Conformance Verification of Privacy Policies

Xiang Fu

Department of Computer Science
Hofstra University
Xiang.Fu@hofstra.edu

Abstract. Web applications are both the consumers and providers of information. To increase customer confidence, many websites choose to publish their privacy protection policies. However, policy conformance is often neglected. We propose a logic based framework for formally specifying and reasoning about the implementation of privacy protection by a web application. A first order extension of computational tree logic is used to specify a policy. A verification paradigm, built upon a static control/data flow analysis, is presented to verify if a policy is satisfied.

1 Introduction

The importance of protecting personal information privacy has been recognized for decades (e.g., see the the Health Insurance Portability and Accountability Act [23]). To increase customer confidence, many websites publish their privacy policies regarding the use and retention of information, using standards such as P3P [30] and EPAL [26]. For example, an online store web application can state that the credit card information she collects is used for the purpose of financial charge only and will be destroyed once the transaction is completed.

A publicly stated policy, however, may be violated by malperformed business practices and web application implementations, e.g., saving the credit card information to a "secrete database" instead of destroying it. We are interested in the following problem:

Given a privacy policy which specifies the purpose, retention, and recipient of information, can designers employ static analysis techniques to verify if a web application (including its code level implementation and system configuration) satisfies the policy?

The problem is essentially a verification problem (which we dubbed as "conformance verification"), because this is about checking if a system implementation respects a public specification (e.g., to observe a communication protocol [24, 5]).

We propose PV (Privacy Verification), a framework built upon the first order relational logic [18]. The conformance check is semi-automatic and it consists of the following stages: (1) **Modeling**: A web application is modeled as a reactive system that responds to requests from its customers and stakeholders. Each servlet is modeled as an atomic transition rule that updates the knowledge of entities about private information. Here, an entity can be used to describe any

"live being" in the model, e.g., a servlet, an employee, a database, an operating system call, etc. (2) **Static Analysis**: To ensure the precision of a model, a static analysis is used for extracting the information flow between entities of a web application, which is later used to construct the formal model of each web servlet. The analysis is conservative in that it may have false positives, but every possible information flow path in a servlet is reported. (3) **Verification**: a privacy policy is specified using a temporal logic, by adapting CTL [10] with first order relational logic components. The verification has to be limited to a finite model for ensuring decidability. The Alloy Analyzer [17] is used in the proof of the concept experiments. In the future, the Kodkod constraint solver [28] can be directly used for discharging symbolic constraints.

The contributions of this paper include the following: (1) PV provides a compact representation of knowledge ownership and its dynamic changes with system execution; (2) Instead of deriving PV from a top-down design, the static analysis framework extracts PV model from code level implementation. This would save tremendous time in modeling and can better accommodate with the evolution of software systems; and (3) the verification paradigm includes a processing algorithm that handles highly expressive CTL-FO, which is not originally available in Alloy (as a model finder).

The rest of the paper is organized as follows. §2 briefly overviews P3P, which motivates the formal model in §3. §4 presents the verification algorithm. §5 introduces the static analysis algorithm that extracts information flow. §6 discusses related work and §7 concludes.

2 Overview of Security Policies

There are several competing standards for privacy protection, e.g., EPAL [26] and P3P [30]. Although often under debate and criticism, P3P has gained wide acceptance. Each P3P security policy consists of a collection of security statements. A statement declares the purpose of the data collection activity, the data group to be collected, the intended retention, the recipient, and the consequence.

```
<STATEMENT>
 <CONSEQUENCE>
   We charge your credit card for your purchase order.
   Information is destroyed once transaction complete.
 </CONSEQUENCE>
 <PURPOSE><sales/></PURPOSE>
 <RECIPIENT><ours/></RECIPIENT>
 <RETENTION><stated-purpose/></RETENTION>
 <DATA-GROUP>
  <DATA ref="#user.payment.creditcard">
    <category> <purchase /> </category>
  </DATA>
 </DATA-GROUP>
</STATEMENT>
```

Fig. 1. Sample P3P Statement

Figure 1 shows one sample P3P statement for a web application that charges user credit card. As shown by the policy, the purpose of the data collection is for

`sales`. The information will be maintained at the website (as specified by `ours` in the `RECIPIENT` element), for a limited time until the transaction is completed (as indicated by `stated-purpose`).

P3P provides many predefined constants for each element of a statement. For instance, the following are several typical values for the `PURPOSE` element: (1) `current`: for the current one-time activity, (2) `admin`: for website administration, and (3) `telemarketing`: the information can be reused for promotion of a product later. For another example, the value of the `RECIPIENT` element can be, e.g., `ours` (the website owner), `delivery` (the delivery service), `same` (including other collaborators performing one-time use of the information), and `public`.

Clearly, a P3P policy is an access control specification that describes how information is distributed, stored, and destroyed. Many consider the policy enforcement as a requirement engineering problem [15]. We are interested in automated verification and auditing of policy enforcement, using a logic based framework. This requires a simplified formal model which avoids semantics problems in P3P.

