Verifiable Assume-Guarantee Privacy Specifications for Actor Component Architectures

Claiborne Johnson
University of Texas at San Antonio
imm589@my.utsa.edu

Thomas MacGahan
Accenture Federal Services
thomas.macgahan@gmail.com

John Heaps
University of Texas at San Antonio
john.heaps@utsa.edu

Kevin Baldor
University of Texas at San Antonio
kevin.baldor@utsa.edu

Jeffery von Ronne*
University of Texas at San Antonio
vonronne@acm.org

Jianwei Niu
University of Texas at San Antonio
jianwei.niu@utsa.edu

ABSTRACT

Many organizations process personal information in the course of normal operations. Improper disclosure of this information can be damaging, so organizations must obey privacy laws and regulations that impose restrictions on its release or risk penalties. Since electronic management of personal information must be held in strict compliance with the law, software systems designed for such purposes must have some guarantee of compliance. To support this, we develop a general methodology for designing and implementing verifiable information systems. This paper develops the design of the History Aware Programming Language into a framework for creating systems that can be mechanically checked against privacy specifications. We apply this framework to create and verify a prototypical Electronic Medical Record System (EMRS) expressed as a set of actor components and first-order linear temporal logic specifications in assume-guarantee form. We then show that the implementation of the EMRS provably enforces a formalized Health Insurance Portability and Accountability Act (HIPAA) policy using a combination of model checking and static analysis techniques.

CCS CONCEPTS

• **Social and professional topics** → **Privacy policies**; • **Theory of computation** → *Modal and temporal logics*; • **Applied computing** → *Health care information systems*;

KEYWORDS

privacy policy; first-order linear temporal logic; safety and liveness properties; assume-guarantee specifications; static analysis

ACM Reference format:
Claiborne Johnson, Thomas MacGahan, John Heaps, Kevin Baldor, Jeffery von Ronne, and Jianwei Niu. 2017. Verifiable Assume-Guarantee Privacy Specifications for Actor Component Architectures. In *Proceedings of SACMAT'17, June 21-23, 2017, Indianapolis, IN, USA, ,* 12 pages.
DOI: http://dx.doi.org/10.1145/3078861.3078873

*Now at Google, Inc.

SACMAT'17, June 21-23, 2017, Indianapolis, IN, USA
 ACM ISBN 978-1-4503-4702-0/17/06…$15.00.
DOI: http://dx.doi.org/10.1145/3078861.3078873

1 INTRODUCTION

Our society is becoming increasingly dependent on computer information systems for the proper management of private data. Medical records, financial data, and personal information collected from Internet users are just a few examples. Organizations are required to keep and share such information in a manner that conforms to specific privacy policies, which are mandated by custom, sound business practice, good citizenship, contract, and often by law. Examples of privacy policies that carry the force of law include those resulting from the Health Insurance Portability and Accountability Act (HIPAA) [16], the Gramm-Leach-Bliley Act (GLBA) [2], and the Children's Online Privacy Protection Act (COPPA) [1]. In addition to legal regulations, organizations typically have their own business rules that add further privacy requirements.

Privacy policies differ from access control policies in many ways and privacy policy design has to consider them. First, privacy policies govern access and transmission rights concerning data *about* some individual end users (the *subject*). The subject typically has certain rights to control how data about her is collected, stored, used, and shared. However, this control is not absolute, so the subject cannot be considered to be the "owner" of the data in the sense the term is used in access control literature. Second, privacy policies need to indicate the purposes of collection and sharing of data whereas access control policies largely ignore the purposes. For instance, if personal information is shared or used for some class of purposes, it may become obligatory to notify the subjects that this has occurred. Third, while for the most part, access control policies specify safety requirements, privacy policies can also express liveness requirements. Intuitively, *safety* properties say that some bad thing never happens and *liveness* properties say that some good thing eventually happens [4, 20, 28]. For instance, "in the future the service provider must send the subject a notification about certain usage of data" is a liveness property.

Several frameworks have been proposed for specifying and analyzing privacy policies, including the Enterprise Privacy Authorization Language (EPAL) [5], Contextual Integrity (CI) [6], Privacy APIs [25], Ponder [11], and work by Breaux and Anton [8]. To date, most of the work in this area has concentrated on frameworks for expressing privacy policies [5, 6, 11], answering queries requesting rulings as to whether transmitting a certain piece of information is permitted [5, 11], methodologies for converting regulations expressed in legal language into formal system requirements [8], or analyzing privacy policies to determine whether they satisfy various properties [25]. In contrast, we have found a lack of enforcement

frameworks comprising techniques and tools that support the development and verification of information systems that adhere to privacy policies.

There has been a great deal of work in enforcement of access control policies, particularly in the area of runtime monitors [14, 21, 27]. Runtime monitoring addresses a problem that resembles ours in some respects. The classic approach [27] enforces safety properties, but is unworkable in reactive systems; when a software system attempts to perform an illegal action, execution is terminated. More recent work gets around this problem by introducing edit automata [7, 22], which enable execution to proceed by either suppressing the illegal action, or performing other actions prior to the requested one so as to make the desired action permissible. However, all monitoring-based approaches share a common characteristic: they separate security concerns from basic functionality. This is very suitable for many applications, such as adding security features to legacy code. However, separating policy enforcement from basic functionality is counter productive in privacy policy enforcement for information systems. The user of an information system plays a central role in the system's operation. When a user attempts to transmit information that would cause a privacy violation, providing an explanation is a central part of the system's functionality. For instance, in the context of electronic medical records, an administrative assistant may request the system to transmit portions of a patient's record for which prior consent by the patient is required. If consent has not been obtained, the administrative assistant needs this to be explained, enabling him to request consent from the patient. Such an explanation is best managed as a part of the system's basic functionality, rather than as a separate wrapper. Our prior work on the History Aware Programming Language (HAPL) [29] has outlined a language design with features for providing privacy management without need for a separate component, such as a monitor, to intervene by modifying the system's behavior during runtime.

In this paper, we aim to develop our HAPL design into a full framework with a working prototype of an Electronic Medical Record System (EMRS) that can be statically checked against formal privacy policy specifications for HIPAA. In order to bridge the gap between policy specifications and our implementation, we develop formal system specifications in assume-guarantee form [18]. Static analysis techniques are applied to verify that the implementation matches this specification, while a model checker is used on small specification sets reflecting the formal specifications to ensure that the system's formal specification enforces HIPAA.

Organization. Section 2 introduces background information used in this paper. We overview our framework in Section 3. Section 4 details the decomposition and specification of the EMRS into an actor component architecture. In Section 5, we cover the slicing and small model reduction processes we employ to simplify our specifications into a machine-verifiable form. The evaluation of these specifications are summarized in Section 6. Section 7 discusses the static analysis performed on the EMRS which enforces the behavior described in the specifications. Section 8 concludes and surveys future work.

2 BACKGROUND

This section summarizes prior work that we use or build on, including temporal logic, a privacy policy specification language that formalizes HIPAA, the actor model of concurrent computation, and the assume-guarantee format of specifying an open system.

2.1 Temporal Logic

Temporal logic [26] characterizes the behavior of reactive systems in terms of traces, which are sequences of states and/or events. The policy specification language discussed in subsection 2.2 uses a many-sorted, first-order linear temporal logic (FOTL) [12]. Linear temporal logic (LTL) is a variant of temporal logic that deals with a relative ordering of events, such that events are not described with a discrete time variable, but only as how they temporally relate to other events. A brief summary of FOTL follows.

FOTL generalizes propositional LTL as first-order logic generalizes propositional logic. FOTL formulas include non-temporal formulas made up of a base of atomic formulas, with possible variables, logical connectives, and quantifiers. FOTL formulas may also contain unary and binary temporal operators, which operate on the appropriate number of FOTL sub-formulas. We use just a small subset of the standard temporal operators available, which are as follows. *Future Operators.* Henceforth: $\Box\ \phi$ declares that formula ϕ holds in all future states in the trace. Eventually: $\Diamond\ \phi$ declares that formula ϕ holds in some future state in the trace. *Past Operators.* Historically: $\boxminus\ \phi$ declares that formula ϕ held in all preceding states in the trace. Once: $\diamondminus\ \phi$ declares that formula ϕ held in some preceding state in the trace. Since: $\phi_1\ \mathcal{S}\ \phi_2$ declares that ϕ_2 held at some preceding state, and since then ϕ_1 has held in every state.

