Provable Enforcement of HIPAA-Compliant Release of Medical Records Using the History Aware Programming Language

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

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

Armando Rodriguez
University of Texas at San Antonio
armando.rodriguez@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

Dependence on reliable information systems to safeguard personally identifiable information implies a need for privacy policies which guide the release and management of such information, whose mismanaged disclosure can be damaging to both the subject and the organization that releases it. Enforcing such policies requires attention to detail and care, and thus any aid that a compiler can render may be of value. We present a demonstration of compiler enforcement of privacy policy by implementation of the History Aware Programming Language (HAPL) framework. This framework allows expression of arbitrary HAPL code for actors in an actor system to be used to back a web application. This code is then checked for compliance with privacy policies described in assume-guarantee form before being assembled into a functioning application. The framework is demonstrated by implementing five use cases based on scenarios described in the Health Insurance Portability and Accountability Act (HIPAA), and the performance of the framework is tested.

CCS CONCEPTS

•Social and professional topics → Privacy policies; •Software and its engineering → *Domain specific languages;* **•Applied computing →** *Health care information systems;*

KEYWORDS

privacy policy; domain specific language; static analysis; scala; web application

ACM Reference format:
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 SACMAT'17, June 21–23, 2017, Indianapolis, IN, USA,* , 8 pages.
DOI: http://dx.doi.org/10.1145/3078861.3084176

*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/10.1145/3078861.3084176

1 INTRODUCTION

Society is increasingly dependent on computer information systems for management of private data. Examples of such applications include medical, banking, and identification systems. Organizations are required to keep and share such information in a manner that conforms to specific privacy policies as mandated by custom, sound business practice, good citizenship, contract, and often by law. Privacy policies such as those decsribed in the Health Insurance Portability and Accountability Act (HIPAA) [5], the Gramm-Leach-Bliley Act (GLBA) [2], and the Children's Online Privacy Protection Act (COPPA) [1], all carry the force of law. In addition to legal regulations, organizations typically have their own business rules that add further privacy requirements. It is therefore desirable, where possible, to enforce compliance with such privacy policies.

We implement the HAPL design [10] as 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. We briefly descibe how policy specifications are prepared for use in the framework that take the form of formal assume-guarantee specifications [8].

The design of HAPL in our previous work [10] allows the specification of arbitrary programs and which supports a novel construct, the *ltlquery* for inspecting and asserting over the history of messages received by individual actors in these programs. The language also supports definition of actors and user interface elements for interacting with these actors. As designed, HAPL also contains constructs for defining roles (*inrole*). Static analysis techniques are then applied to this language to verify that the implementation matches assume-guarantee specifications. In this paper, we implement the language as part of a framework that can produce functioning web applications.

Any program that passes this static analysis phase will then be compiled into a functioning web application backed by an Akka actor system to service core functionality and by LiftWeb[1] to run a user interface and to mediate requests between the application layer and the backing actor system. Five use cases (see section 5) are implemented in the language which constitute a representative subset of HIPAA requirements. These use cases pass static analysis and are composed into a single functioning prototype EMRS web application.

Organization. Section 2 introduces foundational concepts and technologies that will be referred to in this paper. We describe the

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

architecture of the HAPL framework in section 3. We outline the different components of the framework as implemented in section 4, and detail aspects of framework implementation in section 5. In section 6, we give examples of code and systems that make use of the HAPL framework, and go into some depth in discussion of some of our use cases. We conclude with section 7.

2 FOUNDATIONS

The technology of HAPL is strongly rooted in Scala and related technologies. Actors are essential to our assume-guarantee based approach to specification, and we make use of Scala's Akka library to generate and manipulate these actors. Further, LiftWeb is an actor-based web framework which we use to allow interaction with systems generated in HAPL. Specification is handled through assume-guarantee decomposition techniques. All of these together are required for our approach to the design and implementation of the HAPL language and its attendant framework.

Scala / LiftWeb. LiftWeb is an actor-oriented framework written in Scala for developing web applications. Comet actors supply dynamic behavior for statically served templates, which communicate with actors by passing messages. The LiftWeb library contains many functions for producing correct and compatible JavaScript, taking advantage of Scala's ability to present library functions in the style of an internal DSL. LiftWeb was selected because of HAPL's actor-based approach, and operates by way of forwarding events to programmer-defined handler. The programmer may then respond to those events in a variety of ways, including, if desired, updates to be pushed to the client. These updates may include HTML rewriting, and in this way it is possible to update a client's view of the webpage in response to events that occur on the server.

