A Model-Based Framework for Security Configuration Analysis

Eunsuk Kang and Daniel Jackson
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA USA
{eskang, dnj}@mit.edu

***Abstract*—Misconfiguration is one of the common causes of security failures today. Properly configuring a complex system, however, is a tedious and error-prone process, especially for users without domain expertise. The system may contain multiple components that must be configured independently, but interact with each other in subtle ways, leading to undesirable behaviors.**

This paper describes a framework for modeling and analyzing security properties of computer systems with complex configuration structures. A key idea behind our approach is a method for specifying modular descriptions of individual component configuration and composing them for global analysis. We applied our framework to construct a configuration analysis tool for the Apache web server, and demonstrated that our approach can be used to detect security vulnerabilities in the configuration of real-world web sites.

I. Introduction

Misconfiguration is one of the common causes of security failures today [1]. An increasing number of systems, such as web servers, social networks, databases, and cloud computing infrastructures require complex configuration tasks by the user. Properly configuring a system, however, is a tedious and error-prone task, especially for users without expertise in the problem domain. A configuration language can be complex and poorly documented, and understanding the effects of configuration on the system requires considerable effort on the user's part. Furthermore, a complex system typically contains *multiple, heterogeneous* components, each with its own configuration language. These components interact with each other in subtle ways; even after configuring all components, the user must still reason about their interactions to ensure that the overall system behaves the way the user intended.

We propose a framework for modeling and analyzing security properties of systems with complex configuration structures. A key idea behind our approach is a method for specifying and composing two different types of description: (1) a high-level system model that describes component interactions, and (2) descriptions of individual component behaviors, which are determined by the underlying configuration logic. By composing these specifications, the framework can be used to reason about the *overall* system behavior, and detect misconfigurations that would normally be considered harmless if the components were analyzed only in isolation.

Our framework provides separation of concerns among three distinct tasks that are, in practice, performed often by a single person in an ad hoc manner: (1) collection and representation of domain knowledge (best carried out by a domain expert), (2) specification of security properties (an end user), and (3) analysis (a test engineer). Once the expert constructs formal models of a system, they can be used by *multiple* users without understanding the details of the underlying models.

As a case study, we used our framework to construct a configuration analysis tool for the Apache HTTP Server [2]. We ran an experiment where we applied the tool to analyze the configuration of existing web sites. We were able to identify a number of security vulnerabilities in the sites, including inadvertent exposure of sensitive data. The outcome from our case study has prompted the owners of the sites to fix their configurations and eliminate the vulnerabilities.

In summary, the contributions of this paper include:

- A framework for building a security analysis tool for systems with complex configuration structures,
- A specification method for composing descriptions of individual component configuration with a high-level system interaction model,
- A method for encoding our specifications in Alloy [3], and checking a concrete configuration against a security property using the Alloy Analyzer, and
- A case study applying our framework to analyze the configuration of web sites on an Apache web server.

The rest of the paper is organized as follows. We begin by describing challenges to analyzing systems with complex configuration structures (Section II). After outlining the major elements of our framework (Section III), we describe our proposed specification method (Section IV), an encoding in Alloy (Section V), and a technique for checking configurations against security properties (Section VI). We describe a case study on the Apache web server (Section VII) and related work (Section VIII). We conclude with a discussion of our approach (Section IX).

II. Motivation

A. Example: Securing a Web Site on Apache

The Apache HTTP Server [2] is a popular web server program used to host millions of sites around the world. Its popularity is in part thanks to the flexibility of its configuration engine; with a wide range of available options, Apache can be

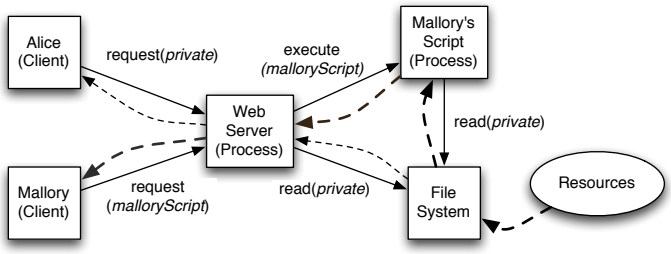

Fig. 1. An example diagram illustrating a potential security vulnerability in a web server system. Boxes represent modules, and circles represent data. A solid arrow from one module to another indicates interaction through an operation, and a dotted arrow represents data flow between two modules. Dotted arrows in bold correspond to a series of data flow that allows Mallory to access Alice's private resources.

configured to meet the needs of many server-side applications. On the other hand, properly configuring Apache is a notoriously challenging task, as hinted by numerous online articles and books on this topic alone [4], [5].

Consider a university computing environment that provides personal accounts and various software services to the members of the institute (e.g., students, faculty, or staff). Each user can set up a personal web site by placing web-related files into a designated directory (usually named *public_html*). Fig.1 illustrates high-level interactions between the components in the system. The file system stores a set of resources, some of which will be served by the web server. An OS process (e.g., an application or a script), running under a certain OS user's credentials, accesses these resources by performing *read* operations on the file system. A client sends a request to the web server by providing a URI for a resource. The web server, if it deems the request to be valid, retrieves the resource from the file system and returns it to the client.

Alice is teaching a course in introductory computer science, and creates a web site to publish course-related materials (*www.mit.edu/alice/cs101*). She wants to be able to share some of the sensitive materials (such as assignment grades) only with her teaching assistants. By default, Apache grants all browser requests, so Alice must configure the web site properly to ensure that the sensitive files can be accessed only by her TAs and herself. To achieve this, Alice decides to use a password authentication mechanism in Apache. She places the sensitive files in directory *cs101/materials/private*, and configures the directory with a password that she gives out to her TAs. Given this setting, the web server grants a browser request to any file under directory *private* only when the correct password is provided along with the request.