3 PV Framework

The PV logic framework intends to model the information flow among entities of a web application, including software components as well as stakeholders. We assume an infinite model in this section, but later in verification, Alloy Analyzer works on a finite model only.

3.1 Data Model

Let $\mathcal{E}$ be an infinite but countable set of *entities* in a web application. An entity represents an *atomic* real entity of the world. It can be a person, a database, and an organization. Let $\mathcal{D}$ be an infinite and countable set of *data items*. Each data item $d \in \mathcal{D}$ is an atomic piece of information (e.g., the name of a person, a credit card number, etc.). The data model is flattened, i.e., we do not allow hierarchical data structure in the model like [7]. Data typing is defined using set containment as in relational logic [18]. A *data type* is a set of data items. A data type D_1 is a *subtype* of D_2 iff $D_1 \subseteq D_2$. If a data item $d \in D$, we say that d has the type D.

3.2 Web Application

A web application is modeled as a reactive software system, consisting of a finite collection of *servlets*. This corresponds to many existing web application platforms such as PHP, JavaEE, and ASP.Net. A *servlet* is a function which takes an HTTP request as input, and returns an HTTP response as output. It may have side effects, e.g., manipulating backend database, and sending emails. Very often the business organization (owner of the website) may have routinely performed procedures (e.g., clearing customer database monthly etc.). They are similar to servlets in that they have side effects. We generalize the notion to *actions* for capturing the semantics of both servlets and business procedures.

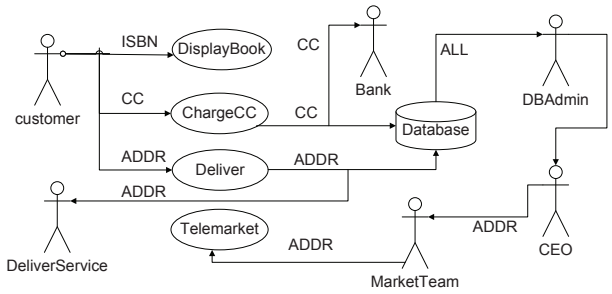

Fig. 2. Sample Bookstore Application

Definition 1. *A web application W is a tuple $(\mathcal{D}, \mathcal{E}_w, \mathcal{A}, \mathcal{P}, \mathcal{E}_e, P)$ where $\mathcal{D}$ is a a set of data items, $\mathcal{E}_w$ is a finite set of entities in W, $\mathcal{A} \subseteq \mathcal{E}_w$ is a finite set of actions (including servlets and business procedures), $\mathcal{P}$ is a finite set of purposes, and $\mathcal{E}_e$ is a set of entities in the environment. $P : \mathcal{A} \rightarrow 2^{\mathcal{P}}$ associates a set of purposes to each action.*

Example 1. Shown in Figure 2 is the architecture of a bookstore web application. $\mathcal{E}_w = \{$`DisplayBooks`, `ChargeCC`, `Deliver`, `Telemarket`, `DB`, `DBAdmin`, `MarketTeam`, `CEO`$\}$ is the set of entities within the application. $\mathcal{A}$ contains the first four elements (i.e., the servlets) of $\mathcal{E}_w$. $\mathcal{E}_e = \{$`Customer`, `Bank`, `DeliveryService`$\}$ contains three external entities that interact with the web application.

$\mathcal{P}$ has three elements: `purchase`, `delivery`, and `marketing`. Clearly, for the three servlets, their purposes can be defined as below. `P`(`ChargeCC`) = {`purchase`, `delivery`}, `P`(`Deliver`) = {`delivery`}, and `P`(`Telemarket`) = {`marketing`}.

$\mathcal{D}$ consists of data items that belong to three data types: `CC` (credit card number), `ADDR` (address), and `ISBN` (book id). In the rest of the paper, we use the above example as a motivating case study.

3.3 Action

We assume that each servlet (action) will eventually complete, i.e., it is never trapped in an infinite loop. In practice, this is guaranteed by the time-out action of web server. An action is atomic. Formally, an action is defined as a transition rule that manipulates predicates on the "knowledge" of entities.

An *action* is a first order relational logic formula [18], built on a predicate named `know`. When a predicate is primed, it represents the value of the predicate in the next system state of a web application. All free variables in the formula are regarded as input parameters.

Example 2. We list the specification of all servlets in Example 1, which can be generated by the static analysis algorithm in §5.