Actors. In the actor paradigm, computation is done through creation of and communication between actors. Actors are constrained to synchronous control only of private data, and to communicate with other actors only by sending asynchronous messages. Actor behavior must only be triggered in response to messages. An actor, while executing in response to a message, may only modify local data, create other actors, and send messages to other actors [4, 10]. For our specific purposes, we make use of the Akka library in Scala, which provides a convenient foundation for specification of their behavior. We use Akka to track actors that are created by our actor assembler, and to handle message passing between actors.

Assume-guarantee Decomposition. The HAPL EMRS is decomposed into a set of actor components described in first-order linear temporal logic (FOTL), each of which has distinct responsibilities that it upholds by communicating with the other actors. In short, the App actor provides an interface into the EMRS to human principals, the Archive actor manages stored records and fields requests for those records, and the Authorizer actor makes policy decisions on how the Archive should manage records when requested. Additionally, the Archive and Authorizer actors are decomposed from a higher-level component called the Backend which models their combined behavior [6].

In general, assume-guarantee specifications formally specify the behavior of an open system. The way a system interacts with its environment can be specified through assumptions about 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 [8][7].

3 ARCHITECTURE

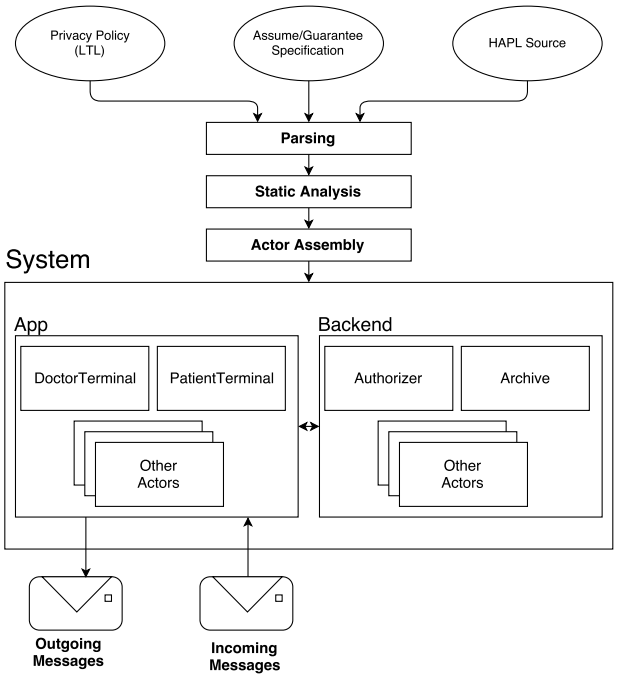

Figure 1: High-level architecture of HAPL

Figure 1 shows the high-level architecture of the HAPL framework. Broadly, a manually generated privacy policy expressed in linear temporal logic (LTL), an assume-guarantee specification, and HAPL source describing a program are inputs of the parsing and static analysis components of the framework [3, 6]. Here, the source is to be parsed, type-checked, and statically analyzed for compliance with the privacy policy and assume-guarantee specification.

This compliance is guaranteed by a static analysis phase requiring that the HAPL implementation honors the specification. Failing provability of compliance, the static analysis phase will return with errors explaining the failure.

The resulting vetted abstract syntax tree (AST) is then used to construct a runtime system in the form of a web application, described by its HAPL source. This AST is then fed to an Actor

Assembler, which produces the System. This System tracks instantiated actors through Akka. It also has an App component which is to respond to incoming messages and to reply with outgoing messages, as handled by the application's PageActor and PageInterpreter modules (see section 5). For the purposes of the present work, this App component serves the role of user interface, while incoming and outgoing messages represent user interactions mediated by the PageInterpreter service. Further, there is a Backend component, which handles bootstrapping of the EMRS as well as directory services, storage and authorization of records and their release.