2.2 Privacy Policy Specification

The privacy policy specification language we have developed [9] decomposes policy formulas into norms by restricting temporal logic formulas to a form similar to that done in the work of Barth et al. on Contextual Integrity [6]. A positive norm allows a message transmission *if* the condition associated with it holds, while a negative norm allows a message transmission *only if* its condition holds. An action is thus allowed by the policy if it satisfies at least one of the positive norms and all the negative norms. A notable difference from Contextual Integrity is that our specification language [9] is limited to a restricted subset of FOTL that makes our privacy policies enforceable.

We use this language to formalize the restrictions and requirements of HIPAA as a set of these communication norms. Of note, the disclosure of Personal Health Information (PHI) is represented by the sending of a message from a principal (i.e., person) that knows information about a subject to another principal that does not. Additionally, principals hold certain roles (e.g., psychiatrist, patient) and messages are sent for specific purposes (e.g., treatment, billing).

A communication action is denoted by $send(p, q, m)$, in which p is the sending principal, q is the receiving principal, and m is the message being sent. Each message contains a set of principal attribute pairs. The predicate $contains(m, q, t)$ holds if message m contains attribute t of subject principal q, such that the recipient of message m would learn the attribute t about q. Roles can be bound

to principals via the *inrole* predicate, where *inrole(p, r)* holds if *(p, r)* ∈ *roleAssignment*. Similarly, *for-purpose(m, u)* enforces that message *m* is sent for purpose *u*.

2.3 The Actor Model and Assume-Guarantee Specification

In the actor model [3, 15], a software system is considered to be a collection of concurrently operating actors that communicate through asynchronous message passing. Each actor has a mailbox through which it receives messages, and an actor may only act in response to these messages. Based on its state/behavior, an actor reacts to the messages it receives one at a time—but not necessarily in the order they arrive—by performing some action. The synchronous actions an actor takes in response to the receipt of an asynchronous message can involve sending a finite number of messages to other actors it knows about, creating a finite number of new actors, or changing its state/behavior so that it will take different actions in response to future messages.

Bengt Johnsson et al. established the use of LTL as sets of assume-guarantee specifications for formally specifying behavior of an open system [18]. In short, the way in which a system interacts with its environment can be specified through assumptions on the behavior of its environment and guarantees on the behavior the system can/will exhibit if the assumptions on its environment hold true. Safety assumptions and safety guarantees describe that bad things cannot happen, by stating conditions that must hold on the information the system receives from its environment and the information the system sends to its environment, respectively. Liveness guarantees describe that good things must eventually happen, by stating conditions that force the system to eventually communicate information to its environment. Assume-guarantee specifications are well-suited for formally specifying the behavior of an actor system by describing under what conditions actors can send or receive certain types of messages.

3 FRAMEWORK

We have developed a framework for implementing actor systems that meet assume-guarantee specifications described above and that can present a compliant system to an end-user (e.g., an auditor) of that system [23]. Broadly, the framework shown in figure 1 takes in formal privacy regulations and organizational policies that are manually refined into assume-guarantee specifications of the information system. This framework is based on and implements our HAPL language [29]. This language can be used to define applications with attendant web-based user interfaces that comply with specification. This compliance is guaranteed by a static analysis phase requiring that the HAPL implementation honors the specification. Failing that, the static analysis phase will return with error contexts explaining the failure. Finally, we evaluate the framework by implementing a functioning EMRS prototype that complies with a representative subset of HIPAA requirements.

3.1 Language and Actor System Generation

HAPL is a general purpose, actor-based, imperative programming language that incorporates history queries. HAPL is split into a user interface specification [24] (which is beyond the scope of this paper),

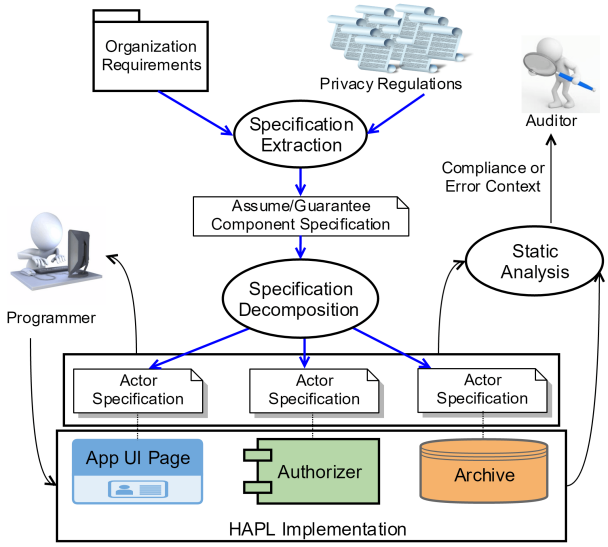

Figure 1: Framework Function

actor behavior specification, static analysis and interpretation. The language, loosely modeled on Scala syntax, allows for defining actors and the principals they represent, as well the role in which actors represent principals. In HAPL, actors are considered to be acting in a particular role for a particular principal whenever they send or receive a message. This assumption simplifies our approach to our assume-guarantee specification and decomposition described in section 4.

HAPL includes a variety of structures for expressing computation, but a subset was implemented due to scope constraints. These structures include: if-statements, assignments, print statements, identifiers in complex expressions, declaration with initialization, tuple definitions, tuple lookups, conses with tails and heads, a *new* command, actor definitions and LTL history queries as shown in figure 2. Example actor definitions can be seen in appendix B.

```
str = "hello" //variable assignment
print( "Here's a message: " + message )
status = "the status is " + ok
var havePatientPrincipal: Bool   = false
phi newPatientRecord: String     = ""
psi newPsychRecord: String       = ""
var names: List[String] = list()
names = "name" :: names
record.get(2) //tuples with lookups
dirElt = tuple( id, principal, name )
var dir: Directory = new Directory( sys)
actor Directory
    [x;inrole(sys,HealthCareProvider)]
    ( sys: Principal ) representing sys
if( newRcrd != "" ) { //code
ltlquery(once receive
    <from PatientActor as P[Patient]>
    (authRelease) with prnc where prnc = subj)
```

Figure 2: Language Features

As per the actor paradigm, computation in HAPL is done through creation of and communication between actors. We constrain HAPL actors to synchronous control only of private data, and communication with other actors only by sending asynchronous messages.

In HAPL, actor behavior must only be triggered in response to messages and, while executing in response to a message, it may only modify local data, create other actors, and send messages to other actors [15, 29]. Through messages, we allow composition of an actor system which is exposed through a web interface. We demonstrate this language by prototyping an EMRS with a number of use cases that are illustrative of the type of specifications needed to specify HIPAA.

We further implement a special control structure, *ltlquery*, which behaves like an if statement. The first block will execute if the LTL expression is true, the second branch otherwise. A representation of the history of messages that an actor has received is kept at each actor which is used to decide the truth of an *ltlquery* condition. This allows us to set up guards to verify that certain messages have been received prior to, for example, releasing PHI.

3.2 Specification and Verification Process

In order to verify the behavior of our EMRS written in HAPL, we first design and detail specifications in section 4. The EMRS is decomposed into a set of communicating actors (figure 1) whose behavior is specified as a set of FOTL formulas in assume-guarantee form. Some of these base level actors are additionally composed into a mid-level component to encapsulate the system's internal communication, and we provide a proven composition theorem for showing the validity of this refinement.

Once we have our EMRS specifications in place, we would like to verify that the system behaves correctly by showing that it is HIPAA-compliant. From our prior work [9], we have a set of FOTL formulas that describe norms of communication in terms of when PHI can be disclosed according to HIPAA. Thus, we can verify HIPAA-compliance of our specifications by showing that any disclosure by our system entails a valid disclosure described in the policy norms.

The natural next step would be to use a theorem prover such as Coq[1] to prove the correctness of this entailment. However we were unable to find an appropriate LTL package that can cleanly implement our sets of FOTL formulas. We turn to using a model checker instead, but model checkers cannot handle first-order formulas due to unbounded state, so in section 5 we take intermediate steps to simplify our input. To reduce computational complexity, we first slice the set of policy norms to remove specification of optional behavior that our system does not perform. We then create small model theorems to reduce both sets of FOTL formulas to larger sets of simpler, behaviorally-equivalent propositional LTL formulas. Lastly, the entailment of the resulting sets are mechanically verified with a model checker in section 6.