1. `DisplayBooks:` $\forall x \in \mathcal{E}_w \cup \mathcal{E}_e \; \forall d \in \mathcal{D} : \; \texttt{know}(x, d) = \texttt{know}'(x, d)$.
   The servlet does not have any side effects. Thus it does not change the valuation of predicate `know`.
2. `ChargeCC:` $cc \in \texttt{CC} \; \wedge \; \texttt{know}'(\texttt{DB}, cc) \; \wedge \; \texttt{know}'(\texttt{Bank}, cc) \; \wedge \; \forall x \in \mathcal{E}_w - \{\texttt{DB}, \texttt{Bank}\} \; \forall d \in \mathcal{D} : \; \texttt{know}(x, d) = \texttt{know}'(x, d) \wedge \forall d \in \mathcal{D} - \{cc\} : \texttt{know}(\texttt{Bank}, d) = \texttt{know}'(\texttt{Bank}, d) \; \wedge \; \texttt{know}(\texttt{DB}, d) = \texttt{know}'(\texttt{DB}, d)$
   Note that cc (the free variable) is the input parameter of the servlet. `ChargeCC` saves the information to local database and submits the information to `Bank`. Here `CC`, `DB`, `Bank` are all constants. The last two clauses (with $\forall$ quantifiers) keep the valuation of the predicate for all other entities and data. In the rest of the paper, we use $\texttt{same\_except}((\texttt{DB}, cc), (\texttt{Bank}, cc))$ to represent such an assignment that maintains predicate valuations in the new system state, except for pairs $\{(\texttt{DB}, cc), (\texttt{Bank}, cc)\}$.
3. `Deliver:` $cc \in \texttt{CC} \; \wedge \; ad \in \texttt{ADDR} \; \wedge \; \texttt{know}'(\texttt{DeliveryService}, ad) \; \wedge \; \texttt{know}'(\texttt{DeliveryService}, cc) \; \wedge \; \neg\texttt{know}'(\texttt{DB}, cc) \; \wedge \; \texttt{same\_except}(\{(\texttt{DB}, cc), (\texttt{DeliveryService}, \{cc, ad\})\})$.
   The `Delivery` servlet is similar to `ChargeCC`. The difference is that the record cc is now removed from `DB`, to achieve the `stated-purchase` property, i.e., the credit card information is destroyed after the transaction is completed.
4. `Telemarket:` it is the same as `DisplayBooks` and does not have any side effects (it simply takes a list of email addresses and sends out emails).

There are certain cases that the information flow between components may not be extracted by a static analysis. We use the notion of `Pipe` to specify such flow directly.

Definition 2. *Let $D \subseteq \mathcal{D}$ be a data type and e_1 and e_2 two entities in $\mathcal{E}_w \cup \mathcal{E}_e$. We say $\texttt{Pipe}(e_1, e_2, D)$ if information of type D flows from e_1 to e_2.*

Each $\texttt{Pipe}(e_1, e_2, D)$ can be translated into a transition rule

$$\forall d \in D : \texttt{know}(e_1, d) \Rightarrow \texttt{know}'(e_2, d)$$

Example 3. The information flow in Figure 2 can be represented using the conjunction of three pipes: (1) $\texttt{Pipe}(\texttt{DB}, \texttt{DBAdmin}, \mathcal{D})$, (2) $\texttt{Pipe}(\texttt{DBAdmin}, \texttt{CEO}, \mathcal{D})$, and (3) $\texttt{Pipe}(\texttt{CEO}, \texttt{MarketTeam}, \texttt{ADDR})$.

3.4 Modeling Security Policy

A privacy policy is about the access control of information: (1) if the information is used for intended use, (2) if the information is destroyed after the specified retention, and (3) if the information is only known by a restricted group of people. This is reflected by the `PURPOSE`, `RETENTION`, and `RECIPIENT` elements in a P3P policy.

Natural language specification, however, lacks formal semantics and can often cause confusion. Consider, for example, the term `stated-purpose` in P3P specification ("`Information is retained to meet the stated purpose. This`

`requires information to be discarded at the earliest time possible`" [30]). The "earliest time" can be interpreted in many ways, e.g., when the transaction is completed or when the website owner feels no needs of the data.

Temporal logic can be used to nicely capture privacy policies. We assume that readers are familiar with computational tree logic (CTL) [10]. In the following we define CTL-FO, an extension of CTL which allows free mixture of first order quantifiers. The definition is borrowed directly from [11, 13], with slight modification for relational logic.

Definition 3. *Let $\mathcal{D}$ be a data domain and $\mathcal{P}$ be a finite set of predicates. The CTL-FO formulas are defined as below:*

1. *Let $p \in \mathcal{P}$ be a predicate with arity n, and $\boldsymbol{x}$ be a vector of variables and constants ($|\boldsymbol{x}| = n$). Then $p(\boldsymbol{x})$ is a CTL-FO formula.*
2. *If f and g are CTL-FO formulas, then all of the following are CTL-FO formulas: $f \ \wedge \ g$, $\neg \ f$, $f \ \vee \ g$, $\mathbf{AX}\ f$, $\mathbf{EX}\ f$, $\mathbf{AF}\ f$, $\mathbf{EF}\ f$, $\mathbf{A}\ f\ \mathbf{U}\ g$, $\mathbf{E}\ f\ \mathbf{U}\ g$, and $\mathbf{A}\ f\ \mathbf{R}\ g$, and $\mathbf{E}\ f\ \mathbf{R}\ g$. Here $\mathbf{A}$ (universal path quantifier), $\mathbf{E}$ (existential path quantifier), $\mathbf{X}$ (next state), $\mathbf{F}$ (eventually), $\mathbf{G}$ (globally), $\mathbf{U}$ (until), and $\mathbf{R}$ (release) are standard CTL temporal operators.*
3. *If f is a CTL-FO formula and $D \subseteq \mathcal{D}$ is a data type, then $\forall x \in D : f$ and $\exists x \in D : f$ are CTL-FO formulas.*

