Towards Automatic Reverse Engineering of Software Security Configuration

Rui Wang[1], XiaoFeng Wang[1], Kehuan Zhang[1,3],Zhuowei Li [2]

[1] School of Informatics, Indiana University at Bloomington.
[2]Center for Software Excellence, Microsoft.
[3]College of Computer and Communication, Hunan University.

Abstract

The specifications of an application's security configuration are crucial for understanding its security policies, which can be very helpful in security-related contexts such as misconfiguration detection. Such specifications, however, are often ill-documented, or even close because of the increasing use of graphic user interfaces to set program options. In this paper, we propose *ConfigRE*, a new technique for automatic reverse engineering of an application's access-control configurations. Our approach first partitions a configuration input into fields, and then identifies the semantic relations among these fields and the roles they play in enforcing an access control policy. Based upon such knowledge, ConfigRE automatically generates a specification language to describe the syntactic relations of these fields. The language can be converted into a scanner using standard parser generators for scanning configuration files and discovering the security policies specified in an application. We implemented ConfigRE in our research and evaluated it against real applications. The experiment results demonstrate the efficacy of our approach.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access

General Terms

Security

Keywords

Reverse Engineering, Configuration, Access Control, Taint Analysis, Context-Free Language

1. INTRODUCTION

Software security configuration describes the security policies an application enforces. Specifications for such configuration can be highly useful to many security-related utilizations. A prominent example is detection of *misconfigurations*, a major operator error with serious security implication [21, 24]. Misconfigurations can only be found by evaluating the security policies defined within applications, and identification of these policies relies on the information about how these applications are configured. Actually, existing configuration checkers such as the Microsoft Baseline Security Analyzer (MBSA) [3] detect common misconfigurations through parsing the configurations of operating systems and service programs. Knowledge of security configuration could also be used to circumvent software manufacturers' restrictions on use of their products. For example, one might alter the configuration of trial software to get the functionalities of the full edition.

Tools like MBSA are provided by software manufacturers, who have the knowledge about their products' configurations, in particular, the formats of configuration files. It becomes much more difficult to acquire the configuration information for the software developed by the third party. Applications may not come with well-documented specifications: for example, Linux programs usually have nothing but a few examples to explain their configuration options, and as a result, many settings can be left undocumented. The tendency to use graphic user interfaces for setting and revising configurations further reduces the need for publishing configuration file formats. Moreover, software manufacturers may have the intentions to hide some configuration settings from their customers, for the purposes of controlling the way in which their applications can be used.

In the absence of specifications, one may have to analyze an application to reverse engineer its security configuration. To avoid intensive human efforts during this process, automated tools need to be developed. As a first step towards this end, we propose *ConfigRE*, a novel technique that automatically analyzes an application's binary executables to generate a specification for its access-control configurations. A program's security configuration usually concerns access control, which describes whether a *subject* (e.g., a client) is permitted to access an *object* (e.g., a file) in a particular manner (e.g., 'read', 'write' or 'execute'). ConfigRE aims at automatically discovering the configuration syntaxes that define subjects, objects, and their permissions, which requires semantically identifying these correlated components from an application's execution. This is achieved in our approach through an instruction-level taint analysis that automatically examines how the information flow from configuration files affects the information flow from the client's request for accessing an object.

As an example, consider an HTTP server whose configuration defines a policy that allows the client from an IP address to read the files under a directory. ConfigRE first partitions the input from the server's configuration file into fields and then uses individual fields as taint sources to analyze a *transaction*, the process for serving a request. This analysis reveals the semantic relations among these fields, in particular, how a field affects the use of another one. It also detects the field that represents the subject, the client, from
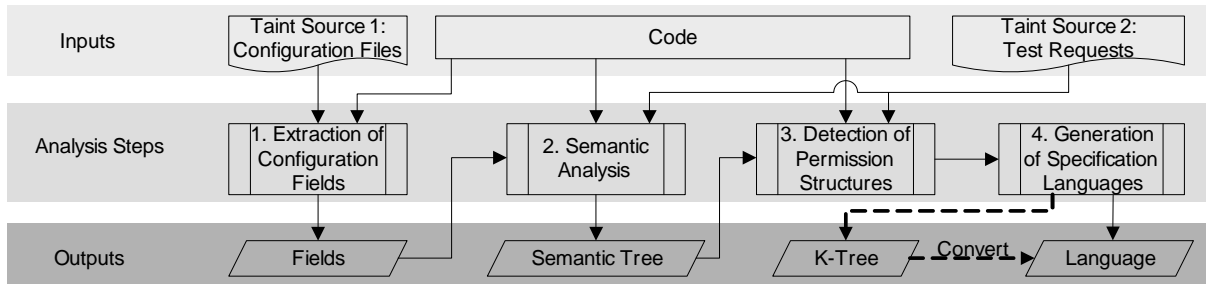
Figure 1: Analysis steps.

the operation that compares its IP with a value tainted by the configuration file, and the field for the object, the directory, from the `filename` parameter of the API call `fopen()` also tainted by the file. Suppose the permission is formatted as `Read = Yes`. The field that accommodates 'Yes' can be detected to be a permission field through identifying its legitimate values, 'Yes' and 'No', and rerunning the program on them to find out whether the success of the transaction actually depends on these values. These fields are correlated according to their semantic relations. Based upon such knowledge, our approach can automatically create a specification language, which can be converted to a scanner for discovering the security policies defined in the configuration files of other instances of the HTTP server.

At a high level, ConfigRE bears some similarity with recent work on automatic protocol reverse engineering [8, 18, 32], which aims at producing the specification for a closed network protocol. However, these approaches focus on parsing an application-level message into fields, whereas ConfigRE needs to semantically understand the influence of the configuration input on the processing of a transaction. Serving our objective are a suite of new techniques for automatically detecting semantic relations among different fields, and recognizing subjects, objects and the permissions that correlate them. These techniques were developed in our research and evaluated against real applications. They were demonstrated to be effective in generating configuration specifications.

We outline the contributions of this paper as follows:

- *Novel techniques for identifying access control configurations.* We propose a suite of new techniques for detecting the semantic relations among configuration fields that affect the processing of a transaction, and the roles they play in enforcing an application's access control policies. These techniques are built upon an innovative approach that takes a different look at the traditional taint analysis: not only did we study the control flows among configuration fields to identify their semantic relations, but we also examined how the information flows are tainted by two different sources, configuration files and a service request, and interact with each other, which reveals the key components of the policies such as subjects and objects. We also developed a technique that automatically discovers fields' alternative values and reruns the application on these values to detect permission structures and generate regular expressions to represent them.

- *Automatic generation of specification languages and scanners.* We present a technique that automatically generalizes the information regarding policy-related fields and their relations into a specification language. The language describes how to syntactically identify subjects, objects and the permission structures that bind them. It can be conveniently converted into a scanner through standard parser generators

for recognizing the security policies defined in an application's configuration files.

- *Implementation and evaluations.* We implemented a prototype system of ConfigRE and evaluated it using real applications, including HTTP servers, FTP servers and P2P software. Our experimental study demonstrates that ConfigRE can effectively generate configuration specifications related to access control and use them to detect security policies from an application's configuration files.