3.3 Static Analysis Overview

We use static analysis to bind HAPL code to the constraints detailed in our assume-guarantee specifications. By making statements about type-correctness in assignment, message parameters, message handlers, and send message *inroles*, as well as verifying that certain properties hold in the abstract syntax tree, such as verification that a specific *ltlquery* guards the release of records in the archive, we are able to verify compliance with certain assume-guarantee specifications. The static analysis phase therefore reports non-compliance and fails compilation so that an actor system is not generated where it cannot be proved that the code matches appropriate assume-guarantee specifications. Section 7 goes over this process in more detail.

3.4 Use Cases and HAPL EMRS Prototype

To demonstrate the language and evaluate our framework, we build a EMRS prototype with simple implementations of five use cases specified to comply with HIPAA. These are intended to be a representative subset of the release scenarios described in HIPAA. In particular, this includes cases involving releases that are required, releases that are permitted without consent, and releases that are permitted only with authorization.

While it is possible to fulfill the privacy requirements of HIPAA by rejecting all release requests except for those made by the patient for their own records[2] [16], we designed the following five use cases in order to demonstrate a responsive non-trivial EMRS.

(1) **Record contains no PHI**: If a message contains no PHI, then that information may be released without authorization.
(2) **Patient requests own PHI**: HIPAA has a liveness requirement that any patient request for their own PHI records must eventually be fulfilled. Thus any request by a patient role for their own records must be honored by the EMRS.
(3) **Originating doctor requests patient's psychotherapy note PHI**: A doctor who is acting as a physician for a patient may obtain that patient's psychotherapy notes without the consent of any principal if that physician is also the author of that PHI.
(4) **Doctor requests patient's non-psychotherapy PHI**: A doctor who is acting as the physician for a patient may obtain that patient's PHI without any principal's consent, so the prototype will honor requests by a doctor for patient data.
(5) **Third party requests patient PHI**: Third parties may request patient PHI for marketing purposes. These requests may be honored if it is determined that the patient that is the subject of this PHI has authorized release of it for this purpose.

As part of the framework, HAPL source is compiled into a web application that is run by the Lift framework[3] to generate a web-based UI that interacts with these actors via page definitions controlling the specific actors handling the application logic of a user interface. An example of the web interface for a physician is shown in figure 3. In this example interface, the physician is able to set PHI for a patient (including psychotherapy note PHI) or retrieve a patient's records. Our EMRS prototype implements a representative subset of use cases in HIPAA involving requests for release of records that may contain PHI or even more strongly protected psychotherapy note PHI by various interested parties. This resulting functional

[1] https://coq.inria.fr/

[2] Under HIPAA, a covered entity is also required to notify the Secretary of HHS under certain circumstances, but we make a concession for the simplicity of the EMRS and ignore this obligation.

[3] Liftweb: https://www.liftweb.net/

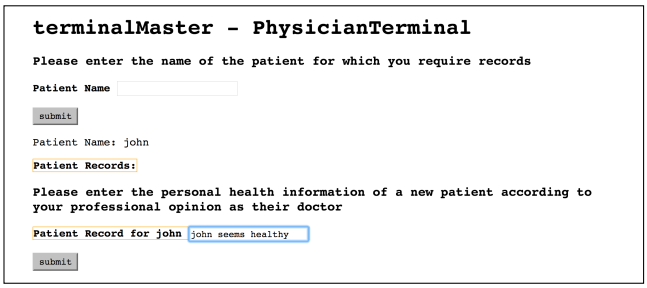

Figure 3: Example Physician Interface Page

prototype represents an evaluation of the framework. The implementation may be open-sourced following review.

4 DECOMPOSITION AND SPECIFICATION

As described in section 3.2, we detail the design of our EMRS by decomposing the system into an actor component specification of FOTL formulas in assume-guarantee form [17]. This is a top-down procedure, but for ease of understanding, we present the details bottom-up. We also describe what is required to verify the consistency of different levels of decomposition, and provide a methodology. Verification of this specification against the EMRS is discussed in section 7.

4.1 System Decomposition

As a first step towards creating specifications for the HAPL EMRS, we conceptualized a decomposition of the system into a set of actors, each of which has a set of responsibilities that it upholds by communicating with the other actors.

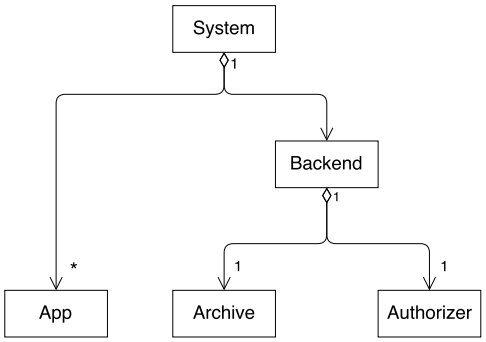

Figure 4: EMRS Decomposition

As shown in figure 4, we decompose the HAPL EMRS into three actor types: the App, the Archive, and the Authorizer.

There are many App actors which act on behalf of human principals and provide them an interface to access the EMRS. Through this interface, a principal may request that the EMRS release medical records to them and receive the corresponding reply. An App also allows patient principals to grant authorization for their records to be released to another principal by sending a message to the Authorizer, or reject an incoming request from the Authorizer for such an authorization. Additionally, an App allows doctor principals to add records to the EMRS by sending them to the Archive.

The Archive and Authorizer are both singleton actors which act on behalf of the organization whose combined behavior can be composed into a component referred to as the Backend. The role of the Archive is to store and retrieve records about patients as requested, and the Authorizer keeps track of information necessary to make policy decisions for the EMRS and makes those decisions for the Archive when requested. When an App sends messages which request some action on a record to be performed, the Archive forwards the necessary information to the Authorizer. The Authorizer then attempts to make the policy decision according to the actor specification and previous system actions. If the policy decision cannot be concretely made from the system history, the Authorizer sends an authorization request message to the App belonging to the subject of the record. After receiving the final decision back from the App if necessary, the Authorizer returns the appropriate response message to the Archive, and the Archive sends the corresponding final reply to the App which made the original request.

4.2 Actor Specification

Actor specifications are the foundation on which higher-level specifications (which have a similar structure) are established. Each actor type has an assume-guarantee specification that prescribes how instances of that actor type may interact with its environment (i.e., the messages it may or must send to other actors).

The language we use for actor specifications shown in figure 5 is built on top of the language we developed to specify policies [9], as prior described in subsection 2.2, and uses a similar meta-variable syntax. The same temporal restrictions apply, most notably that future sub-formulas are only described in β', such that $\Diamond$ can be applied only to positive, non-temporal formulas ($\Diamond\ \mu'$). Likewise, we also do not include the $\ominus$ operator in our past temporal operators available in ψ'. We also specify our formulas as local due to the additional restriction that actors are only aware of actions in which they participate. Note that $R(\vec{x})$ is used to refer to an expression matching the syntax of metavariable R with $\vec{x}$ as the only free variables.

$$
\begin{aligned}
\text{(Local Atomic) } \gamma' &::= R(\vec{x}) \mid true \\
\text{(Local Non-temporal) } \mu' &::= \gamma' \mid \mu' \wedge \mu' \mid \mu' \vee \mu' \mid \exists \vec{x} : \tau . \mu' \mid \\
&\quad \forall \vec{x} : \tau . (\mu'_1(\vec{x}) \rightarrow \mu'_2(\vec{x})) \\
\text{(Local Pure Past) } \psi' &::= \mu' \mid \psi' \wedge \psi' \mid \psi' \, \mathcal{S} \, \psi' \mid \exists \vec{x} : \tau . \psi' \mid \\
&\quad \neg \psi' \mid \forall \vec{x} : \tau . (\mu'_1(\vec{x}) \rightarrow \mu'_2(\vec{x})) \\
\text{(Local Obligation) } \beta' &::= \Diamond \mu' \mid \beta' \wedge \beta' \\
\text{(Local Mixed) } \chi' &::= \beta' \mid \psi' \mid \psi' \rightarrow \beta' \mid \chi' \wedge \chi' \mid \\
&\quad \forall \vec{x} : \tau . \chi'(\vec{x})
\end{aligned}
$$

Figure 5: FOTL Actor Formula Syntax.