3.1 Design Goals

The framework is designed with several use cases in mind and must fulfill certain requirements. Namely, the framework is required to:

Produce an actor system. The framework, given valid source, must result in an actor system that is capable of dynamically executing behavior specified by the HAPL system at runtime in response to messages. This must include the ability to instantiate new actors, send messages, and modify local actor state. Further, any system events that are of concern to the HAPL program must be properly captured and forwarded to the actor system as messages that can be responded to with behavior defined in HAPL.

Support history queries. The framework language must support methods for statically verifying compliance with assume-guarantee specifications. Since these assume-guarantees pre-suppose assertions on message histories, it is useful to expose the message history in the form of history queries. These history queries, if they are part of the language, can then be used in the static analysis phase to verify certain aspects of the assume-guarantee specification.

Support user interaction. The framework must support user interaction. In our design, this interaction takes place as part of a web application. The framework must therefore support the production of a web application that is specified by HAPL source. The framework must also ensure that the application is able to send and receive messages to and from the generated actor system. This must be done in such a way as to make possible meaningful user interaction with that actor system. Hence, it is necessary to ensure that the language is able to express user interface concerns.

Require provable compliance. The framework must ensure that a useful subset of programs are approved by a static analysis component. However, since the privacy policy we aim to enforce, HIPAA, is law, this is contingent on the constraint that no non-compliant programs are approved.

3.2 Design

The main components of the framework are the parser, the static analyser, the actor assembler, and the page interpreter. HAPL sources are fed into the parser, and if parsing passes then the resulting AST is given to the static analyser which decides acceptance of the program. This AST is used to assemble Akka actors at boot-time for the web application. The sections of the AST that describe the behavior of pages in the web application are kept in memory and

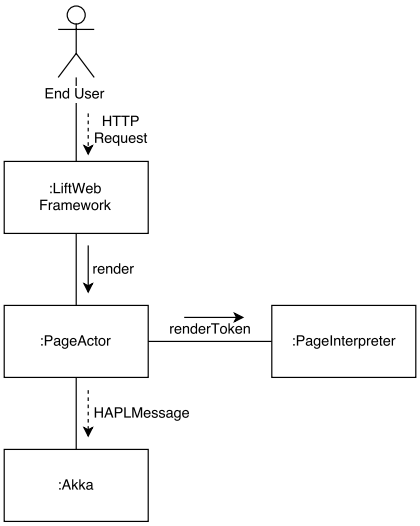

Figure 2: Map of Communications Between Runtime Entities

dynamically interpreted when requests for pages are received by the LiftWeb framework.

As can be seen in figure 2, these requests may result in control messages which need to be forwarded to the Akka actor system. At this stage, communications between actors are decided by the behaviors defined in HAPL sources. This behavior may result in updates that must be pushed to the user, and in this case, messages will be forwarded back to the page interpreter, which then will forward messages to the LiftWeb framework. The LiftWeb framework in turn has facilities for pushing updates to the end user.

4 FRAMEWORK

To implement this architecture, we have built a framework which we refer to collectively as HAPL. This framework contains the tools required to parse programs written in the HAPL langauge itself, statically analyse them for compliance, and assemble a web application for end-user interaction from these sources where compliance with our specification of HIPAA can be proven.

The HAPL framework supports a general purpose, actor-based programming language that incorporates history queries. HAPL is split into a user-interface specification, an actor behavior specification, a static analysis phase, an actor assembly phase, and a runtime ui interpretation system. In general, our framework is aimed at the implementation of five specific use cases.

4.1 Language and Parsing

The framework's language, loosely modeled on Scala syntax, allows for defining actors and the principals they represent, as well as in what role those actors represent these principals.

HAPL is an imperative language founded on the actor paradigm, which makes use of message passing as specified by our assume-guarantees. HAPL actors are considered to be acting in a particular role for a particular principal at the time they receive a message,

and when they send a message [4, 10]. In our implementation, we require that every actor represent a specific principal in a particular role.

This 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 10 and 11. Example actor definitions can be seen in section 6. Parsing is accomplished by use of the Scala parser combinators library[2].

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.

4.2 Static Analysis

In order to statically verify the behavior of our prototype as specified by their HAPL source code, we first design and detail specifications in section 2 for the EMRS that is implemented. The EMRS is decomposed into a set of communicating actors (figure 2) whose behavior is specified as a set of FOTL formulas in assume-guarantee form.

