct a more comprehensive empirical study in order to evaluate relevance of the suggested revisions, and compare their acceptability by software developers, who may not always follow security practices [38]. In particular, OASIS assumes that the requirements can be expressed in LTL, which might not always be the case, especially for cases relating to complex data structures. The proposed revisions are based on changing the order of actions or removing some actions while other potential revisions might also involve frequency of actions or adding actions. These revisions might require techniques other than automated synthesis and we plan to investigate whether these revisions can be learnt. We also plan to evaluate the effort required to assess the identified attack scenarios and the willingness to address them by revising specifications.

VII. Related Work

The proposed approach OASIS is related to existing research in a number of areas, which as summarised below.

1) Requirements engineering for system security. Focusing on security properties such as non-interference, Rushby [19] argues that it is difficult to write security requirements because some security properties do not match with behavioural properties that can be expressed using formal methods. However, at the system level, certain security requirements such as preventing vote flipping can be described in terms of safety and liveness properties. Looking at the system from the point of view of an attacker is a common way of eliciting security requirements, and is the basis of threat modelling and attack tree approaches to security [11], [39]. Requirements of an attacker are called negative requirements, or anti-requirements [40]. Once identified, the software engineer has to design the system that prevents the anti-requirements from being satisfied. The idea has been extended by considering various patterns of anti-requirements, known as “abuse frames” [41]. In goal-oriented modelling, anti-requirements are called anti-goals, and the anti-goals can be refined in order to identify obstacles to security goals, and generate countermeasures [42]. In a similar vein, a systematic process to analyse security requirements in a social and organisational setting has been proposed [43]. The OASIS approach focuses on the behaviour of users interacting with the software, and identifies potential security vulnerabilities by weakening their obligations. To the best of our knowledge, existing work has not addressed this issue explicitly. Formal and semi-formal argumentation approaches have been used to reason about



(a) State Space (b) Time (milliseconds) (c) Memory Usage (kilobytes)

Fig. 12: Evaluation of tool scalability

system security [44]. It is easy to argue that if the users fail to discharge their obligations fully, then the software system cannot be expected to satisfy its security requirements. The OASIS approach shows instead that it is often possible to weaken user obligations, and as a result the system becomes more robust with respect to its security requirements.

2) Usable security. System security mechanisms are more effective when they are user-friendly [12]. Other studies of usable security have focused on the issues of password policies [45], [46], user behaviour when confronted by security warnings on web browsers [47], and so on. Shifting focus to developers and their mistakes when writing security-critical code, recent studies have examined the issue of the usability of cryptographic APIs [48]. From the usability point of view, the vote flipping problem discussed in this paper is known as the “missed sub-goal problem”, and the general solution is to focus on the simplicity of the path for users to achieve their goal. It means for example, whether the language used is appropriate, and whether the system states reflect the mental model of the user. Complementing these approaches, the OASIS approach shows that the software behaviour can be designed so that the system remains secure even when the user does not complete their tasks fully.

3) Formal verification of authentication protocols. Meadows [49] surveys approaches to formal verification of authentication protocols which include methods based on communicating state machines, modal logics, and algebraic models. Recognising that many authentication problems stem from user behaviour, and not necessarily from the protocols themselves, recent work has begun to examine the interactions between user behaviour and authentication protocols (the top layer in Fig. 2). Basin *et al.* [50], for example, focus on modelling and reasoning about human error in security protocols. First, they define human error as deviation from the role specification, which produces a partial order on these errors. They integrate the human error model within existing formalisation of security protocols, and verify the security properties of the security protocol under human errors. However, unlike OASIS, their approach focuses on verification rather than repair and operates at the protocol level.

4) Controller synthesis. There are many approaches to mediator synthesis [21]–[27], and they differ in their assumptions and the expressiveness of the goals involved. It is not always possible to synthesise a mediator that will maintain the requirements satisfied whatever the environment properties. D’Ippolito *et al.* [20] propose a multi-tier framework for graceful degradation by switching machine specifications, which are organised as a stack where higher layers making strong assumptions about the environment and providing stronger guarantees. However, the environments and the controllers must be in simulation. The OASIS approach relaxes this assumption through iterative abstractions and synthesis.