Given our actor formula syntax, our actor specifications take the form:

$$\left(\Box \bigwedge_{\psi' \in \text{A-Assum}} \psi'\right) \rightarrow \left(\Box \bigwedge_{\psi' \in \text{A-Safety}} \psi' \wedge \bigwedge_{\chi' \in \text{A-Liveness}} \chi'\right)$$

in which, for the given actor, A-Assum is a set of safety assumption clauses, A-Safety is the set of safety guarantee clauses and A-Liveness is the set of liveness guarantee clauses. Safety assumption clauses $\psi' \in$ A-Assum are restricted to take the form[4]:

$$\forall a : A.\forall m : M.\text{receive}(self, a, m) \rightarrow \psi'(a, m)$$

safety guarantee clauses $\psi' \in$ A-Safety are restricted to take the form:

$$\forall a : A.\forall m : M.\text{send}(self, a, m) \rightarrow \psi'(a, m)$$

and liveness guarantee clauses $\chi' \in$ A-Liveness are restricted to take the form:

$$\forall \vec{x}.\psi'(\vec{x}) \rightarrow \Diamond \exists a : A.\exists m : M.\mu'(a, m, \vec{x}) \wedge \text{send}(self, a, m)$$

Here we assume that $\forall \vec{x}$ includes specification of appropriate sorts. The requirements on the syntactic forms of safety assumption and safety guarantee clauses (an implication with a receive by *self* or a send by *self*, respectively, as antecedent) serve to syntactically isolate a specific action, so that the clause as a whole describes under what circumstances that action is permitted. Furthermore, this restriction serves to ensure that the safety clause is incrementally verifiable, and moreover, that violations of the safety clause can be associated with a specific send action. Similarly, liveness clauses are restricted to take the form of an implication which has an antecedent that contains a local pure past formula, whereas the consequent is restricted to requiring the actor to send some action in the future as constrained by a local non-temporal formula.

To show how these clauses are used for our EMRS, we provide some examples. One Archive safety assumption states that it will only receive an authorization or rejection to release a record if it once sent out the corresponding request. In contrast, a safety guarantee for the Archive says that it may only send a message that releases a record if it once received an authorization to release that record. A liveness guarantee for the Authorizer states that if it receives a request to authorize release of a patient's PHI, and the request for the PHI originated from the subject of that PHI, it is obligated to eventually authorize that request.

4.3 Component Specification

Specifications for components are similarly structured, with clauses on communication taking place between actors internal and external to the component.

Figures 6 and 7 show a safety guarantee for the EMRS Backend component [17] that stores and releases records and an associated negative policy norm [9] that it enforces. This norm in the context of the policy specifications states that a communication of psychotherapy notes from the covered entity to another entity implies that the individual that the psychotherapy notes are about gave authorization for the communication. The safety guarantee maps on top of the norm, with the local component actor *self*, which acts on behalf of covered-entity *p1*, sending a message containing a record to the requesting actor *a*, which acts on behalf of recipient *p2*. As such, the safety guarantee enforces that a release of a record containing psychotherapy notes can only take place if the component prior received a message from the individual which granted authorization. Note that for flexibility, the safety guarantee also allows communication to take place if this is not a true disclosure, either because there is not PHI involved or it is communicated to an individual that has access by definition.

We skip formal specification of the fully composed system, as there is not any external communication to describe due to the scope of our EMRS prototype. However, the decomposition and verification techniques we describe between components and their comprising actors follow similarly for higher-level components which decompose into subcomponents.

4.4 Verification of Component Specification

In principle, a component's specifications are shown to be correctly implemented by showing that they are entailed by the composition of specifications of the subcomponents or actors that make up the component. Without loss of generality, we will focus on a component composed directly of actors and assume that there is only one assumption, guarantee, and liveness clause per component/actor. These clauses should be of the syntactic form described above for actors and of an analogous form for components.

The following theorem establishes sufficient conditions (ASSUMPTION CONSISTENCY, SAFETY REFINEMENT, and LIVENESS REFINEMENT) for showing that the specification of the actors that make up a component (SAFETY VALID and LIVENESS VALID) are a correct refinement of the specification of the component (COMPONENT SPEC).

THEOREM 4.1 (COMPONENT REFINEMENT). *Given, a component C made up of actors $\mathcal{A}$, an assumption about operation causality $\chi^A_{\cdot\cdot}$, a specification of C (consisting of an assumption ψ^A_C, a safety guarantee ψ^S_C, and a liveness guarantee χ'_C), a specification of each actor $i \in \mathcal{A}$ (consisting of an assumption $\psi_i'^A$, a safety clause $\psi_i'^S$, liveness clause χ'_i), and a universe Σ of infinite traces of interleaved send or receive operations, then the component specification is valid, that is, for any trace $\sigma \in \Sigma$ for which $\sigma \models \chi^A_{\cdot\cdot}$,*

$$\sigma \models \psi^A_C \rightarrow \psi^S_C \wedge \chi'_C \qquad \text{(COMPONENT SPEC)}$$

if every individual actor specification is valid, i.e., for any trace $\sigma \in \Sigma$ for which $\sigma \models \chi^A_{\cdot\cdot}$,

$$\sigma \models \bigwedge_i \left(\Box\psi_i'^A \rightarrow \Box\psi_i'^S\right) \qquad \text{(SAFETY VALID)}$$

$$\sigma \models \bigwedge_i \left(\Box\psi_i'^A \rightarrow \Box\chi'_i\right) \qquad \text{(LIVENESS VALID)}$$

and for any trace $\sigma \in \Sigma$, for which $\sigma \models \Box\psi^A_C \wedge \chi^A_{\cdot\cdot}$,

$$\sigma \models \bigwedge_i \Box\psi_i'^S \rightarrow \bigwedge_i \Box\psi_i'^A \qquad \text{(ASSUMPTION CONSISTENCY)}$$

$$\sigma \models \bigwedge_i \Box\psi_i'^S \rightarrow \Box\psi^S_C \qquad \text{(SAFETY REFINEMENT)}$$

$$\sigma \models \bigwedge_i \Box\psi_i'^S \rightarrow \Box\chi'_C \qquad \text{(LIVENESS REFINEMENT)}$$

PROOF. Suppose an arbitrary σ for which $\sigma \models \psi^A_C \wedge \chi^A_{\cdot\cdot}$. It must be the case that

$$\sigma \models \Box \bigwedge_i \left(\psi_i'^A \rightarrow \psi_i'^S\right), \qquad (1)$$

[4] *self* is a metavariable referring to the local actor which is used to restrict specification to local communication.

$$\forall a : A.\forall m : M.(\text{send}(self, a, m) \rightarrow \forall subject : P.\forall record : E.\forall requester : P.\forall role : R.\forall id : I.(\text{releaseRecord}(m, subject, record) \rightarrow$$
$$\exists m' : M.\diamondsuit(\text{receive}(self, a, m') \wedge \text{requestReadRecord}(m', subject, id) \wedge \text{recordId}(record, id) \wedge \text{recordSubject}(record, subject)$$
$$\wedge \text{messageSender}(m', requester) \wedge \text{messageSenderRole}(m', role) \wedge (\neg\text{containsPHI}(record) \vee (requester = subject)$$
$$\vee (role = \text{“}HealthCareProvider\text{”} \wedge (\neg\text{containsPsychNotes}(record, subject) \vee \text{recordOriginator}(record, requester)))$$
$$\vee (\exists a' : A.\exists m'' : M.\diamondsuit(receive(self, a', m'') \wedge \text{grantAuthorization}(m'', requester, role, id) \wedge \text{messageSender}(m'', subject)))))))$$

Figure 6: Example Safety Guarantee

$$\S164.508(a)(2) : \text{inrole}(p1, coveredentity) \wedge \text{inrole}(q, individual) \wedge in(t, psychnotes) \rightarrow obtainedAuthorization(p1, p2, q, t, u)$$

Figure 7: Example Policy Norm

because otherwise there would be some i, some prefix σ' of σ and the infinite extension σ'' of σ' by stuttering for which $\sigma', |\sigma'| \models \psi_i'^A$, $\sigma', |\sigma'| \not\models \psi_i'^S$, and $\sigma'' \not\models \Box\psi_i'^A \rightarrow \Box\psi_i'^S$, contradicting SAFETY VALID. Similarly, from ASSUMPTION CONSISTENCY, it must be that

$$\sigma \models \Box\left(\bigwedge_i \psi_i'^S \rightarrow \bigwedge_i \psi_i'^A\right). \tag{2}$$