The rest of the paper is organized as follows. Section 2 gives an overview of the general design of ConfigRE. Section 3 elaborates the detailed design and implementation of our prototype system. Section 4 reports an empirical study of our technique using the prototype. Section 6 discusses the limitations of our current design. Section 5 presents the related prior research, and Section 7 concludes the paper.

2. OVERVIEW

The general idea of ConfigRE comes from the insight that how data are used by an application can actually reveal the semantic meanings of the data. For example, use of a character string as an input to the function `gethostbyname()` unambiguously points to the fact that the string actually represents a host name. Our approach utilizes this observation to identify security-related configuration data and their relations, and then converts such knowledge into a specification. In this section, we first present the key steps for this analysis, and then survey ConfigRE using an example. We also introduce *taint analysis*, a technique intensively used in our approach.

Analysis steps. ConfigRE takes four steps to reverse engineer an application's security configurations, as described in Figure 1. In the first step, it monitors the initialization process of the application to identify its configuration files, and extract the structure information of these files, in particular, configuration fields. Then, our approach performs an instruction-level taint analysis on the application using individual fields and a test request as taint sources. This analysis acquires the semantic information of the configuration structure, such as the semantic relations among the fields, the fields that define subjects and objects, and those that could be related to permissions. The possible permission fields are further studied in the third step, which detects their alternative values, and reruns the application over these values to generate the conditions for permitting or denying a request. The last step generalizes the knowledge about subjects, objects and permissions into a specification language.

An example. We use the example in Figure 2 to illustrate our approach. The example includes the code fragments of an HTTP server for parsing and initializing its configuration settings, and for

Taint Source 1:
Configuration Files
httpd .conf
...
C1 Directory "/usr/www/" {
C2   IPPrefix = 192.168.1.0/24
C3   FileAccess = Yes
C4 }
...

Taint Source 2:
Test Requests
GET /filename HTTP/1.1

Code

1. Initialization
...
I1 parseLine(line,&keyword,&value);
I2 if (keyword == "Directory") {
I3   directory = value;
I4   if (strchr(line, "{"))
I5   getRelatedField(file);
I6 }
...

```
void getRelatedField(CFG_FILE *file) {
I7  while(getline(file,line)&&!strchr(line,"}")){
I8    parseLine(line,&keyword,&value);
I9    if (keyword == "IPPrefix ")
I10     prefix = value;
I11   else if (keyword=="FileAccess") {
I12     if (value == "Yes")  fileAccess = 1;
I13     else if (value == "No")  fileAccess = 0;
I14     else ERROR("Invalid value");
I15   }
I16 }
I17 ...
```

2. Transaction
...
T1 if((ChkPrefix(prefix,outerAddr)
       == true) && (fileAccess == 1))
T2   file = ReadFile(directory +
         req ->filename);
...

Semantic Tree

K-tree

Language

Σ ={Directory, '{', '}', '\n', '=', IPPrefix, FileAccess, Yes, No, ε, subject, object}
V={$A_{00}, A_{10}, S_{10}, T_{10}, U_{10}, A_{20}, A_{21}, B_{21}, I$}
$A_{00} \rightarrow A_{10}$
$A_{10} \rightarrow S_{10} U_{10} T_{10}$
$S_{10} \rightarrow$ **Directory** object '{' '\n'
$T_{10} \rightarrow$ '}' '\n'
$U_{10} \rightarrow A_{20} I A_{21}$
$A_{20} \rightarrow$ **IPPrefix** '=' subject '\n'
$A_{21} \rightarrow$ **FileAccess** '=' B_{21} '\n'
$B_{21} \rightarrow$ **Yes** | **No**
$I \;\; \rightarrow$ '{' I | '}' I | '\n' I | '=' I | ε
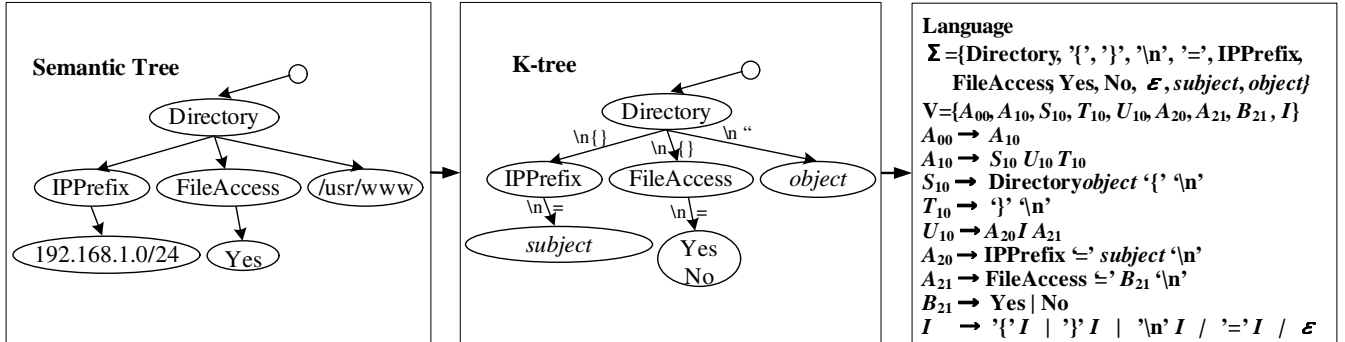
Figure 2: An Illustrative Example.

processing a service request it receives. We also present part of its configurations related to access control. The server has a security policy that only allows the client from '192.168.1.0/24' to read the files under the directory '/usr/www/'. This policy is first interpreted for initializing the configuration settings in memory (from Line I1 to Line I16), and then enforced during a transaction (from Line T1 and T2). The example uses C for illustration purpose. ConfigRE actually works on binary executables.

ConfigRE first analyzes the execution of the HTTP server until the point that it is ready to accept service requests. This analysis traces the propagation of the information tainted by the files the server read to discover configuration fields. For example, the configuration fragment in Figure 2 is partitioned into fields such as 'Directory', 'usr' and 'www'. It also identifies delimiters like '{' and '}' which describe the syntactic relations of fields.

To acquire semantic information, ConfigRE monitors the propagations of the information flows tainted by individual fields to study their semantic relations and influences on the processing of a test request. The relations among these fields are modeled according to the propagation of control flows between them, which are used to construct a *semantic tree* (Section 3.2) as illustrated in Figure 2. Each field is represented by a node in the tree, which determines the uses of its offspring nodes. For example, the instructions from Line I11 to Line I14 shows that the field 'FileAccess' must be matched before its child node 'Yes' can be processed. The analysis further detects the operation at Line T1 that compares '192.168.1.0/24' with the IP address of the request, and annotates this field as a subject. The fields 'usr' and 'www' are found to be appended with the content of a string variable req->filename that is tainted by the request, and used as the filename parameter for ReadFile(). As a result, they are combined and labeled as a directory-related object. We also observe that fileAccess (Line I12), a value tainted by the field 'Yes' through control flow, has been used as a branch condition (Line T1), which causes the field 'Yes' to be selected as a possible permission.