Once we have our EMRS specifications in place, we can verify that the system behaves correctly, and that it is HIPAA-compliant. We accomplish this by encoding our assume-guarantee specifications in our static analysis phase [9].

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[7, 9].

4.3 Actor Assembly

In order to construct a functioning actor system, it is necessary to decide the behavior of each actor in accordance with the parsed program that describes it. In general, all dynamic actor behavior is triggered in response to a message. We can therefore view actors having a state, a routine for initializing that state implied by declarations with initialization, and a map of message names to the executable actions associated with those names. It is thus necessary to determine what variables are in scope for the actor, to produce an executable routine that correctly initializes that actor's state in the order expressed in the language, and then to produce some way of instructing the runtime to execute the action described in HAPL code while making the state available for read and write during the execution of that action.

This process takes place at compile time, receiving the abstract syntax tree that results from parsing and producing a set of factories associated with each actor's name as described in its source. These factories will produce an actor with behavior described by its source upon invocation. Factories may be invoked as the result of actions that are executed at runtime. This method allows the servicing of *new* commands.

To bootstrap the actor system, the actor assembler invokes the factory corresponding to the name "Main," runs its initialization routine, and sends it the *execute* system signal (see section 4.5). At this point, the assembled system is now a functioning runtime.

To assemble the factories, a list of all actor definitions in the form of AST nodes is generated. A list of messages is then extracted from each actor definition node, and sent to a subsystem of the actor assembler that produces commands, where a command is here defined as a Scala function value that takes an actor's state and returns nothing, but which may have side effects. Each command is itself composed of commands, and in this way we can recursively descend through the AST representation of a message handler. Commands may include sends, state updates, foreach loops, or any of the various structures permitted by the grammar [9]. In general, wherever a state is read or written, this is accomplished by closing over the name of the identifier to be read or written, and that name is resolved to a value when the command is supplied with the appropriate state at execution time.

This method of using function values closed over the names of identifiers is useful for prototyping, although it comes with a performance penalty (see section 7).

4.4 Page Interpreter

The user-facing components are layered on top of LiftWeb, whose bootstrap sequence been modified to include an interpretation process. This interpretation process results in an AST which is stored in memory on the server until it becomes necessary to service a request.

A special Scala class, PageInterpreter, interprets the Scala native AST data structure on page request. With some pre-processing to separate out the pages, the PageInterpreter is notified of a page request, looks up the appropriate subtree of the AST corresponding to that request, and interprets that subtree to dynamically construct the page HTML in the process.

This HTML, once generated, is pushed to the client, which renders as can be seen in figure 3. It contains JavaScript that will send notifications to the LiftWeb framework whenever certain events are triggered. These events can be caught, and in response to form submission events in particular, the HAPL framework can be notified that user input has been entered. Handling these notifications is the responsibility of the PageActor described in section 5.

4.5 System Bindings

While implementing the actor assembler and attempting to integrate it with the page interpretation layer, it became clear that capturing certain system events and allowing the programmer to define behavior in response to those events was desirable. In order

[2]http://www.scala-lang.org/files/archive/api/2.11.4/scala-parser-combinators

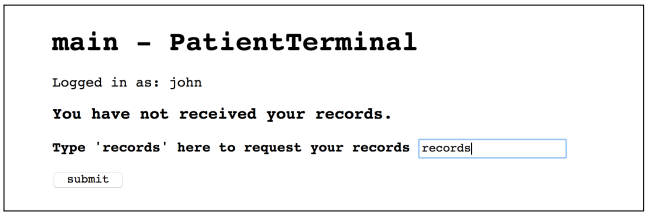

Figure 3: Patient interface

to honor the requirement that all actor behavior should be defined as actions in response to messages, we implemented special system-generated messages, which we call system signals, that are sent to specific actors when certain events are detected by the HAPL framework. Actors can choose to define behavior to capture those signals and react to them, or simply ignore them.

While there are other signals in the framework, one example is *execute*. This signal was added because of a need that became apparent in early testing of the actor assembler. We found that some bootstrapping mechanism, analagous to a main function or method as in C or Java, was necessary. The *execute* system signal satisfies this need by providing a place that is guaranteed to be called exactly once by the system when static analysis completes.