Now, note that (as in the logic of Jonsson and Yih-Kuen [18]) in our FOTL, for any pure past FOTL formula ψ, $\sigma \models \Box\psi$ if and only if $\sigma \models \Box\boxminus\psi$. From, this, and the mutual exclusivity of send and receive operations, we can rewrite the equations 1 and 2 (using $\ominus$ for "weak yesterday")[5] as:

$$\sigma \models \Box\boxminus \bigwedge_i \left(\ominus\psi_i'^A \rightarrow \psi_i'^S\right) \tag{3}$$

$$\sigma \models \Box\boxminus \bigwedge_i \ominus\psi_i'^S \rightarrow \bigwedge_i \psi_i'^A \tag{4}$$

Now we can derive,

$$\sigma \models \Box\boxminus\left(\bigwedge_i \psi_i'^A \wedge \bigwedge_j \psi_j'^S\right) \tag{5}$$

by induction on the prefixes of σ and using equations 3 and 4. This can be rewritten as

$$\sigma \models \Box \bigwedge_i \psi_i'^A \wedge \bigwedge_j \psi_j'^S \tag{6}$$

using the equivalence of $\Box\psi$ and $\Box\boxminus\psi$. Finally, by modus ponens and equations 6, SAFETY REFINEMENT, LIVENESS VALID, and LIVENESS REFINEMENT, we have $\sigma \models \psi_C^S \wedge \chi_C'$. □

In applying this theorem, the causality assumption should specify that no message is received without first being sent and every message that is sent will eventually be received:

$$\chi_{\cdot\cdot}^A \equiv \begin{array}{c}(\forall a_1.\forall a_2.\forall m.\text{receive}(a_2, a_1, m) \rightarrow \diamondsuit\text{send}(a_1, a_2, m)) \\ \wedge \\ (\forall a_1.\forall a_2.\forall m.\text{send}(a_1, a_2, m) \rightarrow \diamondsuit\text{receive}(a_2, a_1, m))\end{array} \tag{7}$$

Due to the small scale of the actor component set in our EMRS specifications, we leave the formal proof of the composition of the Archive and Authorizer into the Backend as future work. In lieu of this, we provide an example on how our composition is applied, by showing the composition of the following three safety guarantees for our EMRS actors. For brevity, we replace the FOTL with an English approximation:

(1) sg_1 on *Archive* a_1: outgoing *requestAuthorizeRelease* (m_2 to *Authorizer* a_2) follows incoming *requestReadRecord* (m_1 from *App* a_3) since confirm add was sent (to some *App*)
(2) sg_2 on *Authorizer* a_2: outgoing *authorizeRelease* (m_3 to *Archive* a_1) follows incoming *requestAuthorizeRelease* (m_2 from *Archive* a_1) and authorization subclause holds
(3) sg_3 on *Archive* a_1: outgoing *releaseRecord* (m_4 to *App* a_3) follows incoming *authorizeRelease* (m_3 from *Authorizer* a_2)

We compose these guarantees together by removing all internally communicated messages but keeping any restrictions placed on their communication. From this, we obtain sg_4 on *Backend* c: outgoing *releaseRecord* (m_4 to *App* a_3) follows incoming *requestReadRecord* (m_1 from *App* a_3) since *confirmAdd* was sent (to some *App*) and authorization subclause holds. Note that this is the safety guarantee displayed in figure 6, though the since clause that sanity checks that the record exists has been trimmed for brevity.

5 SLICING AND REDUCTION

As discussed in section 3.2, our FOTL specifications are too computationally complex to be analyzed with model checking. A small model theorem states that we can use finite sets of elements from infinite carriers to reduce a language to a propositional form that represents a target behavior [13]. In this section, we detail a preliminary slicing procedure, followed by the creation of small model theorems which we use to reduce our slices of the EMRS specifications and policy specifications to propositional input for model checking in section 6.

5.1 Component and Policy Slicing

Before creating our small model theorems, we first want to simplify our input sets by slicing out the portions relevant to our models from the full sets of formulas. For our EMRS specifications, we only need the specifications which map to the disclosures in the entailment we are trying to prove. The entailment we want to show is that any record release case described in the EMRS specifications entails a corresponding PHI release case in the policy norms. Notably, our specifications do not describe any behavior external

[5]The $\ominus$ oeprator is not part of our specification language, but $\ominus\psi$ is used here with its standard LTL semantics to use the prior rather than current operation in evaluating the antecedant.

to the system at the top level and the App is simply a UI that delivers messages between the user and the rest of the EMRS. Thus we keep only the safety and liveness guarantee specifications for the Backend component that specify when the system can or must release records.

As the policy specifications are very large, we must perform a more involved slice. Specifically, we remove many exceptions to requirements for PHI transmissions as well as some enabling release cases which are not applicable to our EMRS prototype. We additionally remove some restrictions on PHI transmission to family members or related equivalents, however these releases then fall to a more general negative norm which is stricter. These removals are sound as the resulting set of policy norms are at least as strict as before about releasing PHI. We make one concession on soundness by removing a liveness requirement which requires transmission to a secretary for investigation, as our EMRS prototype currently does not model this obligation.

5.2 Component Slice Reduction

We create the small model theorem in theorem 5.1 for our component slice. More details on the resulting set of finite carriers and the restrictions we place on combinations can be found in appendix A.

Theorem 5.1 (EMRS Small model theorem). *The EMRS specifications allow the release of a record for infinite carriers of sorts $\mathcal{I}$, $\mathcal{E}$, $\mathcal{P}$, $\mathcal{R}$, $\mathcal{A}$, $\mathcal{M}$ if and only if the EMRS specifications allow the release of a record for finite carriers $\hat{\mathcal{I}}$, $\hat{\mathcal{E}}$, $\hat{\mathcal{P}}$, $\hat{\mathcal{R}}$, $\hat{\mathcal{A}}$, $\hat{\mathcal{M}}$ in which $|\hat{\mathcal{I}}| = 3$, $|\hat{\mathcal{E}}| = 2$, $|\hat{\mathcal{P}}| = 5$, $|\hat{\mathcal{R}}| = 3$, $|\hat{\mathcal{A}}| = 5$, and $|\hat{\mathcal{M}}| = 7$.*

We consider two types of records, one that contains normal PHI and one that contains psychotherapy note PHI. We do not represent a record containing no PHI, as its release is irrelevant to HIPAA. We match each of these records to a corresponding id. We also add an id that refers to no existing record to model the behavior where a nonexistent record is requested. This does not map to any specified behavior in HIPAA but provides completeness for the specifications which explicitly handle this case. This covers all distinct types of PHI that would affect release behavior as specified by HIPAA.

We consider three types of roles, which are doctor, patient, and third party which cover all distinguishing role behavior in the EMRS specifications. We then match these roles to the principals that act in them. One third party principal is sufficient, but we split doctors and patients into two principals each due to distinguishing behavior for record originators and subjects, respectively. This covers all behaviorally distinct principals in the specifications. Equivalently, we could have had multiple instances for each record instead. We also match these principals to the actors, with each principal getting their own actor. As actors are a wrapper used by the system to communicate with the principals they act on behalf of, one for each principal is sufficient. As we are only concerned with communication coming in and going out of the EMRS for release behavior, we do not need to model actors for the system itself.

We model a different message instance for each message type used in the Backend specification guarantees, which map simply into each place they are used, as every specification always uses a predicate to match the type of message being discussed. One instance of each message is sufficient to cover specification behavior, as they are just wrappers for describing the type of communication taking place.

5.3 Policy Slice Reduction

Before we attempt to develop a small model theorem for our policy slice, we impose a restriction proposed in [9], which aids in the concreteness of the model by forcing each send event to be for a single message containing a single attribute sent for a single purpose to a single principal. To this end, we remove the *contains* and *for-purpose* predicates from the policy and move those bindings into the *send* predicate, such that its new signature is $\mathcal{P} \times \mathcal{P} \times \mathcal{P} \times \mathcal{T} \times \mathcal{U}$, and then remove the message sort ($\mathcal{M}$) entirely. This enforces our restriction by forcing all specification of message contents into the new send predicate send($p1, p2, q, t, u$), whose signature must obey it. We have the resulting small model theorem for our policy slice in theorem 5.2.