To verify whether the possible permission is indeed a permission,

ConfigRE re-examines the execution that initializes the server and identifies the alternative value of the field accommodating 'Yes', which is 'No', from Line I13. The value is used to modify the configuration file, over which the server is rerun to study its response to the same service request. This time the request does not go through, and therefore 'Yes' and 'No' are confirmed to be the alternative values for a permission field.

From the semantic tree, our approach extracts the fields annotated as subject, object and permission, along with all their ancestor nodes which serve as their context. After that, it identifies the delimiters that bind these fields together, for example, '{' and '}', to form a *K-tree* (Section 3.4). From the K-tree, ConfigRE automatically generates a specification language and converts it to a scanner for automatically discovering security policies from other instances of the HTTP server.

Taint analysis. A building block for ConfigRE is taint analysis, a technique that monitors the execution of an application at the instruction level to track its tainted information flows, including data flows and control flows. Tainted data flows are generated when an operation such as data transfer happens to tainted data. They are identified and followed according to a set of taint-propagation rules similar to those used in other taint-analysis techniques such as RIFLE [27], TaintCheck [23] and LIFT [25]. Tainted control flows describe how a tainted branch condition controls the executions of other instructions. All the instructions affected by the condition are considered to be within the *scope* of that condition and their outputs should be tainted. For example, the condition value == 'Yes' at Line I12 in Figure 2 determines the operation fileAccess = 1; as a result, the variable fileAccess is tainted by value. Here, the scope of the condition ranges from Line I12 to Line I15, where all branches originated from this condition converge. This location (I15) is called *post dominator* of the condition. Our analyzer uses Lengauer Tarjan [17] algorithm to compute the scope and the post dominator of a tainted condition, and taints the outputs of the instructions within that scope. A well known problem

of such an approach is *taint explosion* caused by excessive taint of the values that actually do not depend on the branch condition. In our research, we mitigate this problem using a conservative strategy, which taints the instructions within the scope according to a set of rules. For example, we allow taint to propagate through the control flow only when a branch condition is a comparison between a tainted value and a constant, and that condition is true.

3. DESIGN AND IMPLEMENTATION

In this section, we elaborate our design for individual analysis steps, and an implementation of the design to a prototype system under Linux. Our implementation is based on PIN, a dynamic binary instrumentation tool [19]. Standard parser generators such as FLEX [1] and BISON [13] were also used in our research to convert the specification ConfigRE generates into a configuration scanner.

Our current implementation is only semiautomatic and oriented towards text-based configuration files. Moreover, the type of polices we are dealing with mainly concerns a remote client's attempts to read or write local files. However, the idea behind our approach is more general, which we discuss in Section 6.

3.1 Extraction of Configuration Fields

To reverse engineer an application's configurations, we first need to extract its configuration structures. This is achieved through partitioning an input stream from configuration files into fields. It is important to note that we do not assume knowing these files *a priori*: we can monitor all the files the application reads during its initialization stage, remove Dynamic Linking Libraries and well-known system files and detect from the rest those that do contain configuration settings in the follow-up steps.

The problem of extracting fields from an input stream has been intensively studied recently under the scenario of protocol reverse engineering [8, 18, 32]. We do not intend to reinvent the wheel in our research, and instead, try to build ConfigRE upon the existing techniques. Specifically, we enhanced the technique proposed by Wondracek, et al [32] to handle some special configuration features, and implemented it into our prototype system as a module for field extraction. It should be noted here that our design can also accommodate other field-partition techniques.

Simple delimiters. ConfigRE automatically partitions an input stream into fields through analyzing the propagations of tainted information within an application. This is achieved by finding from the stream a set of *delimiters* that indicate the ends of individual fields. Wondracek, et al [32] propose an approach that detects delimiters from the way in which the application processes tainted data: a delimiter is usually used by the application to scan the input, and therefore can be recognized from the activity that consecutively compares a constant with a tainted string. For example, the HTTP server described in Figure 2 scans Line C1 for the new-line character '\n' until the end of the line and the space character ' ' before '{' is encountered, and '/' from the beginning of '/usr/www/' to its end; those characters are identified as delimiters. Using delimiters, ConfigRE divides the input into multiple fields such as 'Directory', 'usr', 'www' and '{'. The syntactic relations among these fields are also described by the delimiters: for example, '\n' binds all these fields together.

Paired delimiters. A feature of configuration format which has not been addressed by the prior approaches is *paired delimiters* such as '{' and '}'. These delimiters are important because they are widely used by applications to specify the syntactic relations of multiple fields. For example, '{' and '}' in Figure 2 link all the fields from Line C1 to Line C3. We also present a fragment

of Apache configuration in Figure 3, in which a multi-byte delimiter pair, '⟨Directory' and '⟨/Directory', correlates several lines of configuration settings to describe the access-control policy for the directory '/web/docs'. Here we propose a technique that detects such delimiters.

A prominent feature of paired delimiters is that a program cannot accept the existence of one of them without the other. This is usually achieved by scanning an input stream for the close part of the pair such as '}' *only after* the start part, '{', has been identified. Here, we call the latter *head* of the pair and the former *tail*. Based upon the above observation, we propose a new approach for detecting these delimiters. Our approach first constructs a list of "head" delimiters, including all the simple delimiters and the heads of paired delimiters. This list can be acquired by recording all the delimiters a program uses to compare with a non-delimiter byte from the input. Whenever a delimiter on the list has been matched, ConfigRE monitors the operations within the scope of that condition (matching the delimiter) to look for a *new* delimiter which does not appear on the head list. If such a delimiter is found, it is picked up as the tail for the delimiter in the condition. As an example, consider the code fragment in Figure 2 from Line I4 to Line I6: after the server identifies '{', it starts scanning for '}' within the scope of the tainted condition strchr(line, '{') at Line I4, which reveals that these two characters actually belong to a pair.

Although the above approach describes a pretty general *behavior signature* for the way that a program processes paired signatures, there are exceptions. As an example, consider the case described in Figure 3. Apache does not treat '⟨' as a normal delimiter and never conducts consecutive comparisons to find it. Actually, the character is combined with keywords such as Directory as part of a field. To pair '⟨Directory' with '⟨/Directory', Apache first identifies '⟨/', and then checks the closest field that begins with '⟨' for the string that follows '⟨/', which in the example is 'Directory'. ConfigRE detects such delimiters using another behavior signature: whenever a tainted branch condition becomes true, we check its scope for the comparison between two tainted strings from different fields; if such a comparison has been detected and turns out to be a match, these two strings are labeled as paired delimiters, together with the characters that taint the condition. In the Apache example, our approach discovers 'Directory' within the field '⟨Directory' and the field '⟨/Directory' as a pair[1]. Since comparisons rarely happen between two configuration fields, this signature does not introduce many false positives, as we observed in our research.

We have to stress here that these two behavior signatures by no means cover all possible ways an application can take to process paired delimiters. It is possible that they could allow some delimiters to slip under the radar, though all the applications we have studied so far do not have such a problem. Developing a more general alternative is left as our future work.