Definition 4. *A CTL-FO formula φ is said to be* well formed *if φ has no free variables, and every quantified variable v appears in some predicate in φ.*

The semantics of CTL-FO formula can be defined by directly extending the CTL semantics in [10] with the addition of the following two semantics rules. Given a formula φ and a free variable v in φ, $\varphi_{x \leftarrow d}$ is the result of replacing every x with constant d. Then given a Kripke structure M and a state s:

1. $M, s \models \forall x \in D : \varphi \ \Leftrightarrow$ for each value d in D: $M, s \models \varphi_{x \leftarrow d}$
2. $M, s \models \exists x \in D : \varphi \ \Leftrightarrow$ there exists a value d in D s.t. $M, s \models \varphi_{x \leftarrow d}$

Note that in the above definition, the $\forall x \in D : \varphi$ and $\exists x \in D : \varphi$ are required to be well-formed. In another word, $\varphi_{x \leftarrow d}$ has no free variables. In the following we introduce one security policy specified using CTL-FO for Example 1.

Example 4. Policy 1: any credit number collected by the `ChargeCC` servlet is eventually destroyed by the web application, i.e., no entities in the bookstore web application knows about the credit card number eventually. The property can be expressed as below:

$$\forall d \in \mathtt{CC} : \mathbf{AG}(\mathtt{know}(\mathtt{DB}, d) \ \Rightarrow \ \mathbf{AF}(\forall x \in \mathcal{E}_w : \neg \mathtt{know}(x, d)))$$

3.5 Conformance Verification Problem

Given a web application $W = (\mathcal{D}, \mathcal{E}_w, \mathcal{A}, \mathcal{P}, \mathcal{E}_e, P)$, it is straightforward to define a Kripke structure on W, written as $M(W)$. The basic idea is that each state of

$M(W)$ is a distinct valuation of predicate `know` on each pair of entity (in $\mathcal{E}_w \cup \mathcal{E}_e$) and data item (in $\mathcal{D}$). The initial state s_0 of W needs to be manually defined by the designer. Transitions between states can be derived by the action rules of $\mathcal{A}$. Then the conformance verification problem is defined as the following.

Definition 5. *Let W be a web application, and (M, s_0) the derived Kripke structure and the initial state. W conforms to a CTL-FO formula φ iff $M, s_0 \models \varphi$.*

4 Verification

4.1 Overview

This section introduces a symbolic verification paradigm which takes a web application specification and a CTL-FO formula as input. It either outputs "yes", or generates a firing sequence of servlets (or other actions) that leads to the violation of the property. We rely on the Alloy Analyzer [17] for model checking if a PV model satisfies a privacy policy specified in CTL-FO logic. Using SAT-based model finder Kodkod [28], Alloy performs scope-restricted model finding. The Alloy specification supports first order relational logic [18], which is very convenient for specifying the transition system of a PV model.

The verification paradigm consists of the following steps:

1. *Translation from PV Transition System to Alloy:* It consists of two parts: (a) specification of all PV data entities using the Alloy type system, and (b) translation of each action into a first order relational logic formula that contains both current/next state predicates. Notice that Alloy is originally designed for static model analysis, the state of a PV transition system has to be explicitly modeled to simulate a Kripke structure. Subsection 4.2 presents the technical details about this part of translation.
2. *Translation from a CTL-FO formula to Alloy predicates and assertions:* Alloy itself does not support temporal logic. This translation is about simulating the fixpoint computation of temporal operators. Subsection 4.3 presents the details.
3. *Verification using Alloy:* Given a property (expressed as Alloy assertions), Alloy is able to find a model that violates the assertion using a finite model/scope search. The error trace can be easily identified from the visual representation provided by Alloy. In the visual model, an error trace comprises of a sequence of PV states. Note that these states and their transition relation are already explicitly encoded in the model.

4.2 Translating PV Transition System to Alloy

The translation algorithm is straightforward. Currently, for the case study example, the translation is accomplished using manual simulation of the algorithm. In our future work, the translation will be automated.

Taking Example 1 as an example, its Alloy specification is given in Figure 3. The specification contains three parts: (1) the general data schema that defines