Theorem 5.2 (Policy Slice Small model theorem). *The sliced policy allows the release of PHI for infinite carriers of sorts $\mathcal{T}$, $\mathcal{U}$, $\mathcal{P}$ if and only if the sliced policy allows the release of PHI for finite carriers $\hat{\mathcal{T}}$, $\hat{\mathcal{U}}$, $\hat{\mathcal{P}}$ in which $|\hat{\mathcal{T}}| = 3$, $|\hat{\mathcal{U}}| = 6$, $|\hat{\mathcal{P}}| = 6$.*

Three types of attributes exist in the policy slice, which include PHI, psychotherapy notes (PSN), and directory information (dii). For the carrier of attributes $\mathcal{T}$ in our policy slice, PSN,PHI,dii $\in \mathcal{T}$, PSN $\in$ PHI, and dii $\notin$ (PHI $\cup$ PSN). We divide these sets into three distinct attributes such that t1 $\in$ (PHI $\setminus$ PSN), t2 $\in$ (PHI $\cap$ PSN), and t3 $\in \mathcal{T} \setminus$ (PHI $\cup$ PSN). These three distinct attributes can simulate all possible attributes in the policy slice. The purposes found in our policy slice include access, treatment, payment, health-care-operations, and marketing. We additionally add other-purpose to capture purposes other than those explicitly labeled for the sake of completeness. These six distinct purposes capture all possible purposes present in the policy slice. For the carrier of principals $\mathcal{P}$ in our policy slice, covered-entity,patient,provider,work-force-member $\in \mathcal{P}$, provider $\in$ work-force-member, work-force-member $\in$ covered-entity. We initially model one principal for each of these roles, and one additional principal with no roles. One principal for each role is sound as long as the policy does not differentiate between two principals in a role. This is not true of principals in the patient role, which can be checked if they are the same principal that is the subject of PHI. Thus, we split patient into two distinct principals, one of which is the subject of all PHI and the other which is the subject of none. We additionally divide the subset roles to be distinct from their supersets, as done before with attributes. Our model thus contains six principals: p1 $\in$ (covered-entity $\setminus$ (work-force-member $\cup$ provider)), p2 $\in$ (covered-entity $\cap$ (work-force-member $\setminus$ provider)), p3 $\in$ (covered-entity $\cap$ (work-force-member $\cap$ provider)), p4,p5 $\in$ patient, p6 $\in$ ($\mathcal{T} \setminus$ covered-entity $\cup$ patient $\cup$ provider $\cup$ work-force-member).

6 EVALUATION OF EMRS SPECIFICATION

After using small model theorems to reduce the first-order nature of the privacy policy and EMRS specifications to a finite set of elements, we are able to apply model checking techniques to evaluate whether the assume-guarantee specifications entail the privacy policy in the EMRS prototype. Automation of the proof process

provides a level of confidence in the validity of the EMRS prototype that is essentially too tedious, or even infeasible, to achieve by manual means.

6.1 Proof Using Model Checking

We use model checking to verify that the EMRS specifications imply all negative norms and at least one positive norm of the privacy policy before any disclosure of PHI. We selected the symbolic model checker NuSMV [10] to perform the checking because the size of the specification formulas can be large, causing the state space of the model to be large. Additionally, NuSMV supports both past and future temporal logic operators, which are used in both the formal policy norms and EMRS specifications.

NuSMV is a symbolic model checking tool for verifying a model of a finite-state system against properties specified in temporal logic. The model is specified by states and state transitions. A state is an assignment of all variables within the system. A state transition defines an allowable change in variable assignments in response to an event or condition.

NuSMV supports only variables of finite-range. We use two types of terms for our model: environmental variables and derived variables (macros). An environmental variable is one not directly controlled by the system, but that affects the system's behavior (e.g., an actor requesting authorization to read a record). In this project, macros are used to define terms from the FOTL that have become constants after the application of the small model theorem.

We use our small model to identify the finite elements sufficient to encode the terms of the FOTL EMRS and policy specifications in NuSMV. NuSMV then symbolically determines if the formulas are verified, and if a violation is found, a counterexample is generated.

6.2 NuSMV Encoding

We wish to show that the EMRS satisfies the privacy policy. To do this, the EMRS specifications must entail all of the negative norms and at least one positive norm of the privacy policy for each disclosure of PHI. To show this, we encode EMRS entailment formulas, in LTL form, in NuSMV using the small model and FOTL reductions from section 5.

Environmental variables were defined as booleans since the EMRS specification and policy norm variables could be true or false at any given time. For example, the message *requestReadRecord(requestReadRecord, Subject, PHI)* contains a request to read a record containing PHI about a particular subject. Any actor can send this request at any time. So, at any moment the request is either true (it occurred) or false (it did not occur). These variables are encoded as environmental for two reasons. The first is that these variables (like the request to read a record just mentioned) are not controlled by the EMRS, and are instead generated by actors from outside the EMRS. The second is that we wish to explore all possible combinations and occurrences of events of the privacy policy norms and EMRS specifications. That is, we wish to be certain that the EMRS satisfies the privacy policy under all possible scenarios.

While most terms from the policy specification are encoded as boolean environmental variables, there are some that have been encoded as macros with values *TRUE* or *FALSE*. This was a consequence of the reduction and transformation of the FOTL formulas and variables to propositional LTL. For example, *in(t, phi)* checks if record *t* includes any PHI. When reduced and transformed to propositional form, three distinct values took the place of *t*: *in(psychotherapy-notes, phi)*, *in(phi, phi)*, and *in(dii, phi)*. Each of these checks if a record that contains psychotherapy notes contains PHI, if a record that contains PHI contains PHI, and if a record that is directory information contains PHI, respectfully. Because all variables were encoded as booleans, each of these propositional variables must evaluate to true or false: psychotherapy notes are PHI by definition so *in(psychotherapy-notes, PHI)* must always evaluate to true, *in(phi, phi)* is vacuously always true, and directory information contains no PHI so *in(dii, phi)* must always be false. Because they are always true or always false at any given state, they have been encoded as macros.

6.3 Results

Model checking uncovered some subtle errors in our own preliminary specifications. Additionally, verification of the specifications found some violations for some of the formulas. The violations were due to a temporal logic statement involving subjects granting authorization to certain actors. Since all variables were listed as environmental, there were times when a previous state had granted authorization to release a record, but in a following state that authorization was changed to false. In the system, an authorization only needs to be granted once for the record to be released at any later time. This conflict arose as a result of the encoding process – specifically the use of unconstrained booleans to model external inputs into the system. This caused certain release cases to evaluate to false when they were, in fact, true. A separate pre-condition was encoded in order to overcome this scenario, stating simply that if authorization was granted once in a previous state for a record release that record release should still be authorized for the current state.

No other violations or conflicts were detected by NuSMV. All entered formulas were satisfied by the EMRS. This means that the EMRS specifications implied at least one positive and all negative norms, thereby showing that EMRS entailed the privacy policy. NuSMV was tested on both Windows 7 and Linux environments, 64-bit OS, with 8GB of memory. NuSMV completed in an average of 2.3 seconds.

7 STATIC ANALYSIS

Beyond basic syntax and type checking, HAPL's static analysis is intended to ensure that the implementations of the actors meet the assume-guarantee conditions upon which the model checking relies. The current implementation is rudimentary, but the ultimate approach is to employ an iterative data-flow analysis to compute, for each location in an action method, information about the histories that reach that location.

This information takes the form of three-valued truth assignments over sub-formulas of the *formulas of interest*: the assume-guarantee formulas for, and temporal queries within, the given action method. Each sub-formula that is assigned *true* is satisfied by every reaching history; each that is assigned *false* is not satisfied

by any reaching history. Finally, if the formula is neither known to be satisfied by all reaching histories nor is it known to be satisfied by no reaching history, then the formula is assigned *unknown*.