3.2 Semantic Analysis

The key for reverse engineering configurations is to understand the roles played by individual fields in enforcing a security policy, i.e., subjects, objects and permissions. Such an understanding must be gained under certain semantic context which links all these roles together. For example, in Figure 2, 'IPPrefix' and 'FileAccess' must be defined under 'Directory' to specify their relations with the directory '/usr/www'. This context infor-

---

[1]Note that missing the character '⟨' does not present a serious problem, as 'Directory' is identified in the context of '⟨Directory', which must be there for creating a language (Section 3.4).

Apache configuration file: httpd.conf

```
<Directory "/web/doc1">
  Options FollowSymLinks
  Order allow , deny
  Allow from 192.168.0.0/24
</Directory>
```

```
<Files "sensitive.htm">
  Order allow , deny
  Allow from 192.168.1.0/24
</Files>
```

```
<Directory "/web/doc2">
  Options FollowSymLinks
  Order deny , allow
  Deny from 192.168.2.0/24
</Directory>
```
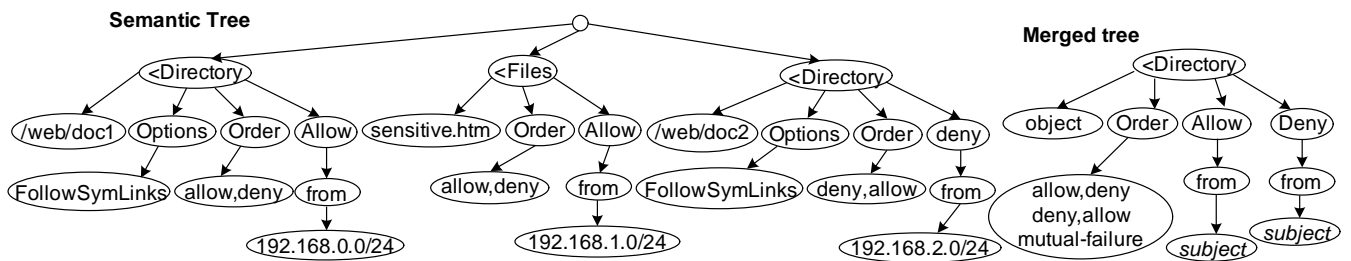

Figure 3: Another Example.

mation is modeled in our research as a *semantic tree*. Following we first describe how to build a semantic tree from an application, and then introduce the techniques that identify individual access-control components.

Semantic tree. Semantic tree is used to understand how a field controls the operations on another field, which unveils the semantic relations between these fields. A semantic tree can be formally modeled as a 2-tuple $\langle N, E \rangle$, where N is a set of nodes and E is a set of edges. Each node except the root represents a configuration field. An edge is extended from one node to another if an application uses the former to determine how to process the latter. The root is assumed to control every node that does not have a parent.

ConfigRE builds the semantic tree of an application through analyzing the propagation of tainted information flows. Specifically, it first marks each field as a different taint source, and connects all these fields directly to the root as its children. During the analysis, ConfigRE locates the comparison between a value tainted by a field and a constant. Such a comparison is called a *control condition*. Whenever a control condition related to a field n is found to be true, our approach checks whether other fields have been operated on within the scope of that condition. Once such a field n' is identified, ConfigRE makes n' an offspring node of n.

For example, consider the program in Figure 2. The instruction at Line I2 is a control condition that compares the field 'Directory' with a constant. Within the scope of the condition (Line I2 to I6), fields like '/usr/www/'[2], IPPrefix and FileAccess are all used for comparisons when the control condition is true. Therefore, we make these fields children nodes for 'Directory'. Moreover, the field 'Yes' appears in the scope of 'FileAccess', which causes an edge to be drawn from the latter to the former. The same happens to the relation between '192.168.1.0/24' and 'IPPrefix'. These relations are described by the semantic tree in Figure 2. Another example is presented in Figure 3, which gives the tree for a configuration fragment of Apache.

Detection of access-control components. ConfigRE detects access-control components through analyzing how an application enforces its security policy on a test request. The request is labeled as a new taint source for the taint analysis and studied against the information flows derived from configuration fields. Our approach observes the interactions among the information flows from different sources and uses a set of behavior signatures to recognize the

fields related to subjects, objects and permissions.

Subjects such as IP addresses and Internet domains can be discovered from the behavior that attempts to match a value tainted by a field to the source IP address of the request. If this attempt is successful, we are pretty sure that the field is either IP address or an Internet domain name. The latter can also be identified from the input parameter for the API calls such as `gethostbyname()`, which maps a host name to a structure containing its IP addresses. More difficult to handle is the field that specifies username. A username included in a request is usually checked by the application against a configuration record. What ConfigRE can observe from this process is a comparison that happens between a value tainted by the request and the other by a configuration field. Such a behavior can be too general to become a signature. Our solution is to make use of some recognizable information on the request. The payload of the request related to a username can be pre-labeled if the request comes through a public protocol, such as HTTP. In the case that the protocol is close, the location can still be found through a taint analysis on the client program which generates the request: we can track the data tainted by a username across the client until they are reflected on the payload of a network output. Whenever such data match a value tainted by a field after they are received by the application, we know that the field is for specifying a subject. Another problem for detecting subjects comes from the keywords that define special types of subjects such as *anyone*. An example is the keyword `All` used by Apache. These keywords may not be involved in any dataflow operations during a transaction and could act like permissions. We describe an approach to recognize them in Section 3.3.

Objects such as directory names and file names can be determined from the parameters of relevant API functions such as `fopen()` and `ReadFile()`. ConfigRE recognizes a directory field if the field taints a string that is appended with another string tainted by the request, and then used together as a parameter of an API call for specifying the directory path and the name of a file. Note that it is important to have the parameter tainted by the request, because otherwise the API call may not relate to the transaction at all: for example, it can be just an operation for opening a configuration file. Another behavior signature that characterizes an object is as follows: a string tainted by the request matches another string related to a field, and then the former is used as a file name or a directory for an API call. This behavior is also sufficient for concluding that the field is for defining an object. In some cases, multiple fields can be found to collectively define an object. An example is 'usr' and

---

[2]This field comes from a merger between 'usr' and 'www', as described later.

'www' in Figure 2. When this happens, we merge these fields into a single field and add back the delimiter that separates them.

The techniques for detecting subjects and objects are based upon dataflow analysis. This also works for some types of permissions. For example, once we notice that a directory field is always used to encapsulate a file name tainted by a request before the file can be opened, and an attempt to directly retrieve its parent directory[3] is blocked, we know that the subject is not allowed to access the files outside the directory. Such a permission is called *home-directory permission*. However, discovering other types of permissions requires more efforts than dataflow analysis. Specifically, though the taint analysis can find from the execution path a set of branch conditions tainted by configuration fields, we do not know whether these conditions affect the success of a transaction, as our analysis only observes one execution path. Therefore, the fields associated with these conditions can only be treated as *candidate permissions*, and a further study is required to determine whether they are indeed permissions. This is achieved in our research through identifying these fields' alternative values and rerunning the application over them, which are elaborated in Section 3.3.

ConfigRE is also designed to discover the configuration settings for multiple subject/object pairs. As an example, consider a request for downloading `index.htm` from `/web/doc2`. During the transaction for processing that request, ConfigRE observes that the part of the request related to directory name has also been compared with '`/web/doc1`'. This prompts us to adjust the request to explore the security configuration for that directory. Such an adjustment can be done automatically using symbolic execution [15].