VIII. Summary and conclusions

We have proposed an approach to identify vulnerabilities due to strong assumptions about the user behaviour and to update the software specification to allow for weaker assumptions about the user behaviour while maintaining the satisfaction of the security requirements. The proposed approach begins by formalising the structure as well as the behaviour of a socio-technical system with critical security requirements, which makes user obligations explicit and enable us to identify attack scenarios. The approach then alternates between abstraction and synthesis to generate a revised specification that fixes the identified vulnerabilities and satisfies the requirements in the more realistic environment with weakened assumptions. We validated the approach by applying it to the e-voting and e-commerce examples and evaluating how it scales with a state space up to 4×10^{11}. Our approach goes beyond refinement for the revised specification and generates a specification that can change the sequencing of existing actions controlled by the machine. The results of the evaluation show that OASIS can identify security vulnerabilities and resolve them by controlling the software behaviour without unnecessarily changing, constraining or controlling the user behaviour.

We plan to carry out further application of the approach to potentially discover new attack scenarios, identify more efficient methods for managing the iteration between abstraction and synthesis, and explore notions of minimal change.

IX. Acknowledgments

This research is supported by EPSRC and SFI (grant 13/RC/2094). We thank Michael Jackson and Sebastian Uchitel for helpful discussions, and Gunter Mussbacher and the anonymous referees for constructive suggestions.

Appendix

In the following we explain how LTSA operators are used to implement Algorithm 1. Assuming L is $\{password\}$ Lines [4–6] are implemented using the hide (\), interface (@) and minimisation (minimal) operators:

```
minimal||M = EM\{password}.
minimal||N = EM@{password}.
||F = (M||N).
```

In order to ensure that M and N are composed correctly, action labels need to be prefixed appropriately. We first rebel the voter and the election official before creating the model for E_2 (Line 7).

```
minimal||Sys0 = ({v}::F || v:Voter)
  @{v.enter,v.password,v.select,v.vote,
  v.back,v.confirm,v.exit}.
minimal||Sys1 = ({eo}::F || eo:EO)
  @{eo.enter,eo.select,eo.vote,eo.back,
   eo.confirm,eo.exit}.
||E2 = (Booth||Sys0||Sys1||{v,eo}::F).
```

Since LTSA does not allow us to state the requirement for synthesis using LTL, we rephrase the security requirement R_s as a behavioural model (NoEOConfirm) together with a partial specification S before synthesising C.

```
S = (back−>back−>END).
property NoEOConfirm = (v.confirm −>
NoEOConfirm).
C = (S || Env || NoEOConfirm).
```

The rest of the algorithm is a wrapper to the LTSA tool. The complete specification are available at https://github.com/amelBennaceur/oasis.

In the e-voting example, the attack scenario is generated by checking the following LTL property.

```
assert NoEOSelectAfterVPassword
     = [](v.vote −> [](!eo.select))
```

Assuming the behaviour models of the machine, voting booth, voter and voting official specified in Fig. 3, 4, and 6 respectively, the following output is produced

```
Depth 9 −− States: 52 Transitions: 103
Memory used: 18610K
Trace to property violation in
   NoEOSelectAfterVPassword:
                v.enter
                v.password
                v.select
                v.vote
                v.exit
                eo.enter
                eo.back
                eo.back
                eo.select
Analysed in: 0ms
```

References

[1] Department for Digital, Culture, Media & Sport, "Cyber security breaches survey," 2017. [Online]. Available: https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2017

[2] C. Herley, "Unfalsifiability of security claims," *Proceedings of the National Academy of Sciences*, vol. 113, no. 23, pp. 6415–6420, 2016.

[3] P. G. Inglesant and M. A. Sasse, "The true cost of unusable password policies: password use in the wild," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 2010, pp. 383–392.

[4] M. Bada, A. M. Sasse, and J. R. C. Nurse, "Cyber security awareness campaigns: Why do they fail to change behaviour?" *CoRR*, vol. abs/1901.02672, 2019. [Online]. Available: http://arxiv.org/abs/1901.02672

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

[6] R. M. Keller, "Formal verification of parallel programs," *Communications of the ACM*, vol. 19, no. 7, pp. 371–384, 1976.