Each iteration of the data-flow analysis computation simulates the effect that executing the program has on the truth value of the *formulas of interest*. Consider the code of figure 8 that describes the steps taken by the Authorizer in determining whether the *sender* should be allowed access to the PHI of *subject*. In short, it authorizes the release of the PHI only if the patient has previously authorized its release.

```
1  //declarations and lookup method omitted
2  actor Authorizer [ auth ; inrole( sys, CareProvider ) ](dir:
       Directory, sys: Principal) representing sys {
3    on requestAuthRelease[]( sender: Actor, requestingPrinc:
       Principal, subject: Principal, record: Tuple[Int,Principal,
       Principal,String] ) {
4      originator = record.get(3)
5      data = record.get(4)
6      if( (!isPHI( data )) || ((requestingPrinc == originator) || (
         requestingPrinc == subject)) ) {
7        send<sender>(authRelease(this,sys,record))
8      } else {
9        ltlquery(once receive<from PatientTerminal as P[Patient]> (
         authOwnRecordFor) with owner, releaseTo where owner = subject
          and requestingPrinc = releaseTo ) {
10         send<sender>(authRelease(this,sys,record))
11       } else {
12         requests = tuple(sender,subject,record)
13         send<dir>(lookupPrinc(this,subject))
14   }}}
15   on authOwnRecordFor[]( owner: Principal, releaseTo: Principal )
       {
16     foreach req in (requests) {
17       subjectPrinc = (req.get(3)).get(3)
18       if( subjectPrinc == owner ) {
19         requestor = req.get(1)
20         subjectRecord = req.get(3)
21         ltlquery( once receive<from Patient as P[Patient]> (
         requestAuthorizeRelease) with subject where subject = owner )
          {
22           send<requestor>(authRelease(this,sys,subjectRecord))
23         } else {
24           print("preemptive auth by: "+owner)
25   }}}}}
```

Figure 8: Authorizer Fragment

From line 2 of the assumptions we learn that *sys* is a *CareProvider* and that information is known to be true at each statement within the method. The static analysis also adds that the condition statement of the *ltlquery* on line 9 is *true* to any statement within its *then* branch of the (in this case only the *send* statement of line 10) and that it is *false* to any statement on its *else* branch. In this case, the static analysis determines that the actor will authorize the sending of medical records for a patient if that same patient has authorized its release in the past.

This analysis is essentially a generalization of a three-value version of the approach of Krukow *et al.* [19], which propagates information from sub-formulas to formulas. Since the query-conditional statements and assumption clauses contain *formulas of interest* directly, it is possible for those statements to 'learn' new facts that must be propagated to the sub-formulas.

The current implementation falls short of this general analysis, but still ensures that the assume-guarantee specifications are met by requiring that conditionals in the code precisely match those of the specifications. That is, it can not infer from a pair of nested query-conditionals that their conjunction is satisfied. Yet, it still provides a guarantee that the runtime code will never violate the specifications upon which its correctness has been evaluated by the model checking step. It is also less general in that a number of HIPAA-specific terms are hard-coded into the implementation for the sake of expedience.

8 CONCLUSION

In this paper we have presented a methodology for designing and implementing verifiable information systems. We have developed the design of HAPL into a full framework and created an EMRS prototype in an actor component architecture. We additionally develop assume-guarantee specifications for this architecture which we statically verify enforce the HIPAA privacy policy. Though this is a proof of concept, we believe that this direction of building information systems with verifiable privacy policy compliance is promising. To conclude, we inventory some areas of future work we have left to explore.

Our prototype EMRS currently supports a limited number of use cases designed to be representative of several key points in HIPAA, but a full system would need to be further developed. Another use case that would be desirable to support includes releasing records to a delegate, such that an individual who has been granted delegation privileges for a subject should be treated the same as the subject. Additionally, a full system should provide a path for a subject to revoke authorization, and an appropriate use case would check that a previously authorized record is not released after the authorization is revoked. There also exist many additional permitted release cases an EMRS can implement, such that the EMRS can release records in these cases without requiring authorization from the subject.

We provide a theorem with an accompanying proof that can be leveraged towards this end, however we do not formally verify that the component specification for our EMRS is entailed by the specification of its comprising actors. As our refinement is very simple, we instead focused our efforts on proving entailment of the system specification to the policy specifications. A full system with a larger actor component architecture would require a stronger assurance that the component refinement step is performed correctly, especially as the system changes.

In its current form, the static analysis is limited in two ways that reduce its general applicability: first, it requires a perfect match between the assume-guarantee specification and the formulas in the query conditional, and second, it is hardcoded to use HIPAA roles and messages. Ideally, the first issue would be addressed by implementing the more general algorithm described in section 7. Barring that, at least some common cases, like conjoining the formulas of nested query-conditionals and allowing commutations such as $A \wedge B = B \wedge A$, should be supported. Parameterizing the static analysis with respect to roles and messages will enable the framework to be used for other specifications.

ACKNOWLEDGEMENT

The authors are supported in part by NSF award CNS-0964710 and by the NSA Grant on Science of Security.

REFERENCES

[1] 1999. Federal Trade Commission, How to comply with the children's online privacy protection rule. (1999). DOI: https://doi.org/bcp/conline/pubs/buspubs/coppa.htm Public Law.

[2] 1999. Senate Banking Committee, Gramm-Leach-Bliley Act. (1999). Public Law 106-102.

[3] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. 1997. A foundation for actor computation. *Journal of Functional Programming* 7, 01 (1997), 1–72. DOI: https://doi.org/10.1017/S095679689700261X arXiv:http://journals.cambridge.org/article_S095679689700261X

[4] Bowen Alpern and Fred B. Schneider. 1987. Recognizing safety and liveness. *Distributed Computing* 2 (1987), 117–126. Issue 3. http://dx.doi.org/10.1007/BF01782772

[5] Paul Ashley, Satoshi Hada, GÃijnter Karjoth, Calvin Powers, and Matthias Schunter. 2003. Enterprise Privacy Authorization Language (EPAL 1.2). (November 2003). W3C Member Submission.

[6] Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. 2006. Privacy and Contextual Integrity: Framework and Applications. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 184–198. DOI: https://doi.org/10.1109/SP.2006.32

[7] Lujo Bauer, Jarred Ligatti, and David Walker. 2002. More Enforceable Security Policies. In *Foundations of Computer Security*.

[8] Travis Breaux and Annie Antón. 2008. Analyzing Regulatory Rules for Privacy and Security Requirements. *IEEE Trans. Softw. Eng.* 34, 1 (2008), 5–20.

[9] Omar Chowdhury, Andreas Gampe, Jianwei Niu, Jeffery von Ronne, Jared Bennatt, Anupam Datta, Limin Jia, and William H Winsborough. 2013. Privacy promises that can be kept: A policy analysis method with application to the HIPAA privacy rule. In *Proceedings of the 18th ACM symposium on Access control models and technologies*. ACM, 3–14.

[10] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. 2000. NuSMV: A New Symbolic Model Checker. *International Journal of Software Tools for Technology Transfer* 2 (2000), 410–425.

[11] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. 2001. The Ponder Policy Specification Language. In *POLICY*. Springer-Verlag, London, UK, 18–38.

[12] E. Allen Emerson. 1990. Temporal and modal logic. In *Handbook of theoretical computer science*, Jan van Leeuwen (Ed.). MIT Press, Cambridge, MA, USA, 995–1072. http://portal.acm.org/citation.cfm?id=114891.114907

[13] E. Allen Emerson and Kedar S. Namjoshi. 1995. Reasoning about rings. In *POPL '95*.

[14] Klaus Havelund and Grigore Roşu. 2004. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.* 6 (2004), 158–173. Issue 2. DOI: https://doi.org/10.1007/s10009-003-0117-6

[15] Carl Hewitt. 1977. Viewing control structures as patterns of passing messages. *Artificial Intelligence* 8, 3 (1977), 323 – 364. DOI: https://doi.org/DOI:10.1016/0004-3702(77)90033-9

[16] HIPAA 1996. Health Insurance Portability and Accountability Act (HIPAA). (1996). (42 U.S.C. §300gg, 29 U.S.C §1181 *et seq.*, and 42 U.S.C §1320d *et seq.*; 45 CFR Parts 144, 146, 160 162, and 164).

[17] Claiborne Johnson. 2016. *An Actor-Based Framework for Verifiable Privacy Policy Enforcement: Assume-Guarantee Specification of an Actor-Component Architecture.* Master's thesis. University of Texas at San Antonio.

[18] Bengt Jonsson and Tsay Yih-Kuen. 1996. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science* 167, 1-2 (1996), 47 – 72. DOI: https://doi.org/10.1016/0304-3975(96)00069-2

[19] Karl Krukow, Mogens Nielsen, and Vladimiro Sassone. 2008. A Logical Framework for History-based Access Control and Reputation Systems. *J. Comput. Secur.* 16, 1 (Jan. 2008), 63–101. http://dl.acm.org/citation.cfm?id=1370684.1370686

[20] L. Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering* 3, 2 (1977), 125–143.