3.3 Detection of Permission Structures

Since the determination of permissions in the second step is inconclusive, we need to further study candidate permission fields to understand whether they indeed control the interactions between subjects and objects. For this purpose, we developed a technique which first identifies the alternative values for these fields and then reruns the application over these values to investigate their accumulated effects on a transaction. Below we elaborate this approach.

Discovery of alternative values. Evaluation of an application using different values of a field can tell us whether a transaction is under the field's control. For example, once we switch '`Yes`' to '`No`' in Figure 2, an attempt to read a file under '`/usr/www/`' will no longer get through, which suggests that the field is a permission. To discover these values, we first need to identify which fields can be used in this game. Some fields cannot be changed without affecting other fields. Consider a field that serves as other fields' parent node in the semantic tree: any change that happens to its content could affect all its descendant nodes and even completely alter the structure of the tree. On the other hand, inner nodes of the semantic tree usually represent the name of a configuration setting, such as '`FileAccess`' in Figure 2, while the value of the setting is represented by the tree's leaves. Given these considerations, our current design of ConfigRE only explores alternative values for the candidate permission fields on the leaves. For those on the internal nodes, we evaluate them under two scenarios: either leaving them there intact, or removing them along with all their descendants. This treatment may cause configuration errors, which however, can be easily identified from an application's error report.

For a leaf field annotated as a candidate permission, ConfigRE takes the following approach to discover its alternative values. It detects the instructions that compare the field with a constant, and then changes the content of the field to a random value to cause the

---

[3]An example is using the file name like '`/../`'.

operation to fail. This forces the application to try all legitimate values for that field one by one until none of them matches, which can be observed from a configuration error reported by the application. For the example in Figure 2, our approach changes '`Yes`' to a random string, which makes the execution reveal '`No`'. In this way, we can capture all the values associated with a field.

Generation of permission structures. Using the alternative values of candidate permission fields, ConfigRE automatically modifies configuration files and reruns an application upon them. Our approach exhaustively tests all combinations of these fields' alternative values, and the two scenarios for evaluating internal nodes to understand the influences of these fields on a transaction. According to the outcomes of the test, these configurations are classified into three categories: those making the transaction successful are kept in a permit set P, those accepted by the application but causing the request to be denied are put in a denial set D, and the rest are thrown away because of configuration errors.

The configurations in P and D may include the fields unrelated to permissions. Identification of these fields in ConfigRE is based upon the observation that their values, once altered, do not change the outcomes of the test. Specifically, we first pick up from P or D a *permission vector* — a conjunction of the fields which makes the test transaction either succeed or fail. Then, we examine every permission field on that vector. For a specific field, its content is changed to all other values it can take one by one for checking whether any of these changes also causes the vector to be moved to the set other than its origin. For example, in Figure 2, once we change '`Yes`' at Line C3 to '`No`', the field that holds these values will be moved from the permit set to the denial set. If this does not happen to that field for all the values it can take, and for every vector from both sets that involves that field, we conclude that its value is not important for an access-control decision, and therefore remove the field from P and D. In this way, all the fields unrelated to permission can be eliminated from both sets. As a result, the permission structure can be represented as the disjunction of all the vectors in P for allow and the disjunction from D for denial.

Discussion. As we described above, an internal node and its subtree can be taken away from a configuration file during the rerun to determine whether it is a permission field. This, however, could cause a subject or object field to be removed when the field happens to be the node's descendent. An example is '`Allow`' in Figure 3. When this happens, some permission vectors will no longer relate to any subject or object in the configuration file. In our research, we drop all the vectors that are not associated with objects, because they do not reflect the targets of access control. On the other hand, a permission not attached to a specified subject can be interpreted as being applied to the unknown (or anonymous) user, and therefore is kept in our permission structures.

Our permission analysis technique can also help detect special subjects such as '`All`'. Such subjects may not be detected by dataflow analysis, and instead, they could be annotated as candidate permission fields because they could affect branch conditions of a transaction, as a permission field does. The semantic meanings of these fields can be uncovered in the step for detecting alternative values, which observes that the values these fields accept are actually used like subjects: for example, they can be the parameter for `gethostbyname()`.

3.4 Generation of Specification Languages

The last step of ConfigRE is to convert the knowledge about subjects, objects and permission structures acquired from previous steps into a means for identifying security policies specified

in other instances of the applications. This is achieved in our research through automatically generating a language that parses the configuration files of these instances to recognize the fields related to a policy definition, which are used to identify a security policy.

Since the language is for scanning a configuration file, it should include a sufficient amount of syntactic information for connecting semantically related fields together. To generate such a language, our approach correlates two connected nodes in a semantic tree with delimiters, and then creates a BNF grammar for recognizing these nodes' semantic relations, as described by the tree, from their syntactic relations, as revealed by these delimiters. For example, consider the semantic tree in Figure 2. Two semantically-related nodes 'FileAccess' and 'Yes' are connected through the delimiter '\n', and 'FileAccess' is further linked to 'Directory' through a delimiter pair '{' and '}'. The language we generated for this example, as illustrated in the Figure, describes such a relation for every pair of connected nodes in the tree.

Preprocessing. To prepare for language generation, ConfigRE makes the following moves. For every node annotated as subject, object or confirmed permission, our approach first extracts from the semantic tree all its ancestor nodes to build a new tree. The tree is further generalized: the specific contents of subject or object nodes are removed and alternative values for leaves are added. In addition, the subtrees of the root's child nodes can also be merged if two such nodes are identical. For example, Figure 3 illustrates a subtree built from a merger between the subtrees for directory '/web/doc1' and '/web/doc2'. Our approach further examines every edge of the tree and tries to identify a set of delimiters that connect its two end nodes: for example, in Figure 3, the new line character '\n' connects all the fields on a line together, and the paired delimiters, 'Directory' and '⟨/Directory', links the node '⟨Directory' with 'Order', 'Allow' and 'Deny'. If there exist multiple such delimiters or pairs, we select the one with the shortest *range*, which is defined as the interval the application scanned for the delimiter. In some cases, the syntactic relation between two nodes is established by more than one delimiter or delimiter pair. Consider the example in Figure 2. The node 'Directory' is connected to 'IPPrefix' and 'FileAccess' by multiple delimiters: first, '\n' is used to combine 'Directory', '/usr/www' and '{' togather, and then, the paired delimiters '{' and '}' connects the root to other fields. ConfigRE adopts an algorithm that first tries to establish a direct relation between two nodes, and if fails, continues to seek indirect relations: it finds the delimiter whose range includes the parent node of these two nodes ('\n' in the example); then, our approach searches within the range for the head of paired delimiters that can be used to link the parent node to its child. Such delimiters are annotated to the edge. In addition to the delimiters that embrace two nodes, we also add onto the edge the delimiters between these two nodes. As a result, we get a tree for language generation, which we call *knowledge tree* or *K-tree*. Figure 2 gives an example.