[7] [Online]. Available: https://www.doc.ic.ac.uk/ltsa/

[8] B. Friedman, "Clay county, ky, election officials sentenced to 156 years in election rigging case," http://www.bradblog.com/?p=8514, 2011, accessed: 2018-02-26.

[9] J. Nielsen, *Usability engineering.* Elsevier, 1994.

[10] S. Faily, *Usable and Secure Software Design: The State-of-the-Art.* Cham: Springer International Publishing, 2018, pp. 9–53.

[11] A. Shostack, *Threat modeling: Designing for security.* John Wiley & Sons, 2014.

[12] A. Whitten and J. D. Tygar, "Why johnny can't encrypt: A usability evaluation of pgp 5.0," in *Proceedings of the 8th Conference on USENIX Security Symposium.* Berkeley, CA: USENIX Association, 1999, pp. 14–14.

[13] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th USENIX Security Symposium*, P. D. McDaniel, Ed. USENIX Association, 2005. [Online]. Available: https://www.usenix.org/publications/proceedings/?f[0]=im_group_audience%3A156

[14] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication," *ACM Trans. Comput. Syst.*, vol. 8, no. 1, pp. 18–36, Feb. 1990.

[15] D. Boneh and V. Shoup, *A Graduate Course in Applied Cryptography*, Sep. 2017. [Online]. Available: https://crypto.stanford.edu/~dabo/cryptobook/

[16] M. Jackson and P. Zave, "Deriving specifications from requirements: An example," in *Proc. of the 17th Int. Conf. on Softw. Eng., ICSE*, 1995, pp. 15–24.

[17] A. Pnueli, "The temporal logic of programs," in *Proc. of the 18th Annual Symp. on Foundations of Computer Science*, 1977, pp. 46–57.

[18] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 1, pp. 1–30, 1997.

[19] J. Rushby, "Security requirements specifications: How and what?" in *Requirements Engineering for Information Security (SREIS), Indianapolis, IN*, 2001.

[20] N. D'Ippolito, V. A. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel, "Hope for the best, prepare for the worst: multi-tier control for adaptive systems," in *Proc. of the 36th International Conference on Software Engineering, ICSE*, 2014, pp. 688–699.

[21] D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors," *ACM Transactions on Programming Languages and System, TOPLAS*, vol. 19, no. 2, pp. 292–333, 1997.

[22] R. Vaculín, R. Neruda, and K. P. Sycara, "The process mediation framework for semantic web services," *International Journal of Agent-Oriented Software Engineering, IJAOSE*, vol. 3, no. 1, pp. 27–58, 2009.

[23] R. Mateescu, P. Poizat, and G. Salaün, "Adaptation of service protocols using process algebra and on-the-fly reduction techniques," *IEEE Transactions Software Engineering*, vol. 38, no. 4, pp. 755–777, 2012.

[24] L. Cavallaro, P. Sawyer, D. Sykes, N. Bencomo, and V. Issarny, "Satisfying requirements for pervasive service compositions," in *Proc. of the 7th Workshop on Models@run.time*, 2012, pp. 17–22.

[25] P. Inverardi and M. Tivoli, "Automatic synthesis of modular connectors via composition of protocol mediation patterns," in *Proc. of the 35th International Conference on Software Engineering, ICSE*, 2013, pp. 3–12.

[26] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel, "Synthesizing nonanomalous event-based controllers for liveness goals," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, 2013.

[27] A. Bennaceur and V. Issarny, "Automated synthesis of mediators to support component interoperability," *IEEE Trans. Software Eng.*, vol. 41, no. 3, pp. 221–240, 2015.

[28] C. A. R. Hoare, "Process algebra: A unifying approach," in *25 Years Communicating Sequential Processes*, 2004.

[29] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proc. of the 16th Annual Symp. on Principles of Programming Languages, POPL*, 1989, pp. 179–190.