Another example of a system signal is *login*, which is sent to the Main actor whenever a new user logs in, which is useful for directory services. It could also be useful for preparing other forms of state required for each user that may not necessarily be appropriate in actors that are more tightly coupled to that user.

5 IMPLEMENTATION

To demonstrate the language and evaluate our framework, we built a prototype EMRS [9] 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 of personal information that are required, releases that are permitted without authorization, 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[3] [5], we designed the following five use cases in order to demonstrate a responsive non-trivial EMRS with respect to the release of personal health information (PHI) under HIPAA.

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. See section 6.
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. See section 6.

As part of the framework, HAPL source is compiled into a runtime that is run by the LiftWeb framework to generate a web-based user interface. This interface interacts with actors via page definitions controlling the specific actors handling the application logic of a user interface. An example of the interface for a physician is shown in figure 5. In this example interface, the physician is able to set PHI for a patient (including psychotherapy note PHI) or retrieve a patient's records. Such a page can be generated from source like that found in figure 6.

Figure 6 describes an early form of the main page for presenting an interface to a patient in the system. The page header, identifiable from the *page* keyword, is declared to belong to Patient, is identified as the main page for that role, and describes an underlying actor called PatientTerminal.

In this example, the function of the page as described by its source is rather simple. Specifically, it will, when the PatientTerminal actor's local variable 'recordsReceived' evaluates to false, show a form with exactly one field that will be annotated with the phrase "Type 'records' to get your records". This field will be populated with the value of the variable 'request', and will store the contents of that field into the field 'result' when submit is pressed. The described actor, 'PatientTerminal' will then be sent an update message, notifying it that its local variables have been updated by the page, and it can then take action in an 'update' message block. Use of 'main' here as the page identifier denotes that this page definition is the main page for this role, and so is the page served to anyone authenticated to this role when no other page is specifically requested.

In any case, updates to the described actor's state may result in messages sent from this block which might, for example, request the patient's records from the Archive. In this way, we can see how a user request can be translated into a request to the Archive.

In the implementation of requests to the Archive, it is clearly necessary to have different principals and the actors that represent them occasionally communicate with each other. To facilitate this, we have implemented language elements (*send* and actor methods) that allow the definition, sending, and receipt of messages. Further, via *ltlquery* keywords, it is also possible to inspect, assert on, and reason about the history of messages seen by a specific actor.

We can see a representative summary of the interactions between actors defined in HAPL source, the HAPL framework, and the

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

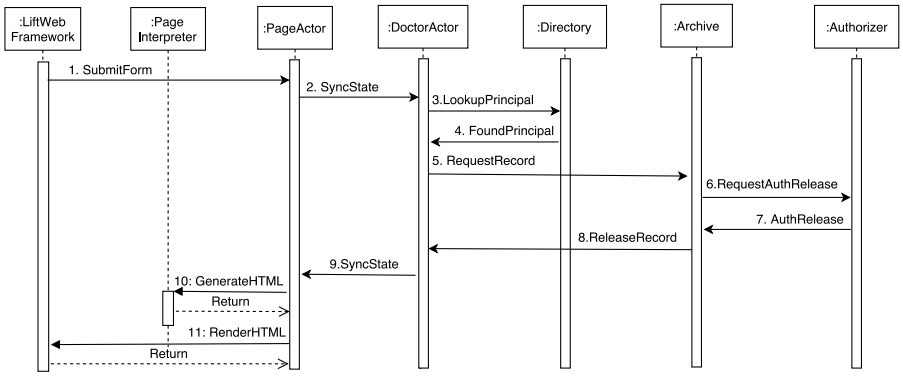

Figure 4: Lifecycle of a doctor requesting a patient's records

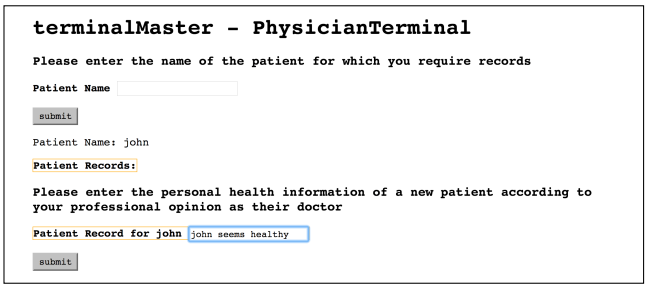

Figure 5: Physician interface

```
page Patient main describes PatientTerminal {
  show( "Logged in as: " + patient )
  if( !recordsReceived ) {
    form( "You have not received your records." ) {
      field( request, "Type 'records' to get your records",
      request )
  }}
  if( !authed ) {
    form( "You have not authorized the release of your records." )
       {
      field( request, "Type 'authorize' to authorize records
      release", request )
  }}
  show( status )
}
```