the world of entities and actions; (2) the specific data setting related to the web application, e.g., its actions and stakeholders, (3) the formal definition of actions, where each action (servlet) is modeled as a parametrized transition rule.

```
module bookstore

//1. World Schema
abstract sig Object {}
abstract sig WA, Env, Data extends Object {}
abstract sig Actions,  Entities extends WA{}
abstract sig actionStatus{}
one sig RUN, SLEEP, READY extends actionStatus{}
abstract sig Purpose {}
sig State{
  know: (WA +Env) -> Data,
  prev: one State,
  actstate: Actions -> actionStatus
}{
  all x: Actions | some status: actionStatus |
      x -> status in actstate
}
sig initState extends State {}
fact generalInitState{
  all x: initState |
    (x.prev = x and
       all y: Actions | x.actstate[y] = SLEEP
    ) and
    (some y: State -initState | x in y.^prev)
}
fact factAllStates{
  all x: State - initState | some y : initState |
      y in x.^prev
}
fact TransitionRelation{
  all x: State   | all y: State - initState |
      ( (x in y.prev => Transition[x,y]) and
      (Transition[x,y] => x in y.prev) )
}

//2. Web Application Specific Setting (Bookstore)
sig NAME, CC, ADDR, ID extends Data{}
one sig DisplayBooks, ChargeCC,
    Deliver, Telemarket extends Actions {}
one sig DB, DBAdmin, MarketTeam,
    CEO extends Entities {}
one sig Bank, DeliverService, User extends Env {}
...

//3. Actions (Servlets and Pipes)
...
pred pChargeCC [s,s': State, d: CC]{
   ChargeCC->READY in s.actstate and
   (
    s'.know = s.know + {DB->d} +{Bank->d} &&
    s'.prev = s &&
    s'.actstate = s.actstate - {ChargeCC->READY}
      + {ChargeCC->SLEEP} - {Deliver->SLEEP}
      + {Deliver->READY}
   )
}

pred pipe[s,s': State, e1, e2: Object, D:  Data]{
  (some d: D | e1->d in s.know
     && !(e2->d in s.know))
  &&(
     s.know in s'.know &&
     (all x: (WA+Env) | all y:Data |
          (x->y in s'.know-s.know) <=>
          (x=e2 && e1->y in s.know
           && !(e2->y in s.know)) )
     && s'.prev=s
     && s'.actstate = s.actstate
  )
}

pred customer[s,s':State]{
  (DisplayBooks->SLEEP in s.actstate and
          s'.actstate = s.actstate
          - {DisplayBooks->SLEEP}
          + {DisplayBooks->READY}
    &&  s'.prev=s  && s'.know=s.know) or
   ...
}

pred Transition[s,s':State]{
  //servlets
  pDisplayBooks[s,s'] or
  pTelemarket[s,s'] or
  (some d:CC | pChargeCC[s,s',d]) or
  (some d:ADDR | some d2:CC |
       pDelivery[s,s',d,d2]) or
  //pipes
  pipe[s,s',DB,DBAdmin,Data] or
  pipe[s,s',DBAdmin,CEO,Data] or
  pipe[s,s',CEO,MarketTeam,ADDR] or
  //other actions
  customer[s,s']

}


//4. CTL-FO to predicates and assertion
pred ef[s:State,d:Data]{
    some s': State | (CEO ->d in s'.know)
         && s in s'.*prev
}

pred fa[s:State]{
  all d: Data | (DB->d in s.know) => ef[s,d]
}

assert AGProperty{
  all s: State | fa[s]
}
```

Fig. 3. Sample ALLOY Specification

The World Schema: The first section of the Alloy specification defines the general data schema for all PV model specifications in Alloy. Here `Object` is a generic type, which has three subtypes: `WA` (all entities within the web application), `Env` (all entities of the environment), and `Data` (all data items). The `WA` entities consist of `Actions` (like servlets) and `Entities` (like databases). To simulate each action as a transition, we define three constants to denote status of an action: `RUN`, `READY`, and `SLEEP` (the `RUN` status is actually not needed as each transition rule is atomic). To build a Kripke structure of the transition system, we declare `State` which includes `Knows` (tracking the knowledge of each entity on data items), `prev` (the previous state), and the status of each action. The restriction "`all x: Actions | some status: actionStatus | x -> status in actstate`" requires that all action has a status. We also declare that there is one or more initial states (as described by `fact generalInitState`): the status of all actions are set to `SLEEP`. More details of the initial states are defined in the section on web application specific settings. Note that Alloy specification is declarative, thus the order of Alloy statements does not affect the semantics. Finally, the fact `factAllStates` requires that all instances of `State` enumerated by Alloy should be contained in the transitive closure of `prev` link to an initial state (note `^` is the non-reflexive transitive closure operator).

Web Application Specific Settings: The settings related to the Bookstore application are specified, e.g., the subtypes `NAME`, `CC`, and `ADDR`, the servlets, and external entities.

Actions (Servlets, Pipes): Each action is modeled as a predicate in Alloy. A typical example is `pChargeCC` which represents the transition rule for the `ChargeCC` servlet. The predicate takes three parameters: `s` and `s'` are the current and next states. `d` is a credit card number. Clearly, the predicate specifies that in the next state, the `DB` is able to know `d`, and it updates the `prev` link and the action status correspondingly. The `pipe` predicate models a template of piping information. Given entities e_1 and e_2, if e_1 knows d, then in the next state, e_2 also knows d. Notice that there is a `customer` action, which non-deterministically invokes `DisplayBooks` and `ChargeCC`.

The `Transition` predicate defines the transition relation, which is composed of the predicates that model the servlets, pipes, and other actions/roles in the system. Clearly, with `Transition` and `factAllStates`, a Kripke structure is formed using the `prev` field of each state. Alloy will analyze the states reachable from the initial state only.