Mallory, a student in Alice's course, browses the course web site out of curiosity and discovers that directory *private* is password-protected. Despite not knowing the password, he manages to access all files in the directory by exploiting the fact that the web server is configured to host multiple sites. The exploit is as follows: He first logs onto his university account and creates a simple script that executes file system commands to list and read the contents of directory */users/alice/public_html/cs101/materials/private*. Normally, if he executes this script on his account, the commands would fail, because OS user *mallory* does not have sufficient permissions to access the files that are owned by user *alice* (given that she has correctly set up her file permissions). Instead, Mallory places the script under his *public_html* and executes the script on the web server by sending a browser request for the script. The script now runs under the credentials of the web server (typically user *www* in UNIX); since *www* has permissions to access any files that it serves, Mallory's script also gains the ability to access Alice's web files, including those under directory *private*.

This exploit, a form of privilege escalation, is an example of a security failure that arises from *interactions* between multiple components. Although Alice believes that she has properly configured her site, she fails to consider potential interactions between the web server, a script, and the file system, which together lead to a violation of her security intent.

B. Challenges

Given this example, we outline some of the challenges that arise when reasoning about the security of a system with complex, multi-component configurations:

a) System-Wide Reasoning: Many security failures originate not within a single component, but from unexpected interactions among multiple components. In addition to ensuring that each component is configured properly, we must analyze their combined behaviors. This task requires domain knowledge about the configuration of each component as well as interactions among components. Most users do not have this knowledge and face challenges during configuration.

b) Modularity in Composition: As a component evolves over time, its behaviors and the underlying configuration language may also change. An ad hoc approach to building a configuration analysis tool can result in a maintenance burden, since a change in a configuration language may cause changes to the entire tool. Furthermore, parts of the configuration may need to be *pluggable*; for example, the Apache web server can be deployed on multiple operating systems with different types of file systems. Thus, we must be able to support modular composition of multiple component configuration descriptions.

c) Automation: Manually reasoning about interactions between components is a tedious and error-prone process. In addition, some of the information about the system may be unknown. For example, a part of the configuration might be unspecified or inaccessible; it is difficult to pre-determine the set of input values from the environment (e.g., malicious client requests) that could lead to a security failure. Thus, we must be able to perform an analysis to detect potential security failures, given only *partial* information about the system.

III. Overview of the Proposed Framework

Fig.2 contains a high-level overview of our proposed framework for security configuration analysis. Two different kinds of users interact with our framework: *domain experts* and *end users*. A domain expert has extensive knowledge about (1) the high-level system structure and *components interactions*,

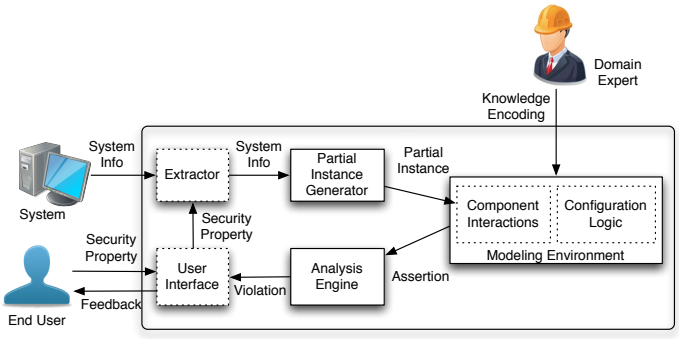

Fig. 2. Overview of our framework for security configuration analysis. Boxes with solid edges represent generic parts of the framework, and dotted boxes represent domain-specific parts that the expert builds for a problem domain.

and (2) the underlying *configuration logic* and its effects on the security of each component. The expert(s) encodes this knowledge in formal domain models.

Our end user, who may be a developer, a system integrator, or an ordinary computer user, faces the task of configuring a system to conform to some high-level intents about the security of data. Note that the user does not interact directly with the existing domain knowledge; one of the goals of the framework is to shield the user from the underlying details of the system. The user interacts with the system only through an *user interface*, which can be used to specify his or her intents as a *security property*. The interface then communicates to the *extractor*, which gathers system information that is necessary for analyzing the property, such as current configuration parameters and relevant data.

The *partial instance generator* takes the concrete system information, as well as the user-specified property, and encodes them as part of the domain model. The underlying analysis engine then checks the following assertion: *Does the system, as currently configured, satisfy the user's desired property?* If it determines that the system violates the property, it produces a *violation* as a counterexample illustrating a potential security attack on the system and displays it to the user.

IV. Specification Method

The main requirement of our framework is to enable reasoning about the overall security of a system, given the configurations of individual components. To achieve this, we integrate the following types of descriptions: (1) a high-level model of component interactions, and (2) the effects of configuration on the behavior of a component. The key idea is to model each component as an *access control module* that restricts how other modules may access data through an interface that it exports, and specify its access control behavior as *a function of its configuration*. In this section, we use a first-order relational logic as an underlying formalism, and refer to the Apache web server from Section II as a running example.

A. System Structure and Component Interactions

In our formulation, a system S consists of sets of *modules* M and *data elements* D. Modules interact with each other by performing *operations* O. Each module exports a set of operations (represented by relation $exports \subseteq M \times O$) to be invoked by another module that it interacts with ($interacts \subseteq M \times M$). Each operation is performed on a data element, and can be used to transfer the data from one module to another. We use relation $invokes \subseteq M \times O \times D$ to represent instances of operations, where $(m, o, d) \in invokes$ if and only if module m successfully invokes operation o on data d. In other words, if m never invokes o, or attempts to perform o on d but fails, then tuple (m, o, d) is excluded from $invokes$. A module can invoke only the operations that are exported by modules that it interacts with.

Each module accesses a set of data ($accesses \subseteq M \times D$). A module m can access a piece of data only if m is the owner of the data ($owns \subseteq M \times D$), or m receives the data from another module by invoking an operation.

Example Recall the Apache web server example from Section II. The system consists of the following modules: the web server, the file system, clients, and scripts. The file system owns a set of resources ($Resource \subseteq D$), some subset of which are to be served by the web server to its clients. A client sends a request for a resource $r \in Resource$ to the server by invoking operation $request \in O$ on r. Given a request, the server locates file f that corresponds to r and accesses its content through the file system's $read$ operation on f. After the server successfully retrieves f, it delivers its content back to the client. An implication of our description is that a client can access a resource only by invoking operation $request$; formally, the following constraint is imposed on the system:

$$\forall c \in Client, r \in Resource \cdot \\ (c, r) \in accesses \implies (c, request, r) \in invokes$$

Similar constraints are imposed on other components to establish data flow throughout the entire system.