Figure 6: Page definition source

LiftWeb framework in figure 4. There, we can see that a user request begins with a message sent by LiftWeb to the HAPL framework, to be handled by PageActor. PageActor will then take the updated variables from the page, notify the actor described by the current page which in this case is DoctorActor. DoctorActor will then, by virtue of internal logic defined in its HAPL source, look up the appropriate principal and request their records from the Archive. This request will be validated by the Authorizer, and in the case that it is approved will result in the DoctorActor sending a notification back to the PageInterpreter, which in turn will generate HTML for the web application. Note that although the Authorizer may approve the doctor's request, in the event that the doctor is the the originator of the record, permission from no principal will be required.

6 EMRS STRUCTURE AND FUNCTION

We now present a view of the function of the prototype EMRS in the context of how HAPL was used in its construction. There are five use cases, but we will focus here mainly on the implementation of the Backend. In particular, we will examine the Advertiser and the Authorizer the most detail.

6.1 Backend Overview

HAPL programs are in general bootstrapped from a main actor, given the name 'Main'. There, certain boot-strapping activities can be taken care of, such as ensuring that specific actors are instantiated and always available. Further, this Main actor is sent signals whenever certain system events are trapped. By this means, we are able to set up a directory and ensure that there is exactly one such directory. We can also, through the use of system signals, guarantee that the directory is notified whenever someone authenticates, and so serve its purpose of tracking all users that log in to the system, and which actors are associated with those users.

Also important to this process is that the Archive has a reference to the Authorizer as soon as it is instantiated. Thus it is necessary for the Main actor to create the Authorizer and Archive in sequence, and to retain and supply a reference to the Authorizer for the Archive's use. More generally, the Main actor can ensure that any Backend actors that need to be in communication with each other are able to do so.

Directory. The directory is necessary for verification purposes we test equality of principal-type values. While it is legal to hold a reference to a principal, there is no mechanism for creating one. Rather, principals are supplied to the actors described by the main pages for each role. It is the responsibility of that actor to make the rest of the system aware of the principal-type value it holds. Since it is necessary for some principals such as doctors to be aware of principals it does not know about a priori, such as patients, there needs to be some central repository of stored principal values.

The directory is this repository. It works by receiving notifications of login events forwarded to it by the main actor, and it stores

each principal with its username in a list of username / principal-value tuples. When a request is made to the Directory requesting the principal value for a particular username, which is assumed to be communicated out of band, it will return the principal value so that records that are stored by principal may be retrieved.

This Directory service was not included in our initial design, but the expressivity of the HAPL language allowed us to solve this problem of principal binding within the language.

User-serving actors. Actor definitions have a signature with a *representing* keyword and a signature describing what types of value are required to create an instance of that definition. Within the actor, state variables may be declared, initializations defined, and message responses may be described.

The keyword *representing* denotes the principal value the actor is acting on behalf of. This has a number of practical effects including requiring the actor declaration to contain that principal value, and ensuring that the variable is in scope in the actor. Further, the keyword *inrole* is a type hint that requires both that some principal be described in the actor's signature, and that the principal so described is in the role supplied to the *inrole* as an argument. While all actors are required to be in some role, this is particularly important and intuitive for those actors that serve users directly, such as the PatientTerminal, the PhysicianTerminal and the Advertiser.

6.2 Advertiser

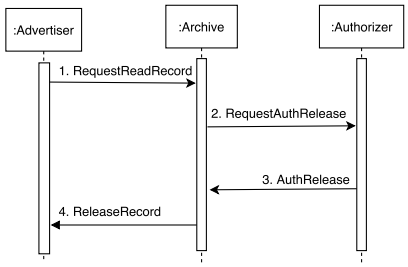

Figure 7: Advertiser requests patient records

In the AdvertisingHandler definition in figure 10, we can see these keywords at play as well. There, we see that AdvertisingHandler must represent a principal (advertiser) in the role of Marketing.