Language generation. From a K-tree, ConfigRE automatically generates a specification language. The grammar of the language can be described as a 3-tuple $\langle \Sigma, V, R \rangle$, where Σ is a finite set of terminal symbols, V a set of nonterminals and R a set of production relations. ConfigRE first adds the delimiters and the values of the fields in the K-tree to Σ. Note that for the fields representing subjects or objects, their specific contents are removed and replaced with labels *subject* or *object*. Also placed into Σ is a special terminal, ε for empty.

The rest of the grammar is built through a breadth-first traversal of the K-tree. Starting from the root $n_{0,0}$, our approach des-

ignates a nonterminal $A_{i,j}$ to $n_{i,j}$, the jth node on the ith level of the K-tree, if the node is not a leaf. The production relation of the symbol is in the following form: $A_{i,j} \rightarrow S_{i,j} U_{i,j} T_{i,j}$, where $S_{i,j}$ represents a start symbol, $U_{i,j}$ is a nonterminal for describing the relations among the non-leaf children of $n_{i,j}$, and $T_{i,j}$ is an end symbol. ConfigRE determines the start symbol using the field of node $n_{i,j}$, the leaf directly attached to that node and the delimiters connecting them. If there is a pair of delimiters that embraces the node's subtree, the head of the pair also becomes part of $S_{i,j}$. The end symbol $T_{i,j}$ is a delimiter (or the tail of a delimiter pair) that connects the node to all its children. The production relation of $U_{i,j}$ is determined by the relations among the non-leaf children of $n_{i,j}$ as described by the permission structure. For the example in Figure 3, let $A_{2,1}$, $A_{2,2}$ and $A_{2,3}$ be the nonterminals for 'Order', 'Allow' and 'Deny' respectively; according to the permission vectors that contain the combinations of the fields ('Order', 'Allow'), ('Order', 'Deny'), ('Order'), ('Allow'), ('Deny') and empty, following production relation is constructed: $U_{10} \rightarrow A_{2,1}A_{2,2}|A_{2,1}A_{2,3}|A_{2,1}|A_{2,2}|A_{2,3}|\varepsilon$. For simplicity of presentation, we also assign a nonterminal to the leaf with multiple alternative values. This process continues until every node of the K-tree has been described by the language.

As an example, we present in Figure 2 a grammar for the configuration of the HTTP server. The grammar is constructed in the following way. After ConfigRE encounters the node 'Directory', it gives the node a nonterminal $A_{1,0}$, determines the start symbol $S_{1,0} \rightarrow$Directory object { \n, the end symbol $T_{1,0} \rightarrow$'}', and the production of $U_{1,0}$ that describes the relation among the node's internal child nodes. These children include 'IPPrefix' and 'FileAccess', which are given new nonterminals $A_{2,0}$ and $A_{2,1}$ respectively. The production rule is $U_{1,0} \rightarrow A_{2,0}IA_{2,1}$, where I is a nonterminal for ignoring all delimiter terminals between two nonterminals. Note this is sufficient for skipping all the configuration settings unrelated to access control, because a lexicon analyzer used in a language parser can automatically bypass all the characters not in the terminal set. 'IPPrefix' and 'FileAccess' only have leaf nodes as their children, and the end symbols associated with them is '\n'. The production of $A_{2,1}$ also involves a nonterminal B, as the child of 'FileAccess' has alternative values 'Yes' and 'No'. The grammar for Apache configurations is described in Section 4.

Configuration scanning. A language can be conveniently converted into a configuration scanner using the standard parser generator such as BISON [13]. Such a scanner recognizes terminals from a configuration file using a lexicon analyzer such as FLEX [1], and then identifies the access control components related to a policy definition. These components are used to determine an access control policy.

4. EVALUATION

In this section, we describe our experimental study of ConfigRE. The objective of this study is to evaluate the effectiveness of our technique in recovering the specifications for applications' security configurations. To this end, we ran our prototype system against 6 real applications as demonstrated in Table 1. Our study involves extraction of the semantic information of these applications' access-control settings, generation of specification languages and evaluation of the configuration scanners automatically constructed from the languages. The experiments were conducted on a Linux workstation installed with Rehat Enterprise 4. The host has 2.40GHz Core 2 Duo processor and 3GB memory. Due to the space limit, here we use Apache and Bftpd as two examples to explicate this
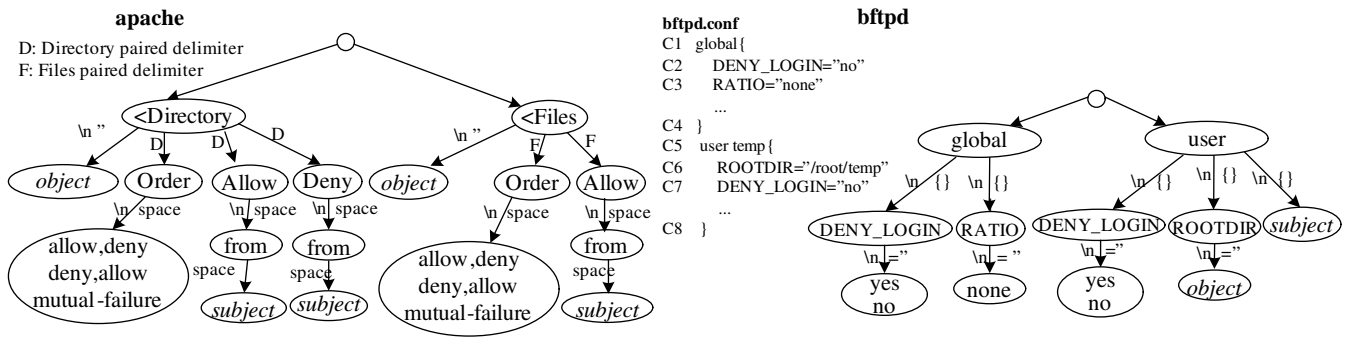
Figure 4: K-trees for Apache and Bftpd.

study. The results for other applications can be found in a longer version of the paper [29].

Table 1: Effectiveness of ConfigRE.

Programs	Types	Test req	Tot # of fields	# of sec fields	Detected
Apache	HTTPD	GET	51	21	21
pServ	HTTPD	GET	18	2	2
Null-HTTPd	HTTPD	GET	21	2	2
vsftpd	FTPD	login, get	16	4	4
Bftpd	FTPD	login, get	18	10	10
Napster	P2P	P2P download	21	2	2

4.1 Access Control Configurations

In the first experiment, we ran our prototype on these applications to partition their configuration files into fields, extract those that represent subjects, objects and permissions, and identify their permission structures. Table 1 illustrates some experimental settings and outcomes, including the names and types of the applications, the test requests used in the experiment, total number of the fields in configuration files, the number of the fields related to access control and the number of these fields ConfigRE detected. From the table, we can see that our prototype successfully captured all the security-related fields.

Apache. We evaluated our prototype using an Apache server with security configurations as described in Figure 3. Apache allows the user to specify security policies for individual directories and files through a set of *directives*. The main directives for access controls are Order, Allow and Deny. Allow permits a remote subject to read a local file and Deny blocks such a request. Order defines the order in which Apache checks the permission rules for a request and the permissions for unspecified subjects: if the value of the directive is 'Allow, Deny', then the server drops all the requests not explicitly specified by an Allow rule; if the value is 'Deny, Allow', requests will be processed unless they are explicitly denied. These permissions are defined under the directives ⟨Directory for directories or ⟨Files for individual files.