B. Access Control Model

We augment the system description with an access control model that describes how each module restricts access to the data that it owns or received from another module.

Each module can control an operation $o \in O$ that it exports, by restricting the set of other modules that may invoke o or data that may be provided as an argument to o. We represent this by assigning an *access control list* to every module. Let relation $acl_{m_1} \subseteq M \times O \times D$ represent the access control list for module m_1, where $(m_2, o, d) \in acl_{m_1}$ if and only if m_2 is allowed to invoke operation o on data d. An access control list restricts relation $invokes$ in the following way:

$$\forall m_1, m_2 \in M, o \in O, d \in D \cdot \\ (m_1, o) \in exports \wedge (m_2, o, d) \in invokes \implies \\ (m_2, o, d) \in acl_{m_1}$$

In other words, m_2 can successfully invoke an operation on m_1 only if m_1 allows it.

C. Configuration Logic

The access control behavior of a module may not be fixed, and vary depending on how the module is currently configured. We model configurable behaviors by assigning each module a function that determines the content of its access control list, given a particular configuration.

Let C represent a set of *configuration objects*. A configuration object is an entity that represents a particular set of configuration values (e.g., a set of file permissions or an Apache configuration file). Then, $config_m : C \rightarrow \mathcal{P}(M \times O \times D)$ for module m is a *configuration function* that accepts a configuration object and computes a relation that corresponds to the access control list of that module (i.e. acl_m). In other words, this function is a partial specification that describes the security policy of the module; it specifies the effects of configuration on the access control behavior of the module. A configuration function is declaratively specified using constraints, instead of explicitly listing all of the tuples in an ACL.

Example Defining a configuration function requires an expert's knowledge about how different parameters affect the behavior of a module. Returning to our running example, let us first consider the configuration of the file system. Assuming that it is a standard UNIX-based file system, it uses file permission bits (e.g. *rwx* bits) to determine whether a certain user is allowed to invoke an operation on a particular file. Then, a configuration object for the file system is a list of permission bits for all files. The configuration function $config_{fs}$ for the file system is straightforward to define. Let $FileOp \subseteq O$ be the set of file operations, $Proc \subseteq M$ the set of OS processes, and $File \subseteq D$ the set of files. Then, the tuple $(p, o, f) \in Proc \times FileOp \times File$ is a member of $config_{fs}(ps)$ for some list of permissions $ps \in C$ if and only if ps lists the OS credential of process p as belonging to the group of users with the permission to invoke o on f.

The configuration function for the web server is more complex to define. Instead of being a flat list of parameters, a configuration object for Apache is a tree-based structure with two different types of parameters: *global* parameters, which apply to every resource in the server, and *local* parameters, which apply only to resources within a particular directory. To add to the complexity, these parameters may interact with each other in subtle ways (such as overriding). This gap between the high-level model of the server's behavior and the low-level configuration parameters is one of the factors that make Apache difficult to configure.

D. Security Property

Our framework allows an end user to specify a security property that restricts the set of modules in the system that should be able to access a particular set of data. Formally, a property is a formula that evaluates to true if and only if every module m that accesses a certain data set $D' \subseteq D$ satisfies a particular predicate *cond*:

$$\forall m \in M, d \in D' \cdot (m, d) \in accesses \implies cond(m, d)$$

Predicate $cond$ describes conditions that m must satisfy in order to be granted access to d. The definition of $cond$ is domain-specific; the expert may construct different $cond$ predicates to be used by the user to specify his or her intent.

The problem of checking whether the system in a particular configuration $cs = \{c_{m_1}, c_{m_2}, ..., c_{m_k}\}$ (for k modules) satisfies a property p can be formally stated as follows:

$$(\forall m \in M \cdot acl_m = config_m(c_m)) \implies p$$

That is, if every module in the system behaves as configured, then the property must hold true.

Example In the Apache web server, a desirable property might be defined as follows:

$$\forall c \in Client, r \in Resource \cdot$$
$$(c, r) \in accesses \implies \neg(ip(c) \in blacklist(r))$$

where $ip(c)$ represents the IP address of a client, and $blacklist(r)$ the list of IP addresses that are blacklisted for resource r. This property requires that every module that accesses a private resource is not blacklisted.

V. Encoding in Alloy

Alloy is a modeling language based on first-order relational logic with transitive closure [3]. Alloy is suitable for encoding our proposed specifications because: (1) its underlying relational logic is expressive enough to model the types of complex structures that arise in our specifications, (2) it has built-in constructs, such as signatures and subtypes, that are useful for building modular descriptions of configurations, and (3) it supports automated analysis for checking assertions and consistency of models. However, our approach does not prescribe the use of a particular formalism, and other languages that satisfy the above three criteria may be suitable.

A. Achieving Modular Specifications

Fig.3 shows snippets from an Alloy encoding of the specifications for Apache. The model in Fig.3(a) declares the basic *domain-independent* constructs for specifications that we introduced in Section IV. A domain expert will further extend this model with problem-specific models. The Alloy keyword *sig* (short for *signature*) declares a set of elements, and a *signature field* introduces a relation whose leftmost column is indexed on the signature. For example, in Fig.3(a), *sig Module* declares a set of modules, and *exports* introduces a binary relation of type $Module \times Op$. An *abstract* signature cannot be instantiated, and must be extended by other signatures.