4.3 Translating CTL-FO Formula

A CTL-FO Formula can be translated into one Alloy assertion and a collection of Alloy predicates. The translation runs from top to bottom, following the syntax tree. The top level CTL-FO formula is formulated as an assertion. Then each component is modeled as an Alloy predicate (with one input parameter on

PV state, and the necessary parameters for quantified variables). The fixpoint computation of CTL formula can be modeled using first order logic plus the `prev` link (which models the transition relation between states).

Take the following CTL-FO formula as one example.

$$\mathbf{AG}(\forall d \in \texttt{Data} : \texttt{know}(\texttt{DB}, d) \;\Rightarrow\; \mathbf{EF}(\texttt{know}(\texttt{CEO}, d)))$$

The top level **AG** property is first translated into an Alloy assertion (as shown in `AGProperty`). The **EF** formula is defined as a predicate `ef` which has two parameters: `s` and `d`. Here `d` is a variable which is restricted by the universal quantifier, and `s` is a PV state. The predicate `ef[s,d]` is true iff at `s` and for data item `d` eventually there is a path leads to a state that `CEO` knows `d`. This is defined using the formula inside `ef`, which leverages the Kleene closure of the `prev` link (see "`s in s'.*prev`").

4.4 Initial Experimental Results

We performed an initial experiment with the Alloy model. The verification cost explodes quickly with the number of states. The system runs out of memory when it exceeds 50 PV states, on a PC with 4GM RAM.

5 Static Information Flow Analysis

This section proposes a semi-automatic static analysis algorithm that produces the PV model. The algorithm has not been implemented, but the idea is straightforward. The algorithm consists of four stages: (1) a static code analysis that extracts the external entities, e.g., file system and databases that are accessed by servlets, (2) a manual definition stage, where the designers supply the rest of the roles and stakeholders, e.g., the `MarketTeam` and `CEO` in Example 1, (3) modeling of all system calls, e.g., to define the data sink and operations performed by system calls such as JDBC `executeUpdate`, and (4) a fully-automatic analysis of servlet bytecode, which extracts information flow and builds the transition system. We omit the details of steps (1) to (3), and concentrate on step (4).

5.1 Path Transducer

The fully automatic static code analysis assumes that each web servlet can be represented by a set of *path transducers*. A *path transducer* of a servlet is essentially one possible execution path of the servlet from its entry to exit (by unwrapping loops, branches, and tracing into function calls). The analysis can be inter-procedural in the sense that it traces into every function defined by the web application, but not into the function body of any system functions provided by the environment (e.g., OS system calls).

Definition 6. *A* path transducer T *is a tuple* $(\boldsymbol{I}, \boldsymbol{M}, \boldsymbol{S})$ *where* $\boldsymbol{I}$ *is a finite set of input parameters,* $\boldsymbol{M}$ *is a finite set of variables, and* $\boldsymbol{S}$ *is a sequence of statements. Each* statement *has one of the following two forms (letting* $v \in \boldsymbol{M}$*,* $V \subseteq \boldsymbol{M}$*, and* C *be a sequence of constants):*

1. (Assignment) $v := E(V, C)$.
2. (System Call) $v := f(V, C)$.

Here $\boldsymbol{M}$ includes all static, stack, and heap variables that could occur during the execution, as unwrapping is bounded. E is an arithmetic or logical expression on V and C. f has to be a *system call* that is provided by the external environment of the web application.

```
1 protected void processRequest(
2  HttpServletRequest request ...){
3  PrintWriter out = response.getWriter();
4  String sUname = request.getParam("sUname");
5  String sPwd = request.getParameter("sPwd");
6  Connection conn = DM.getConnection("...");
7  Statement stmt = conn.createStatement();
8  String strCmd= "INSERT ..."
9     + massage(sUname) + ... + massage(sPwd);
10 int n = stmt.executeUpdate(strCmd);
11 if(n>0) out.println("Welcome"+sUname);
12 }
13 protected String massage(String str){
14   return str.replaceAll("'", "''");
15 }
```

Fig. 4. Add Member Servlet

```
1  out = f1();
2  sUname = input_sUname;
3  sPwd = input_sPwd;
4  //new entity DBConn_Addr
5  stmt = f2();
6  //now call massage(sUname)
7  str = sUname;
8  s1 = f3(str,"'", "''");
9  //now call massage(sPwd)
10 str = sPwd;
11 s2 = f3(str,"'", "''");
12 //now call executeUpdate
13 strCmd = f4("INSERT..."+s1+...+s2);
14 n = f5(stmt, strCmd)
```

Fig. 5. Sample Path Transducer

Example 5. Figure 4 presents a simple Java servlet that adds a user to a web email system. It calls a self-defined `massage()` function to sanitize user input. Then it submits an `INSERT` query to the database. The statement sequence of a sample path transducer is given in Figure 5. Here system calls are replaced by a shorter name for simplicity (e.g., `response.getWriter(...)` is replaced by `f1(...)`). All branch statements and calls of self-defined functions (i.e., `massage`) are removed by unwrapping. For example, `f3` (`String.Replace`) is called twice because the execution enters the `massage` function twice. Temporary variables, e.g., `s1` and `s2`, are created to handle the temporary function call results. But only a finite number of them are needed because the unwrapping depth is bounded. By defining the data sinks of system calls and applying string analysis such as [31], it is possible to extract flow information from system calls,e.g., from `f5` (the `executeUpdate`) function.