Our prototype detected delimiters of the configuration file and used them to partition the file into fields. An interesting observation here is that Apache does not treat '⟨' and '⟩' as normal delimiters and as a result, our approach took '⟨' as part of a directive field and failed to catch '⟩'. Fortunately, this did not affect the follow-up analysis. In the second step, those fields were tainted for analyzing how the server processed a GET request. This analysis revealed subjects, objects and candidate permissions. Among these candidate fields were Options and FollowSymLinks, which were found to be unrelated to access control during the rerun stage. Alternative values 'Deny, Allow', 'Allow, Deny' and 'mutual-failure' were also detected. The access-control

components and permission structures identified by our prototype are described in Table 2.

Bftpd. Bftpd [2] is a small FTP server which has been widely used on a variety of platforms including Linux, BSD and routers. Like many other FTP server, Bftpd supports both anonymous user access and local user access. In the latter case, it lets the users of an operating system (OS) access files and directories using their privileges bestowed by the OS. The interesting part of this FTPD is that it can be configured to overrule a user's OS permissions. For example, one's root directory can be redefined, which could enable her to download the files from a directory she cannot access with her OS privileges. Though Bftpd enforces some restrictions to this potentially dangerous functionality, for example, forbidding one to set /root as a user's home directory, its security implications remain. In our experiment, we evaluated our prototype using a configuration file containing such settings. The fragment of the file is presented in Figure 4.

The fragment contains a set of global configurations defined under the keyword 'global'. They are applied to every user but can be overridden once the same settings are explicitly specified for that user. Consider the example in Figure 4. 'DENY_LOGIN' appears under both 'global' and user temp to specify whether the user is not allowed to login. The latter supersedes the former. Actually, this setting and 'ROOTDIR', which changes one's root directory, are the main access-control configurations Bftpd provides to adjust a user's OS-defined read privilege.

Our prototype analyzed Bftpd using a test request for downloading a file. This analysis detected the field 'temp' as a subject from a match that happened between the field and part of the request related to username, and '/root/temp' as an object from the observation that it was used to encapsulate the value tainted by the request as a directory path for opening a file. We also found alternative value for the field 'no', which is 'yes', and permission fields and permission structures, as illustrated in Table 2.

Our approach effectively captured the relations between two DENY_LOGIN fields defined under 'global' and 'user' through reruns: the second setting overrules the first one, but when the former is not there, the latter takes effects. The field ROOTDIR was confirmed as a permission and a parent of the object. As a result, all the permission vectors not including it were removed. Another interesting observation is that a field seemingly unrelated to security, RATIO, was also identified as a permission. The field is for specifying upload/download ratio. In our experiment, we found that removal of the field did not incur any configuration errors but caused the processing of the request to fail. Therefore, it can be deemed as an implicit permission field.

Table 2: Apache and Bftpd. Here we list both the fields identified as access control components (highlighted items) and their parent nodes in semantic trees. Note that production relations like Rule 3 are constructed in a recursive manner for continually scanning a configuration file after recognizing a security policy until the end of the file.

App	Subjects/Objects/Permissions	Structures	Languages
Apache	*Subjects*: ⟨Directory Allow from **192.168.0.0/24** ⟨Files Allow from **192.168.1.0/24** ⟨Directory Deny from **192.168.2.0/24** *Objects*: ⟨Directory **/web/doc1** ⟨Files **sensitive.htm** ⟨Directory **/web/doc2** *Permissions*: p_1:⟨Directory **Order allow,deny** p_2:⟨Directory **Order deny,allow** p_3:⟨Directory **Order mutual-failure** p_4:⟨Directory **Allow from** p_5:⟨Directory **Deny from** p_6:⟨Files **Order allow,deny** p_7:⟨Files **Order deny,allow** p_8:⟨Files **Order mutual-failure** p_9:⟨Files **Allow from** Home Directory Permission	• *Directory Permit*: $(p_4) \vee (p_4 \wedge p_3) \vee (p_4 \wedge p_2) \vee (p_4 \wedge p_1) \vee (p_2) \vee (\varepsilon)$ • *Directory Denial*: $(p_1) \vee (p_3) \vee (p_5) \vee (p_5 \wedge p_3) \vee (p_5 \wedge p_2) \vee (p_5 \wedge p_1)$ • *Files Permit*: $(p_9) \vee (p_9 \wedge p_8) \vee (p_9 \wedge p_7) \vee (p_9 \wedge p_6) \vee (p_7) \vee (\varepsilon)$ • *Files Denial*: $(p_6) \vee (p_8)$	1. Σ = { ⟨Directory, ⟨/Directory, '\n', Order, allow,deny, deny,allow, mutual-failure, Deny, from, Allow, ⟨Files, object, subject, ⟨/Files} 2. V = { $A_{00}, A_{10}, U_{10}, A_{21}, B_{21}, A_{22}, U_{22}, A_{23}, A_{23}, A_{11}, U_{11}, A_{25}, B_{25}, A_{26}, U_{26}, U_{25}, A_{26}, I$ } 3. $A_{00} \rightarrow A_{10}A_{00} | A_{11}A_{00} | I | \varepsilon$ 4. $A_{10} \rightarrow$ ⟨Directory *object* '\n' U_{10} ⟨/Directory '\n' 5. $U_{10} \rightarrow A_{21}IA_{22} | A_{21}IA_{23} | A_{21} | A_{22} | A_{23} | I | \varepsilon$ 6. $A_{21} \rightarrow$ Order B_{21} '\n' 7. $B_{21} \rightarrow$ allow,deny	deny,allow	mutual-failure 8. $A_{22} \rightarrow$ Allow U_{22} '\n' 9. $U_{22} \rightarrow$ from *subject* 10. $A_{23} \rightarrow$ Deny U_{23} '\n' 11. $U_{23} \rightarrow$ from *subject* 12. $A_{11} \rightarrow$ ⟨Files *object* U_{11} ⟨/Files '\n' 13. $U_{11} \rightarrow A_{25}IA_{26}$ 14. $A_{25} \rightarrow$ Order B_{25} '\n' 15. $B_{25} \rightarrow$ allow,deny	deny,allow	mutual-failure 16. $A_{26} \rightarrow$ Allow U_{26} '\n' 17. $U_{26} \rightarrow$ from *subject* 18. $I \rightarrow \varepsilon | I$ '\n'

Bftpd	*Subjects*:
	    user **temp**
	Objects:
	    user ROOTDIR **/root/temp**
	Permissions:
	  p_1: global **DENY_LOGIN no**
	  p_2: global **DENY_LOGIN yes**
	  p_3: global **RATIO none**
	  p_4: user **DENY_LOGIN no**
	  p_5: user **DENY_LOGIN yes**
	  p_6: user **ROOTDIR**
	Home Directory Permission
		• *Permit*:
		$(p_1 \wedge p_3 \wedge p_4 \wedge p_6) \vee (p_2 \wedge p_3 \wedge p_4 \wedge p_6) \vee (p_3 \wedge p_4 \wedge p_6) \vee (p_2 \wedge p_3 \wedge p_6) \vee (p_3 \wedge p_6)$
		