Alloy supports model reuse by allowing a set of statements to be packaged inside a *module* (not to be confused with signature *Module*, which represents system components) and imported from other Alloy modules. When Alloy module $a1$ imports $a2$, it can refer to all of the declarations and constraints in $a2$. Fig.3(b) shows declarations for the model of a generic web server system, which imports $Base$. The keyword *extend* defines a subtyping relationship between two signatures; for instance, *Process*, *FileSystem*, and *Client* are declared as disjoint subtypes of *Module*. The field $runsAs$ denotes a

```
1  module Base
2  /* Domain−independent constructs */
3  abstract sig Module {
4    interacts : set Module,
5    exports : set Op,
6    invokes : Op −> Data,
7    accesses : set Data,
8    owns : set Data,
9    acl : ACL,
10   config : ConfigObj −> lone ACL
11 }
12 abstract sig Data {}
13 abstract sig Op {}
14 abstract sig ConfigObj {}
15 abstract sig ACL {
16   tuples : Module −> Op −> Data
17 }
```

(a) Basic definitions for system modules and configuration.

```
1  module GenericWebServerSystem
2  import Base
3
4  abstract sig User {}
5  abstract sig Process extends Module { runsAs : User }
6  abstract sig WebServer, Script extends Process {}
7  abstract sig FileSystem extends Module {}
8  abstract sig Addr {}
9  abstract sig Client extends Module { addr : Addr }
10 /* Component−specific configuration objects */
11 abstract sig ServerConfig, FsysConfig, ScriptConfig extends
       ConfigObj {}
12 /* Server operation */
13 abstract sig ServerOp extends Op {}
14 sig Req extends ServerOp {}
15 /* Web server resources */
16 abstract sig Resources extends Data {}
17 abstract sig File extends Resources {}
18 abstract sig Dir extends Resource { contains : set Resource }
```

(b) A model of a generic web server system.

```
1  module ApacheWebServer
2  import GenericWebServerSystem
3
4  one sig ApacheServer extends WebServer {}
5  sig ApacheConfig extends ServerConfig {
6    global : GlobalSettings, /* global settings */
7    local : Dir −> lone LocalSettings /* local settings */
8  }
9
10 abstract sig Settings { directives : set Directive }
11 sig GlobalSettings, LocalSettings extends Settings {}
12 abstract sig Directive {}
13 sig DirectoryDirective extends Directive {
14   target : Dir,
15   directives : set Directive
16 }
17 sig Allow, Deny extends Directive { addrs : set Addr }
18
19 /* Definition of the configuration function for Apache */
20 fact ApacheConfigDefn {
21   all cobj : ApacheConfig |
22     let acl = ApacheServer.config[cobj] |
23       all c : Client, q : Req, r : Resource |
24         c −> q −> r in acl implies
25           validReq[cobj, c, q, r]
26 }
27 /* True iff request "q" is considered valid by the server */
28 pred validReq[cobj : ApacheConfig, c : Client, q : Req, r : Resource] {
29   let ds = relevantDirectives[cobj, r] |
30     checkAllowDeny[c, allow[ds], deny[ds]]
31     checkDirectoryListing[c, index[ds]]
32     checkAuthentication[c, auth[ds]]
33     ... /* other constraints */
34 }
35 /* True iff client "c" is allowed by the server */
36 pred checkAllowDeny[c : Client, allow : Allow, deny : Deny] {
37     c.addr in allow.addrs or
38     c.addr not in deny.addrs
39 }
```

(c) A model of an Apache web server.

Fig. 3. Snippets from the Alloy encoding of the Apache web server.

process's user credential (line 5), and is inherited by both $WebServer$ and $Script$. Beside the signature declarations, the model also contains constraints that describe the relationships between modules ($interacts$, $exports$, etc.)[1].

The subytping mechanism in Alloy, along with its import system, is useful for achieving modularity in specifications. For instance, we may augment a generic description of a file system with a model of a standard UNIX-based file system, which specifies how the UNIX file permissions determine its ACL; we can independently construct a model of a different kind of file system (for example, NTFS), in a separate Alloy module, with its own permission language. This allows us to add models of additional component configurations without having to modify the existing collection of models, as long as the overall structure of the system remains the same. Furthermore, given models of different component variants, the configuration tool can select which models to import for analysis, depending on the environment on which the tool is running.

[1]Due to limited space, we omit these constraints and other details. The full version of the Alloy models is available at: http://people.csail.mit.edu/eskang/apache/models

B. Specifying a Configuration Function

The declarative nature of Alloy is suitable for specifying a configuration function as a set of constraints on an ACL. Fig.3(c) shows a part of the model for the Apache web server. Understanding Apache configuration is a complex task partly due to its distributed nature. Instead of relying on a single, global configuration file (usually named *httpd.conf*), each directory in the web server can be given its own configuration (*.htaccess*); to model these, we introduce *GlobalSettings* and *LocalSettings* (line 11), each containing a set of configuration parameters called *directives*. Apache provides a large number of directives, some of which impact the security behavior of the web server. For example, the user can use a *Deny* directive to specify a set of client addresses that should not be allowed to access a resource. Some of the directives can be nested within another; a *Directory* directive allows the user to specify a set of directives that should apply only to the resources under a specific directory *target* (line 14).

Recall that a configuration function takes a configuration object, and computes a set of tuples that represent entries in an ACL. In our Alloy encoding, each module is assigned a signature field *config* that represents the configuration function

(Fig.3(a), line 10). One way to define the function is by using an Alloy *fact* to declaratively specify the relation between an input configuration object and its corresponding output ACL. For example, fact *ApacheConfgDefn* defines the configuration function for Apache in the following way (line 20): If a tuple *(c, q, r)* belongs to the ACL resulting from a particular configuration object $cobj$, then the tuple must satisfy the set of conditions in *validReq*. Predicate *validReq* itself is complex, and decomposed into a set of helper predicates. Let *relevantDirectives* be a function that examines the configuration structure and extracts the set of directives that directly affect resource r. Predicate *checkAllowDeny* evaluates to true if and only if client c belongs to the *Allow* list for resource r, or does not belong to the *Deny* list. The client must satisfy additional predicates in *validReq* in order to be granted access to the resource.

The configuration function for a file system or a script can be defined similarly as a part of a separate Alloy module.

C. Specifying a Property

We express a security property as a predicate that is parameterized with user-specified input. For example, the following property states that only the clients who are not on blacklist bl are allowed to access a set of private resources $privateRs$:

```
pred noBlacklistAccess [privateRs : set Resource, bl : set Addr] {
  all c : Client, r : privateRs |
    r in c.accesses implies r.addr not in bl }
```

An Alloy *assertion* is used to express a desired property about the model. Let us first introduce a predicate *applyConfig*, which constrains the behavior of a module with respect to a particular configuration:

```
pred applyConfig[m : Module, c : ConfigObj] { m.acl = m.config[c] }
```

We then write the following assertion, stating that if the system is configured as specified by the user, the property must hold:

```
assert ConfigSatisfiesBlacklistProperty {
  /* MyApacheConfig, MyPermissions, MyPrivateResources, and
       MyBlacklist are user−provided parameters */
    applyConfig[ApacheWebServer, MyApacheConfig] and
    applyConfig[UNIXFileSystem, MyPermissions]
  implies
    noBlacklistAccess[MyPrivateResources, MyBlacklist] }
```

The next section describes how we encode the user's configurations and property parameters as a *partial instance* in Alloy.