Symbolic execution [22] can be used to extract path transducers from the bytecode of servlets. A typical approach is to instrument the virtual machine and skip the real decision of a branch statement so that both branches can be covered (see e.g., [3]). An alternative approach is to instrument the bytecode of

the web application being inspected, to change its control flow, using tools such as Javassist [9]. A servlet may have an infinite number of path transducers.

```
1  Procedure CalcInfoFlow(𝒯 = (M, I, S), ρ : I → 2^𝒟)
2  //𝒯 is a path transducer, ρ is a mapping from input variables to the data items.
3     know := ρ
4     foreach s in S do:
5       case M_i := E(V, C):
6         for each v ∈ V: for each x s.t. (v, x) ∈ know: know.add(M_i,x)
7       case M_i := f(V, C):
8         for each v ∈ V: for each x s.t. (v, x) ∈ know: know.add(M_i,x)
9         Let e be the data sink of f
10        for each x s.t. (v, x) ∈ know:
11          know.add(e, x)
12    return {(x, d) | (x, d) ∈ know ∧ x ∉ M ∪ I}
```

Fig. 6. Static Analysis Algorithm

5.2 Static Analysis for Constructing Transition System

Figure 6 displays the static analysis algorithm. It takes two inputs: a path transducer $\mathcal{T}$ and a mapping ρ that associates each input variable with the corresponding private information. `CalcInfoFlow` returns a collection of tuples that represent the new facts about the knowledge of information by entities. `know` models the knowledge of all entities (including variables) on private information. We populate its contents from the initial knowledge of ρ. Whenever an assignment is reached, the knowledge of all variables on the right will also be the knowledge of the left hand side. When a statement is an invocation of system call, the information is propagated to the proper data sink. Finally, the collection of `know` tuples is returned. Then we can easily generate the transition rule that models the servlet. The following lemma implies that we do not need an infinite collection of path transducers to compute the complete result.

Lemma 1. *For any servlet a_i (letting ρ be the mapping in Figure 6), there exists a* finite *set of path transducers for a_i (letting it be Ψ) s.t. for any path transducer set for a_i (letting it be Φ) the following is true:*

$$\bigcup_{\tau \in \Phi} \texttt{CalcInfoFlow}(\tau, \rho) \subseteq \bigcup_{\tau \in \Psi} \texttt{CalcInfoFlow}(\tau, \rho)$$

6 Related Work

Using formal methods to model privacy has long been an area of interest (see [29] for a comprehensive review). One typical application is the study of the semantics of security policies. While P3P [30] is able to express a security policy in a machine understandable format (i.e., XML), its lack of formal semantics is often under debate as well as criticism. Barth and Mitchell pointed many pitfalls

of P3P and APPEL in [7], e.g., the early termination causing non-robustness, the lack of distinction between provider and consumer perspectives, and the missing of fine-grained access control on subset of data groups. Similarly, Yu *et al.* showed that a P3P privacy policy can have multiple statements conflicting with each other [32] (e.g., imposing multiple retention restrictions over one data item). There are several proposals to fix the problems of P3P, e.g., the EPAL standard [26], and the privacy policy based on semantic language DAML-S [19]. This paper uses a first order extension of computational tree logic (CTL) for modeling privacy policies. The benefit of using temporal logic is the simplicity of model and the very expressive temporal operators for expressing the notions of information control that is related to time. The idea of using temporal logic for specifying privacy policies is not new. In [6], Barth *et al.* presented an extension of Linear time Logic (LTL) for modeling a variety of privacy policy languages. Similar efforts include the `REVERSE` working group on trust and policy definition [25], led by M. Baldoni. Compared with [6], our contribution is the modeling of a web application as a transition system and the verification scheme that addresses the model checking of first order temporal logic, which is not discussed in [6].

Deployment of privacy policies (e.g., [1]) is not the concern of this paper. We are more interested in the *enforcement* (or conformance check) of privacy policies. Many works enforce privacy policies using a *top-down* fashion, e.g., role engineering in the software architectural design stage (assigning permission rights of storing and distributing information to stakeholders) [15], enforcing P3P policy in a web application by leveraging existing enterprise IT systems [4], asking designers to follow specific design patterns (IBM Declarative Data Privacy Monitoring) [16], and enforcing data access control using JIF (a variant of Java) [14]. This work adopts a *bottom-up* approach: given an existing web application, we perform static analysis on its bytecode, extract a formal model on information flow, and verify if the model satisfies a privacy policy. The conservative code analysis helps to increase confidence in privacy protection.

This work follows the general methodology of symbolic model checking and its applications to web services [8, 12]. However, we face the challenge of handling first order logic, which in general is an undecidable problem. Our approach is to bound the scope of the model, and rely on Alloy Analyzer [17] to perform a bounded model checking. Alloy has been widely applied to many interesting problems, e.g., multicast key management [27], correcting naming architecture [21], and solving relational database constraints [20]. Most of its applications are applied to static models, while we attempted the modeling of a dynamic transition system (and computing fixpoint of first order temporal logic formula) using Alloy. The experiment shows that the verification cost explodes quickly with the number of states. More efficient verification can be performed, e.g., by invoking the Kodkod model finder [28] directly. An alternative is to prove privacy preservation by studying refinement relation between transition systems [2].