[30] R. Ehlers, "Generalized rabin(1) synthesis with applications to robust system synthesis," in *Proc. of NASA Formal Methods - Third Int. Symp., NFM*, 2011, pp. 101–115.

[31] P. C. Kanellakis and S. A. Smolka, "CCS expressions, finite state processes, and three problems of equivalence," *Inf. Comput.*, vol. 86, no. 1, pp. 43–68, 1990.

[32] J. Fernandez, "An implementation of an efficient algorithm for bisimulation equivalence," *Sci. Comput. Program.*, vol. 13, no. 1, pp. 219–236, 1989.

[33] G. Holzmann, *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[34] D. Giannakopoulou, "Model checking for concurrent software architectures," Ph.D. dissertation, University of London, 1999.

[35] S. S. Lam and A. U. Shankar, "Protocol verification via projections," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 325–342, 1984.

[36] A. Bennaceur and B. Nuseibeh, "The many facets of mediation: A requirements-driven approach for trading-off mediation solutions," in *Managing trade-offs in adaptable software architectures*, I. Mistrík, N. Ali, J. Grundy, R. Kazman, and B. Schmerl, Eds. Elsevier, 2016.

[37] A. Bennaceur, T. T. Tun, A. K. Bandara, Y. Yu, and B. Nuseibeh, "Feature-driven Mediator Synthesis: Supporting Collaborative Security in the Internet of Things," *ACM Transactions on Cyber-Physical Systems*, 2017, to appear. [Online]. Available: http://oro.open.ac.uk/50803/

[38] T. Lopez, H. Sharp, T. T. Tun, A. K. Bandara, M. Levine, and B. Nuseibeh, ""hopefully we are mostly secure": views on secure code in professional practice," in *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE@ICSE 2019, Montréal, QC, Canada, 27 May 2019*, 2019, pp. 61–68. [Online]. Available: https://doi.org/10.1109/CHASE.2019.00023

[39] B. Schneier, "Attack trees," *Dr Dobb's Journal*, vol. 24, no. 12, December 1999.

[40] R. Crook, D. C. Ince, L. Lin, and B. Nuseibeh, "Security requirements engineering: When anti-requirements hit the fan," in *10th Anniversary IEEE Joint International Conference on Requirements Engineering (RE 2002), 9-13 September 2002, Essen, Germany*, 2002, pp. 203–205.

[41] L. Lin, B. Nuseibeh, D. C. Ince, and M. Jackson, "Using abuse frames to bound the scope of security problems," in *12th IEEE International Conference on Requirements Engineering (RE 2004), 6-10 September 2004, Kyoto, Japan*, 2004, pp. 354–355.

[42] A. van Lamsweerde, "Elaborating security requirements by construction of intentional anti-models," in *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, 2004, pp. 148–157.

[43] L. Liu, E. S. K. Yu, and J. Mylopoulos, "Security and privacy requirements analysis within a social setting," in *11th IEEE International Conference on Requirements Engineering (RE 2003), 8-12 September 2003, Monterey Bay, CA, USA*, 2003, pp. 151–161.

[44] C. B. Haley, R. C. Laney, J. D. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *IEEE Trans. Software Eng.*, vol. 34, no. 1, pp. 133–153, 2008.

[45] A. Adams and M. A. Sasse, "Users are not the enemy," *Commun. ACM*, vol. 42, no. 12, pp. 40–46, Dec. 1999.

[46] D. Florencio and C. Herley, "A large-scale study of web password habits," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 657–666.

[47] D. Akhawe and A. P. Felt, "Alice in warningland: A large-scale field study of browser security warning effectiveness." in *USENIX security symposium*, vol. 13, 2013.

[48] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic apis," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 154–171.

[49] C. Meadows, *Formal Verification of Cryptographic Protocols: A Survey*, ser. ASIACRYPT '94. London, UK: Springer-Verlag, 1995, pp. 135–150.

[50] D. A. Basin, S. Radomirovic, and L. Schmid, "Modeling human errors in security protocols," in *Proc. of the IEEE 29th Computer Security Foundations Symposium, CSF*, 2016, pp. 325–340.