VI. Analysis

We are interested in the following analysis problem: Does the system, as currently configured, satisfy a given property? Before the analysis, we first perform a set of pre-processing steps, where we extract the current configuration settings and concrete data from the system and encode them as a *partial instance* in the Alloy model. We additionally apply an abstraction technique to reduce the size of the analysis problem. The analysis engine then explores all possible interactions between modules and attempts to find a potential scenario in which the system violates the property. When the analysis completes, we display the outcome of the analysis as feedback to the user.

A. Background on the Alloy Analyzer

The Alloy Analyzer is a built-in analysis engine for Alloy [3]. The analyzer itself relies on Kodkod [6], a SAT-based constraint solver for first-order relational logic. Given a problem, the Alloy Analyzer translates the input Alloy model into a set of relational constraints in Kodkod, which, in turn, translates them into a SAT problem. If a satisfying instance is found, the Alloy Analyzer maps it back to an instance in the original Alloy model, where elements of the signatures are bound to concrete atoms in a way that satisfies all of the relational constraints in the model.

Given an assertion P and a set of constraints C over relations R that describe the system, the Alloy Analyzer attempts to find a satisfying instance to formula $C \wedge \neg P$. Since our property takes the form:

$$\forall m \in M, d \in D' \cdot (m, d) \in accesses \implies cond(m, d)$$

when logically negated, it becomes:

$$\exists m \in M, d \in D' \cdot (m, d) \in accesses \wedge \neg cond(m, d)$$

A satisfying instance to $C \wedge \neg P$ contains witnesses to the quantified variables m and d, as well as bindings of tuples in all relations, including acl, $invokes$, and $accesses$, which together represent a particular snapshot of the system. This instance forms a counterexample that demonstrates an inappropriate access of data d by module m.

Our analysis problem involves checking a *concrete* system configuration, acting over a *concrete* set of data, against a particular property. Thus, our analysis differs from a typical constraint solving problem in that some of the variables have fixed values. More generally, a partial solution to a constraint solving problem is called a *partial instance* [6]. Kodkod, given a partial instance, attempts to reduce the size of the input constraint set (and consequently, the size of the resulting SAT formula) by partially evaluating the logical expressions.

Before the analysis in Alloy, the *partial instance generator* from Fig.2 perform pre-processing steps to combine concrete information from the user's system with the abstract Alloy model. The result of the pre-processing is a new Alloy model where parts of the original model have been concretized with the known information about the system.

B. Encoding Concrete Information as a Partial Instance

Consider the example from Section II. Alice's desired property states that only clients with her password can access resources under the directory $cs101/materials/private$. The analysis problem is to check whether the system satisfies this property, given the current site configurations. Suppose that directory $private$ has the structure shown in Fig.4; it contains a file and a directory, which in turn contains three additional files. We can encode this part of the file system topology as a partial instance by extending the original, abstract model as follows[2]:

[2]In Alloy, *one sig* declares a single concrete atom of a signature, and $+$ represents an union operator.

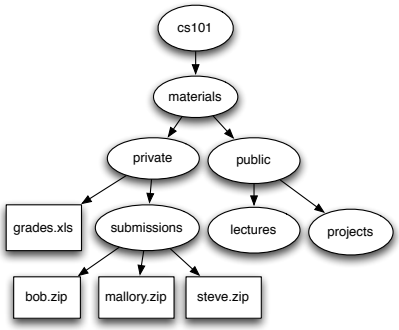

Fig. 4. A sample file system structure corresponding to Alice's course site. Circles represent directories, and boxes files; edges represent a directory-to-child relation.

```
/* Partial instance representing structure cs101/materials/private */
one sig private, submissions extends Dir {}
one sig grades_xls, bob_zip, mallory_zip, steve_zip extends File {}
fact FileSystemTopology {
  private.contains = submissions + grades_xls
  submissions.contains = bob_zip + mallory_zip + steve_zip }
```

Concrete configuration objects for the Apache server and the file permissions can also be encoded in the similar fashion.

Certain types of information about the system may not be known or obtainable. Examples include inputs from the environment (e.g. a malicious client's request that exploits a weak configuration) or part of system configuration that is missing or inaccessible. The analyzer will exhaustively explore all possible assignments to these unknown parts of the model, in order to generate a counterexample that demonstrates a potential violation of a property.

C. Finitization

Alloy's first-order logic is undecidable, and so in order to produce a SAT instance, the Alloy Analyzer bounds the size of each signature in the universe, as specified by the user. The set of objects that we encode as a partial instance provides bounds for some of the signatures in the model. For example, we know that there are four different elements of type *File* in the model from Fig.4. For other signatures that are not fully specified, such as *Client* or *Req*, the user must provide an upper bound for each of them. Due to this finitization, our analysis is *bounded*; if the analyzer fails to find a property violation, there might still exist a violation that involves a large number of objects beyond the given bounds. The user may increment the bounds and re-run the analysis to gain further confidence.

D. Domain-Specific Abstraction

The size of SAT problem that Kodkod generates depends on the number of data objects in the system that is being analyzed. In some cases, we can leverage domain-specific knowledge to eliminate objects that are irrelevant to a property and reduce the size of the partial instance that we encode in Alloy.

Our abstraction is based on the following observations. First, recognizing that a property is a constraint over access to a *particular* set of data, we can prune away the remaining data elements as being irrelevant to the analysis. For example, Alice's property concerns only those resources under directory *private*; thus, the resources including *public* and its children in Fig.4 can be excluded from the partial instance.