[21] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303. DOI: https://doi.org/DOI:10.1016/j.jlap.2008.08.004 The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[22] Jay Ligatti, Lujo Bauer, and David Walker. 2009. Run-Time Enforcement of Nonsafety Policies. *ACM Trans. Inf. Syst. Secur.* 12, Article 19 (January 2009), 41 pages. Issue 3. DOI: https://doi.org/10.1145/1455526.1455532

[23] Thomas MacGahan. 2016. *Towards Verifiable Privacy Policy Compliance of an Actor-Based Electronic Medical Records System: An extension to the HAPL language focused on exposing a user interface.* Master's thesis. University of Texas at San Antonio.

[24] Thomas MacGahan, Claiborne Johnson, Armando Rodriguez, Jeffery von Ronne, and Jianwei Niu. 2017. Provable Enforcement of HIPAA-Compliant Release of Medical Records Using the History Aware Programming Language. *In Proceedings of 22nd ACM Symposium on Access Control Models and Technologies*, 10.

[25] Michael J. May, Carl A. Gunter, and Insup Lee. 2006. Privacy APIs: Access Control Techniques to Analyze and Verify Legal Privacy Policies. In *CSFW*. IEEE Computer Society, Washington, DC, USA, 85–97. DOI: https://doi.org/10.1109/CSFW.2006.24

[26] A. Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, Vol. 526. 46–67.

[27] Fred B. Schneider. 2000. Enforceable security policies. *ACM Transactions on Information and System Security* 3 (2000), 2000.

[28] A. Prasad Sistla. 1994. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing* 6 (1994), 495–511. Issue 5. http://dx.doi.org/10.1007/BF01211865 10.1007/BF01211865.

[29] Jeffery von Ronne. 2012. Leveraging actors for privacy compliance. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*. ACM, 133–136.

A REDUCED EMRS CARRIERS

Table 1: Reduced EMRS Carriers

Finite Carrier	Instances
$\hat{\mathcal{I}}$ (id)	idPHI, idPsychPHI, idNotExists
$\hat{\mathcal{E}}$ (record)	PHIRec, PsychPHIRec
$\hat{\mathcal{P}}$ (principal)	subject, nonSubject, originator, nonOriginator, thirdParty
$\hat{\mathcal{R}}$ (role)	doctor, thirdParty, patient
$\hat{\mathcal{A}}$ (actor)	subjectApp, nonSubjectApp, originatorApp, nonOriginatorApp, thirdPartyApp
$\hat{\mathcal{M}}$ (message)	requestReadRecord, releaseRecord, rejectReadRequest, requestGrantAuth, grantAuthorization, rejectGrantAuthorizationRequest, confirmAdd

The finite carriers resulting from theorem 5.1 can be found in table 1. Although the sets of instances in our carriers would seem to create an overly large combination, instances are combined strictly along valid bindings. For example, we do not combine the *doctor* instance of $\hat{\mathcal{R}}$ with the *subject* instance of $\hat{\mathcal{P}}$, nor do we combine the *idPsychPHI* instance of $\hat{\mathcal{I}}$ with the *PHIRec* instance of $\hat{\mathcal{E}}$, as these do not create logical pairings in the system. Additionally, we do not need to combine with all instances of $\hat{\mathcal{I}}$ and $\hat{\mathcal{M}}$ for every specification. We only need to use the *idNotExists* element from $\hat{\mathcal{I}}$ when discussing an id received in a *requestReadRecord* message, as it has no corresponding record and will be precluded from progressing into following system messages that require a record to be identified. We also only need to combine with instances of $\hat{\mathcal{M}}$ when they are matched to their predicate in the specifications, with multiple instances of $\hat{\mathcal{M}}$ only necessary when an instance can be matched to one of multiple message predicates, which occurs only in a couple specifications.

B HAPL LANGUAGE EXAMPLES

This appendix includes examples of a couple actor definitions in HAPL, as well as some supporting role and message definitions:

```
role CareProvider
role Patient
role Marketing
role Doctor
message requestAuth[ x ; inrole( sys, CareProvider ) ]( sender:
      Authorizer, sys: Principal )
message decideAuth[]( sender: Actor, requestor: Principal,
      subjectRecord: Tuple[Int,Principal,Principal,String] )
message authRelease[ x ; inrole( authingPrinc, CareProvider ) ](
      sender: Actor, authingPrinc: Principal, record: Tuple[Int,
      Principal,Principal,String] )
message authOwnRecordFor[ x ; inrole( owner, Patient ) ]( owner:
      Principal, releaseTo: Principal )
message requestAuthRelease[ x ; inrole( sendingPrinc, CareProvider
      ), inrole( subject, Patient ) ]( sender: Actor, sendingPrinc
      : Principal, subject: Principal, record: Tuple[Int,Principal,
      Principal,String] )
message registerLogin[]( landingActor: Actor, loggedInPrinc:
      Principal, nameOfPrinc: String )
message lookupPrinc[]( sender: Actor, nameOfPrinc: String )
message foundPrinc[]( owner: Principal, ownersActor: Actor )
message princNotFound[]()
```

Figure 9: Message and Role Definitions

```
//actor for binding names to principals
actor Directory [ x ; inrole( sys, CareProvider ) ]( sys:
      Principal ) representing sys {
  var dirList: List[Tuple[Int,Actor,Principal,String]] = list()
  var idCounter: Int = 0 ; var found: Bool = false
  //Receive notification of a login
  on registerLogin[]( landingActor: Actor, princ: Principal, name:
       String ) {
    found = false
    print( "Directory received notification of a login by " + name
      )
    foreach record in (dirList) {
      if( (record.get(3)) == princ ) {
        found = true
        print( "Found record " + record )
    }}
    if( !found ) { //then first login
      dirList = tuple(idCounter,landingActor,princ,name ) ::
      dirList
      idCounter = idCounter + 1
  }}
  on lookupPrinc[]( sender: Actor, nameOfPrinc: String ) {
    found = false
    foreach record in (dirList) {
      if( (record.get(4)) == nameOfPrinc ) {
        send<sender>(foundPrinc(record.get(3),record.get(2)))
        found = true
    }}
    if( !found ) { send<sender>( princNotFound() ) }
}}
```

Figure 10: Directory Definition

```
actor Authorizer [ auth ; inrole( sys, CareProvider ) ](dir:
      Directory, sys: Principal) representing sys {
  var requests: List[Tuple[Actor,Principal,Tuple[Int,Principal,
      Principal,String]]] = list()
  var requestor: Actor ; var requestingPrinc: Principal
  var originator: Principal ; var subjectPrinc: Principal
  var subjectRecord: Tuple[Int,Principal,Principal,String]
  var data: String
  on requestAuthRelease[]( sender: Actor, requestingPrinc:
      Principal, subject: Principal, record: Tuple[Int,Principal,
      Principal,String] ) {
    originator = record.get(3)
    data = record.get(4)
    if( (!isPHI( data )) || ((requestingPrinc == originator) || (
      requestingPrinc == subject)) ) {
      send<sender>(authRelease(this,sys,record))
    } else {
      ltlquery(once receive<from PatientTerminal as P[Patient]> (
      authOwnRecordFor) with owner, releaseTo where owner = subject
       and requestingPrinc = releaseTo ) {
        send<sender>(authRelease(this,sys,record))
      } else {
        requests = tuple(sender,subject,record)
        send<dir>(lookupPrinc(this,subject))
  }}}
  on foundPrinc[]( owner: Principal, ownersActor: Actor ) {
    foreach req in ( requests ) {
      if( ((req.get(3)).get(3)) == owner ) {
        requestingPrinc = req.get(2)
        subjectRecord = req.get(3)
        send<ownersActor>(decideAuth( this, requestingPrinc,
      subjectRecord ))
  }}}
  on authOwnRecordFor[]( owner: Principal, releaseTo: Principal )
      {
    foreach req in (requests) {
      subjectPrinc = (req.get(3)).get(3)
      if( subjectPrinc == owner ) {
        requestor = req.get(1)
        subjectRecord = req.get(3)
        ltlquery( once receive<from Patient as P[Patient]> (
      requestAuthorizeRelease) with subject where subject = owner )
       {
          send<requestor>(authRelease(this,sys,subjectRecord))
        } else {
          print("preemptive auth by: "+owner)
}}}}}
```

Figure 11: Authorizer Definition