7 Conclusion

This paper has presented a logic based framework for reasoning about the privacy protection provided by a web application. A first order extension of the computation tree logic is used to specify a privacy policy. Then a formal transition model is constructed by performing a semi-automatic code level analysis of the web application. The verification relies on Alloy Analyzer and is performed on a finite model, expressed in first order relational logic. Our future directions include implementing the PV framework, applying it to non-trivial web applications, and exploring more efficient constraint solving techniques.

References

1. Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Implementing p3p using database technology. In *Proceedings of 19'th International Conference on Data Engineering*, pages 595–606, 2003.
2. Rajeev Alur and Steve Zdancewic. Preserving secrecy under refinement. In *Proc. of the 33rd Internat. Colloq. on Automata, Languages and Programming (ICALP)*, pages 107–118, 2006.
3. Saswat Anand, Corina S. Pàsàreanu, and Willem Visser. JPF-SE: A symbolic execution extension to java pathfinder. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007)*, pages 134–138, Braga, Portugal, March 2007.
4. Paul Ashley. Enforcement of a p3p privacy policy. In *Proceedings of the 2nd Australian Information Security Management Conference*, pages 11–26, 2004.
5. M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In *Proc. of 2005 International Workshop on Web Services and Formal Methods (WS-FM)*, pages 257–271, 2005.
6. Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. Privacy and contextual integrity: Framework and applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 184–198, 2006.
7. Adam Barth and John C. Mitchell. Enterprise privacy promises and enforcement. In *Proceedings of the 2005 workshop on Issues in the theory of security*, pages 58–66, 2005.
8. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.
9. Shigeru Chiba. Getting started with javassit. *Available at* `http://www.csg.is.titech.ac.jp/~chiba/javassist/html/index.html`.
10. E.M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
11. Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web services. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 71–82, 2004.
12. X. Fu, T. Bultan, and J. Su. Model checking XML manipulating software. In *Proceedings of the 2004 Int. Symp. on Software Testing and Analysis (ISSTA)*, pages 252–262, 2004.

13. S. Hallé, . Villemaire, O. Cherkaoui, J. Tremblay, and B. Ghandour. Extending model checking to data-aware temporal properties of web services. In *Proceedings of 2007 Web Services and Formal Methods, 4th International Workshop*, pages 31–45, 2007.
14. Katia Hayati and Martin Abadi. Language-based enforcement of privacy policies. In *Proceedings of Privacy Enhancing Technologies Workshop*, pages 302–313, 2005.
15. Q. He and A. Anton. A framework for modeling privacy requirements in role engineering. In *Proceedings of the 9th International Workshop on Requirements Engineering: Foundation for Software Quality*, pages 1–15, 2003.
16. IBM. Declarative Data Privacy Monitoring. `http://www.redbooks.ibm.com/redbooks/pdfs/sg246999.pdf`.
17. D. Jackson. Alloy 3 Reference Manual. `http://alloy.mit.edu/reference-manual.pdf`.
18. Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering (FSE)*, pages 130–139, 2000.
19. Lalana Kagal, Massimo Paolucci, Naveen Srinivasan, Grit Denker, Tim Finin, and Katia Sycara. Authorization and privacy for semantic web services. *IEEE Intelligent Systems*, 19(4):50–56, 2004.
20. S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid. Query-aware test generation using a relational constraint solver. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 238–247, 2008.
21. S. Khurshid and D. Jackson. Correcting a naming architecture using lightweight constraint analysis. Technical report, MIT Lab for Computer Science, December 1998.
22. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
23. House of Representatives. Health insurance portability and accountability act of 1996. `http://www.gpo.gov/fdsys/pkg/CRPT-104hrpt736/pdf/CRPT-104hrpt736.pdf`.
24. Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In *Proc. 14th International Conference on Computer Aided Verification (CAV)*, pages 166–179, 2002.
25. REVERSE - policies & trust. `http://cs.na.infn.it/rewerse/pubs.html`.
26. W3C Member Submission. Enterprise privacy authorization language (epal 1.2). `http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/`.
27. Mana Taghdiri and Daniel Jackson. A lightweight formal analysis of a multicast key management scheme. In *In Proceedings of Formal Techniques of Networked and Distributed Systems*, pages 240–256. Springer, 2003.
28. Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, 2007.
29. Michael Carl Tschantz and Jeannette M. Wing. Formal methods for privacy. In *Proceedings of the 2nd World Congress on Formal Methods*, pages 1–15, 2009.
30. W3C. Platform for privacy preferences (p3p). `http://www.w3.org/P3P/`.
31. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 32–41, 2007.
32. Ting Yu, Ninghui Li, and Annie I. Antón. A formal semantics for p3p. In *Proceedings of the 2004 workshop on Secure web service*, pages 1–8, 2004.