We can also observe that some of the data objects in the system may be considered *equivalent* with respect to their accessibility under a particular configuration. Based on this observation, we can apply a technique called *symmetry breaking* to reduce the number of objects to be analyzed [7]. For example, let us assume that all of the three files in *submissions* have the same UNIX file permissions and Apache directives that affect their behavior. If we find a violation that involves an inappropriate access to *bob.zip*, we can substitute any one of the other files for *bob.zip* and still expect the same violation to occur. Similarly, if the analysis fails to detect a violation involving *bob.zip*, we may expect that no violations exists that involves one of the other files. Then, before partial instance generation, instead of creating a *File* object for every file under *submissions*, we may pick *bob.zip* as a sole representative for the equivalent set of files.

Formally, given an original system S with data elements D, let $\alpha : \mathcal{P}(D) \rightarrow \mathcal{P}(D)$ be an abstraction function, and S' an abstract system with data $D' = \alpha(D)$. Function α is *sound* with respect to a property P if and only if, for any module configuration, S' satisfies P only when S satisfies P. In general, automatically constructing a sound abstraction function for Alloy's first-order logic is undecidable. However, the abstraction function and its soundness condition can be expressed and checked in Alloy [3, p.222], or proved using a theorem prover; once this has been done, the abstraction function can be reused for the same problem domain[3].

VII. Case Study: Apache Web Server Security

In this section, we present a case study where we used our framework to analyze the security of a real-world system with complex configuration structures. We have implemented the *Apache Configuration Analyzer*, a tool that automatically detects potential security vulnerabilities in the configuration of a web site that is hosted by the Apache HTTP server. We applied our tool to web sites hosted on Apache at the Computer Science and Artificial Intelligence Laboratory (CSAIL) at MIT, and identified vulnerabilities in several of them, all leading to inadvertent exposure of sensitive data to the web. We have notified the network administrators and the owners of these web sites about the security issues, resulting in configuration fixes and a major revamp in one of the sites.

The Apache Configuration Analyzer is freely available for download at http://people.csail.mit.edu/eskang/apache.

A. Implementation

The Apache Configuration Analyzer has the same high-level architecture as shown in Fig.2. The first step in the implementation was building formal models of the system components and configuration logic in Alloy. The primary

[3]Due to limited space, the full definition of an abstraction function that we used for our Apache case study is available at:
http://people.csail.mit.edu/eskang/apache/models/absfun

designer of the tool (the first author) had a good understanding of both Alloy and Apache, and spent 12 weeks building and validating the models; the size of the current version is approximately 2100 lines of Alloy in total. Even though the models took significant effort to build, the cost is amortized, since the tool can be used by multiple Apache users.

In one of the Alloy modules, we defined a set of desirable security properties about a web site. The front-end interface of the tool presents these properties to the user, who provides parameters to one or more properties to be analyzed. The following list shows 6 properties that the tool currently offers; items within the brackets represent parameters to properties:

- `P1(blacklist, rset)`: Clients on `blacklist` should never be able to access resources `rset`.
- `P2(whitelist, rset)`: Only clients on `whitelist` can access resources `rset`.
- `P3(dir)`: A client should never be able to access the listing of directory `dir` and all its sub-directories.
- `P4(rset)`: A client should never be able to access resources besides `rset`.
- `P5(p, rset)`: Only clients with password `p` can access resources `rset`.
- `P6(ca, rset)`: Only clients with a valid certificate issued by certificate authority `ca` can access resources `rset`.

For example, if the user wishes to check an instantiation of property `P1`, the user must provide a list of IP addresses or hostnames for `blacklist`, and a path to a directory whose children correspond to `rset`.

Once the user fully specifies a property, the interface passes it along to the extractor, which: (1) extracts settings from both global and local configuration files, and (2) determines the type of the underlying file system and executes appropriate OS commands to extract the permissions on relevant files. The extractor also applies an abstraction function to the web site objects (files and directories) and derives a smaller set of objects that are relevant for the property. Then, the partial instance generator builds an Alloy partial instance that encodes the concrete information along with an assertion for checking the property. When the Alloy Analyzer completes its analysis, the user interface displays the outcome (including a counterexample, if any) to the user[4].

The user interface, extractor, and partial instance generator were written in around 4000 lines of Python.

B. Experiment

To demonstrate that the Apache Configuration Analyzer can be used to find security vulnerabilities in the configuration of realistic web sites, we performed an experiment where we applied the tool to analyze sites that are hosted by the Apache web server at CSAIL. In particular, we selected web sites that contained sensitive information, such as course-related materials, faculty-student reviews, and private research repositories. Before performing the experiment, we received permissions from the lab's network administrators to access the server configuration and the content of the sites.

For each site, we specified the parameters to the above properties based on the security requirements of the site, and ran our tool to check whether the site configuration satisfied the properties. We checked only those properties that were relevant to each web site. For example, one of the sites contains a sub-directory called `private` that requires the client to own a valid MIT certificate for access; thus, we specified an instance of `P6` with parameter values `MIT certificate` for `ca` and `private` for `rset`. In total, we analyzed 10 websites and 24 properties. For the analysis in the Alloy Analyzer, we used a bound of 5 for every signature, except those that were specified in the partial instance.

C. Results

When we initially ran our experiment, our tool found the same type of violation as we described in Section II in all of the sites, due to a weakness in the global configuration that allows any CSAIL user to execute a script as *www* user. Unfortunately, the users do not have a way to eliminate this vulnerability by modifying their *.htaccess*. In order to detect vulnerabilities that can be fixed by the users, we modified the global configuration file to disallow this exploit, and re-ran our experiment on the sites.

The results from our final experiment are shown in Table I. Due to the sensitive nature of data that these web sites store, we do not show their names or URLs, and instead use numeric labels to refer to them (first column of the table). In most cases, we were able to abstract away a significant portion of the original objects, because (1) the property's parameter specifying resources to be protected often included only a subset of the site's objects, and (2) most files were equivalent with respect to configuration parameters that affected their behaviors (i.e. file permissions and Apache directives).

All of the analysis runs completed under 6 minutes. The analysis times were strongly correlated with the number of objects in the partial instance. Kodkod spends a significant portion of the translation time on evaluating the input relational expressions using the partial instance before converting them into a Boolean formula; thus, with an increase in the size of the partial instance, we observed an increase in translation time.