AdvertisingHandler is an example of a user-serving actor. It is a simple actor backing the Advertiser interface page, shown in figure 8 and defined in figure 9. This page definition will show a form with a single field, populated with the value of the request variable defined in the AdvertisingHandler definition in figure 9.

As we can see in figure 7, this request, once submitted, may be replied to by the Archive if the Authorizer (see section 6.3) is able to verify that permission for the release of those records is granted. If records are received, then the status will be updated, and this will result in a push that will ultimately update the user of the advertiser page.

6.3 Authorizer

The Authorizer, as its name suggests, is responsible for deciding whether or not to authorize the release of certain records. This

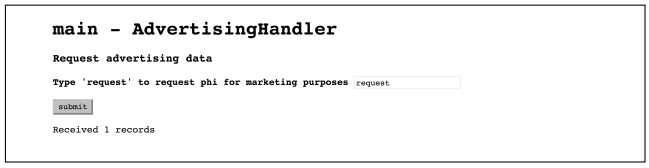

Figure 8: Advertiser interface

```
page Advertiser main describes AdvertisingHandler{
    form("Request advertising data") {
        field(request, "Type 'request' to request phi for
  marketing purposes", request)
    }
    show(status)
}
```

Figure 9: Advertiser Page Definition

```
actor AdvertisingHandler [ advertiser ; inrole( advertiser,
  Marketing ) ](advertiser: Principal) representing advertiser
  {
    var records: List[String] = list() ; var sentOnce: Bool =
  false
    var status: String = "" ; var received: Int = 0
    var request: String = "" ; var archive: Archive
    on notifyServices[](arch: Archive, dir: Directory, auth:
  Authorizer) {
        archive = arch
    }
    on update[]() {
        print( "vars updated!" )
        if( !sentOnce && request == "request" ) {
            send<archive>(requestReadRecord(this,
  advertiser, 1))
            send<archive>(requestReadRecord(this,
  advertiser, 2))
            send<archive>(requestReadRecord(this,
  advertiser, 3))
            send<archive>(requestReadRecord(this,
  advertiser, 4))
            sentOnce = true
            status = "Request sent"
    }}
    on releaseRecord[]( subject: Principal, content: String )
  {
        print( "Receveied a record: " + content )
        received = received + 1
        status = ("Received " + received) + " records"
        records = content :: records
        updateUI()
}}
```

Figure 10: Advertiser Definition

is accomplished by relying on several *ltlqueries*. Inuitively, the Authorizer definition seen in figure 11 requires that any release request must be preceeded by a grant of permission to release those records. Other ways that the data could be released are that the requestor must either be the originator of the data, or if the data is not PHI. If these paths fail, then the Authorizer will locate the subject and ask that subject to decide whether or not to authorize the request for their PHI.

In figure 11 we see the *inrole* statement requiring that the principal system be in the role of HealthCareProvider. This principal is also the first argument in the actor's signature, and is the principal that the Authorizer will be representing. Through type-checking,

we can guarantee that the principal represented by the actor is present at creation time, and will be in the role of HealthCare-Provider.

7 CONCLUSION

In this paper we have presented a framework which in conjunction with assume-guarantee decomposition [7] can implement verifiable information systems. We have developed the design of HAPL into a full framework and created an EMRS prototype in an actor component architecture which when assume-guarantee specifications are appropriately developed, allow it to statically verify and enforce the HIPAA privacy policy. Though this is a proof of concept, we believe that this approach to privacy policy enforcement shows promise.

While performance evaluations of HAPL code executing generally showed an order of magnitude slowdown over native Scala, we believe this to be largely due to executing closures within actors, instead of generating native or java bytecode. We therefore expect that the most significant performance gains would be made by revising the generated runtime.

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.

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, these issues would be addressed by implementing the more generalized approach described in our other published works [7].

```
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

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] 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.

[4] 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

[5] 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).

[6] 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.

[7] 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 22nd ACM Symposium on Access Control Models and Technologies* (2017), 12 pages.

[8] 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

[9] 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.

[10] 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.