In total, the tool detected **9** violations (out of 24 properties analyzed) in **5** of the web sites that we analyzed. For every property violation found, the tool generated a counterexample with a potential client request that led to the violation. Since this request referred to a particular resource, we could replay the same request inside a browser, and observe the response from the server. We used this method to confirm that every counterexample that the tool detected was indeed a true vulnerability that could be exploited. We categorize the detected violations into three distinct kinds, based on the characteristics of the underlying misconfiguration:

Directory Listing Exposure The simplest form of misconfiguration that we found (Sites 1, 2, 4, and 10) was a failure

[4]The tool also has an ability to automatically generate a recommended configuration fix (if it exists) for a property violation. The discussion of the mechanism for fix generation is beyond the scope of this paper.

Site	Property	# Obj. Original	# Obj. Reduced	Extraction Time (s)	Translation Time (s)	Solving Time (s)	Total Time (s)	Violation Found
1	P3	954	58	8.721	56.996	2.006	67.723	Yes
	P4	954	89	12.129	270.01	31.486	313.625	Yes
2	P3	272	27	2.717	3.437	0.224	6.378	Yes
	P4	272	33	2.904	8.869	2.181	13.954	No
3	P3	88	21	1.412	1.895	0.690	3.997	No
	P6	88	21	1.412	1.970	0.782	4.164	No
4	P3	895	60	5.304	82.274	2.582	90.16	Yes
	P4	895	76	5.463	172.799	5.750	184.012	Yes
	P6	895	44	2.947	28.452	2.128	33.527	Yes
5	P3	55	17	1.32	1.112	0.361	2.793	No
	P6	55	17	1.32	1.255	0.463	3.038	No
6	P3	192	11	0.452	1.741	0.228	2.421	No
	P4	192	39	4.605	18.624	1.821	25.050	Yes
	P6	192	11	0.452	0.362	0.153	0.967	No
7	P3	364	35	9.068	11.652	3.010	23.730	No
	P4	364	40	8.522	19.931	4.868	33.321	No
	P5	364	35	9.068	11.592	3.241	23.901	No
8	P3	64	38	1.973	18.162	5.124	25.259	No
	P5	64	38	1.973	17.912	4.750	24.635	No
9	P1	338	27	1.958	18.070	1.807	21.835	No
	P3	338	82	9.073	211.899	6.695	227.667	No
10	P3	163	35	2.238	12.376	1.491	16.105	Yes
	P4	163	74	4.8465	177.124	17.597	199.568	Yes
	P6	163	31	1.668	7.189	2.685	11.542	No

TABLE I

RESULTS FROM THE ANALYSIS OF THE CSAIL WEB SITES. THE 3RD COLUMN CONTAINS THE ORIGINAL NUMBER OF OBJECTS IN THE SITE, AND THE 4TH COLUMN THE REDUCED NUMBER AFTER ABSTRACTION. THE FOLLOWING THREE COLUMNS CONTAIN: THE RUNNING TIME OF THE EXTRACTOR, THE TIME TO TRANSLATE AN ALLOY ASSERTION INTO SAT, AND THE RUNNING TIME OF THE SAT SOLVER (MINISAT 1.14 [8]), RESPECTIVELY.

.

to protect the content listing of a directory either by putting a default index file (e.g.,*index.html*) or disabling directory listing altogether using *.htaccess*. Sometimes a directory listing can reveal names of sensitive files; in Site 1, we were able to access a file that contained a list of user names and hashes of their passwords.

Weak Permissions In Sites 1, 4, 6, and 10, we found files that were located outside the web document root (i.e.*public_html*) but had their permissions as readable by the *www* user. These sites are susceptible to an exploit called *directory traversal*, which attempts to access resources outside the web root through simple manipulation of URLs. Exploiting the weak permission was simpler in one of the sites (Site 4); it had a symbolic link that was pointing directly to a *www*-readable directory outside the web root, and *.htaccess* was missing a directive that would have disabled symbolic links.

Misconfigured Authentication Site 4, a course site, had a set of private directories, protected with an MIT certificate, that were intended to be used by students to submit their assignments. The *.htaccess* file for one of the directories had an error, allowing anyone to access the files under the directory.

We have notified the network administrators and the site owners of the issues that we found, along with recommended fixes. Three of the site owners have applied the fixes; one of the sites is being rebuilt from scratch for improved security.

VIII. RELATED WORK

There is a large body of literature on automated configuration. We focus our discussion on works that address security aspects of configuration.

Margrave [9], [10] is a framework for specifying and analyzing policies. Like our framework, Margrave uses a first-order logic for policy representation, and Kodkod for analysis. However, Margrave is designed to assist users with the task of *policy authoring*, and provide an interface for them to interact directly with policy *rules*, which determine how input parameters map to desired effects on the system. In comparison, our system is designed to aid *end users*, who may not have a good understanding of configuration rules; in particular, our intention is to explicitly shield the user from the complexity of the rules by hiding them.

A number of other recent works use automated techniques, such as constraint solving or model checking, to analyze access control policies [11], [12], [13]. Similar to Margrave, these systems are used to reason about properties of a set of policies, and do not deal with arbitrary system configurations.

MulVal [14] is a framework that combines a variety of information such as vulnerability reports, policies, component interactions, and analyzes a network configuration for vulnerabilities. MulVal uses Datalog as the language for modeling domain knowledge. Datalog is less expressive than Alloy, and lacks quantifiers and subtyping features, which are useful for modeling complex configuration structures.

Our work was strongly influenced by Guttman and his colleagues' work on *rigorous automated security management* [15]. This approach involves an expert constructing formal models of various system components, and allowing a user without domain expertise to specify and check a high-level security goal. Ramakrishnan and Sekar use model

checking to analyze models of operating systems and detect potential vulnerabilities in configuration [16]. Their analysis is purely abstract, and does not check a *particular* configuration.

Two previous works use models to analyze the Apache configuration [17], [18]. Both involve building a formal model of the web server and using a constraint checker to analyze a concrete configuration. These approaches differ from ours in that: (1) instead of allowing the user to specify a property, they check the configuration against a fixed list of recommended settings, and (2) they do not model other parts of the system, such as the file system, clients, and scripts, and thus, cannot detect violations that arise from component interactions. Nikto [19] is a penetration testing tool that attempts to find flaws in Apache by sending a large number of server requests, but it does not look at the actual configuration for analysis.

Bonatti and his colleagues developed an approach for composing access control policies over multiple, heterogeneous components [20]. Their work is at the level of *policy language design*; it focuses on building complex policies out of smaller, independent sets of policies by using composition operators, without explicitly modeling interactions between components.

Feature models in software product lines [21] is an active area of research, with recent advances in techniques for validating and fixing feature configurations [22], [23], [24]. A feature model emphasizes dependencies between different features of a system. Our approach, in comparison, focuses on interactions between modules through operations and their impact on the access control behavior of the system.

Engage is a framework for managing application configurations and dependencies between them [25]. Similar to our approach, they use a high-level modeling language to describe interactions between different components, and a SAT solver to solve a set of configuration constraints. Their framework is used for generating installation plans. Kikuchi and Tsuchiya use Alloy to model constraints over system components and synthesize a valid deployment configuration [26]. Similarly, Narain proposes an approach to performing network configuration using Alloy [27]. These two works address security issues only tangentially.

IX. Conclusion

We proposed a framework for modeling and analyzing security properties of systems with complex configuration structures that arise from interactions between multiple, heterogeneous components. We demonstrated how our specification method can be used to compose descriptions of component configurations with the overall system model, and how these specifications can be encoded and analyzed using Alloy. We also showed that our framework can be used to detect security vulnerabilities in the configuration of a real-world system.

Our specification method is based on modeling *access control* behaviors, and not suitable for modeling other types of security requirements, such availability and information flow. However, we believe that our framework is capable of addressing an important class of security configuration tasks. We have successfully applied our framework to model and analyze other types of configuration problems, including privacy settings on a social network and secure router configuration.

In this paper, we focused on security vulnerabilities that arise from configurations. However, there are other aspects of the system that are just as critical to establishing its security. For example, the owner of a web site should complement the Apache Configuration Analyzer with other types of tools, such as static analysis or testing, to detect vulnerabilities in the web application code as well as the site configuration. In our research vision, this project is a step towards a generic framework for combining results from different types of analyses and reasoning about the overall security of the system.

References

[1] Open Web Application Security Project, "OWASP Top Ten Project," http://www.owasp.org/index.php/, 2012.

[2] Apache Foundation Software, "Apache HTTP Server," http://httpd.apache.org/, 2012.

[3] D. Jackson, *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.

[4] I. Ristic, *Apache security - the complete guide to securing your Apache web server*. O'Reilly, 2005.

[5] R. Barnett, *Preventing web attacks with Apache*. Addison-Wesley, 2006.

[6] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *TACAS*, 2007, pp. 632–647.

[7] I. Shlyakhter, "Generating effective symmetry-breaking predicates for search problems," *Electronic Notes in Discrete Mathematics*, vol. 9, pp. 19–35, 2001.

[8] N. En and N. Srensson, "The minisat page," http://minisat.se, 2012.

[9] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *ICSE*, 2005, pp. 196–205.

[10] T. Nelson, C. Barratt, D. Dougherty, K. Fisler, and S. Krishnamurthi, "The margrave tool for firewall analysis," in *USENIX LISA*, 2010, pp. 1–8.

[11] K. Jayaraman, V. Ganesh, M. V. Tripunitara, M. C. Rinard, and S. J. Chapin, "Automatic error finding in access-control policies," in *CCS*, 2011, pp. 163–174.

[12] J. Hwang, T. Xie, V. C. Hu, and M. Altunay, "Acpt: A tool for modeling and verifying access control policies," in *POLICY*, 2010, pp. 40–43.

[13] M. I. Gofman, R. Luo, A. C. Solomon, Y. Zhang, P. Yang, and S. D. Stoller, "Rbac-pat: A policy analysis tool for role based access control," in *TACAS*, 2009, pp. 46–49.

[14] X. Ou, S. Govindavajhala, and A. Appel, "Mulval: A logic-based network security analyzer," in *USENIX Security Symposium*, 2005, pp. 8–8.

[15] J. D. Guttman and A. L. Herzog, "Rigorous automated network security management," *Int. J. Inf. Sec.*, vol. 4, no. 1-2, pp. 29–48, 2005.

[16] C. R. Ramakrishnan and R. C. Sekar, "Model-based analysis of configuration vulnerabilities," *Journal of Computer Security*, vol. 10, no. 1/2, pp. 189–209, 2002.

[17] C. Sinz, A. Khosravizadeh, W. Küchlin, and V. Mihajlovski, "Verifying cim models of apache web-server configurations," in *QSIC*, 2003, pp. 290–297.

[18] D. Glasner and V. C. Sreedhar, "Configuration reasoning and ontology for web," in *IEEE SCC*, 2007, pp. 387–394.

[19] C. Sullo and D. Lodge, "Nikto2," http://cirt.net/nikto2/, 2012.

[20] P. A. Bonatti, S. C. Vimercati, and P. Samarati, "A modular approach to composing access control policies," in *CCS*, 2000, pp. 164–173.

[21] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (foda) feasibility study," CMU, Tech. Rep., 1990.

[22] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *ICSE*, 2012, pp. 58–68.

[23] M. Cordy, A. Classen, G. Perrouin, P.-Y. Schobbens, P. Heymans, and A. Legay, "Simulation-based abstractions for software product-line model checking," in *ICSE*, 2012, pp. 672–682.

[24] J. White, B. Dougherty, D. C. Schmidt, and D. Benavides, "Automated reasoning for multi-step feature model configuration problems," in *SPLC*, 2009, pp. 11–20.

[25] J. Fischer, R. Majumdar, and S. Esmaeilsabzali, "Engage: a deployment management system," in *PLDI*, 2012, pp. 263–274.
[26] S. Kikuchi and S. Tsuchiya, "Configuration procedure synthesis for complex systems using model finder," in *ICECCS*, 2010, pp. 95–104.
[27] S. Narain, "Network configuration management via model finding," in *LISA*, 2005, pp. 155–168.