		• *Denial*:
		$(p_4 \wedge p_6) \vee (p_2 \wedge p_4 \wedge p_6) \vee (p_1 \wedge p_4 \wedge p_6) \vee (p_1 \wedge p_5 \wedge p_6) \vee (p_2 \wedge p_5 \wedge p_6) \vee (p_5 \wedge p_6) \vee (p_3 \wedge p_5 \wedge p_6) \vee (p_2 \wedge p_3 \wedge p_5 \wedge p_6) \vee (p_1 \wedge p_3 \wedge p_5 \wedge p_6) \vee (p_2 \wedge p_3 \wedge p_6) \vee (p_6) \vee (p_2 \wedge p_6) \vee (p_1 \wedge p_6)$
			1. Σ = {'{', '\n', '}', '=', "", *object*, *subject*, global, DENY_LOGIN, yes, no, RATIO, none, user, ROOTDIR}
			2. V = { $A_{00}, A_{10}, U_{10}, A_{20}, B_{20}, A_{21}, A_{11}, U_{11}, A_{23}, A_{24}, B_{24}, I$ }
			3. $A_{00} \rightarrow A_{10}IA_{11}$
			4. $A_{10} \rightarrow$ global '{' '\n' U_{10} '}' '\n'
			5. $U_{10} \rightarrow A_{20}IA_{21} | A_{20} | A_{21} | \varepsilon$
			6. $A_{20} \rightarrow$ DENY_LOGIN '=' "" B_{20} "" '\n'
			7. $B_{20} \rightarrow$ yes
				no
				8. $A_{21} \rightarrow$ RATIO '=' "" none "" '\n'
				9. $A_{11} \rightarrow$ user *subject* '{' '\n' U_{11} '}' '\n'
				10. $U_{11} \rightarrow A_{23}A_{24} | A_{24}$
				11. $A_{23} \rightarrow$ ROOTDIR '=' "" *object* "" '\n'
				12. $A_{24} \rightarrow$ DENY_LOGIN '=' "" B_{24} "" '\n'
				13. $B_{24} \rightarrow$ yes
					no
					14. $I \rightarrow \varepsilon | '{' I | '\n' I | '}' I | '=' I | "" I$

4.2 Languages and Scanners

After identifying the access control components of the applications, our prototype continued to generate specification languages for their security-related configurations. Based upon these languages, scanners for recognizing access control policies from these applications' configuration files were also automatically created, and evaluated in our experiments. Again, we use Apache and Bftpd as examples here to elaborate this study.

Languages. ConfigRE first created a K-tree for each application through annotating its semantic tree with the delimiters that linked semantically-related fields together. The K-trees for Apache and Bftpd are illustrated in Figure 4. From the K-trees, our prototype automatically generated specification languages for the applications. As an example, Table 2 describes the languages for Apache and Bftpd, which accurately capture the syntactic relations among individual access control components detected from these applications.

Evaluations of scanners. The languages generated by the prototype were automatically converted into scanners by `yacc` [4]. These scanners were tested in our experiments on other configuration files of these applications. In particular, we ran the scanner for Apache on a configuration file used by a major university's web server, which contained 1341 lines of settings. Our scanner successfully detected all configuration settings except those involving SSL authentications and a policy that denies access to a file. This

is because both of them did not appear in the configuration file the prototype used to generate the language. We manually checked the access control policies the scanner identified, and found all of them were correct.

We further evaluated other scanners using synthesized configuration files. These files were constructed in a way that mingles the settings related to access control with those not. Specifically, we injected the configuration commands unrelated to security to those that define access control components. We also randomly picked up the names or addresses for subjects and objects. From the experiments, we found that the scanners always correctly detected the security policies from these files, as long as similar configurations appeared in the files used for extracting configuration knowledge.

5. RELATED WORK

Misconfigurations are well-recognized to be one of the most serious operator mistakes that affect performance, availability and security of a computing system. This problem has been extensively studied under the scenarios of configuration management. For example, Glean infers correctness constraints for Registry [16]; Strider [30] compares the registry of a system with a "healthy" registry to detect potential misconfiguration; PeerPressure [28] further extends the idea of Strider to correct misconfigurations through differential analysis and statistical approaches. Other important research includes Validation [21] and Chronus [31]. More recent work is focused on automatic generation of correct configuration

files for a system using custom declared specifications. Examples include the approach proposed by Zheng, et al [34], Cfengine [7], LCFG and SmartFrog [5]. Besides the efforts from the research community, software manufacturers also offer tools for checking configurations through retrieving a database of known misconfigurations. A prominent example is the Microsoft Baseline Security Analyzer [3]. Most of these approaches assume *a priori* knowledge about system configurations in some forms, for example, correct registries [30, 28], configuration templates [34] and the database of known misconfigurations. Our approach aims at automatically discovering such knowledge from an application's executables. This is of particular importance for checking the applications whose configuration specifications are not well documented or even close.

Closely related to our research is the problem of protocol reverse engineering, which has been intensively studied recently [12, 8, 18, 32]. Most of existing approaches are for automatic extraction of protocol fields. For example, Discover [12] employs clustering algorithms to recover fields from network traffic; Polyglot [8], Autoformat [18] and the approach proposed by Wondracek et al [32] all utilize dynamic taint analysis to detect protocol fields from the applications knowing how to parse the protocol. Different from these approaches, ConfigRE is designed for automatic reverse engineering of program configurations, which requires semantically understanding the roles played by individual fields in defining an access-control policy. Wondracek et al [32] also discussed using the parameters of API functions to understand the semantic meaning of some protocol fields such as file name. However, this is insufficient for our purpose because we also need to recognize the semantic relations among individual fields, for example, the context (such as the field ⟨Files) under which the file name is specified. In addition, our approach bridges the syntax-semantic gap to automatically generate a language for misconfiguration detection. On the other hand, those existing techniques can be used in configuration reverse engineering for partitioning configuration files into fields, as we did in our research.

Techniques for instruction-level taint tracking have been intensively studied in these years. Numerous approaches have been proposed. Prominent examples include TaintCheck [23], TaintTrace [9], Memcheck [26], RIFLE [27] and LIFT [25]. These techniques are widely applied to analyze software vulnerabilities [6, 11], study malware [33], reverse engineer protocols [8, 32] and generate protocol replayer [22]. While most of existing approaches are based upon dataflow analysis, techniques for control-flow based taint analysis have also been proposed. Examples include Dytan [10] and the technique for dynamic spyware analysis [14]. These existing techniques can serve to improve the taint analyzer used in our approach.

6. DISCUSSION

Our research on ConfigRE made the first step towards automatic extraction of configuration knowledge from an application. In this section, we discuss its limitations and the future research our work could inspire.

The basic idea behind ConfigRE is general: it can be used to handle not only text-based configuration files but also binary-based files. However, our current implementation is oriented towards text-based configurations for two reasons: first, the field extraction technique is for processing text-based input streams; second, language generation does not consider the delimiters specific to binary configurations such as length fields. Research on the first problem can benefit from the existing approaches for parsing binary protocols [8, 32] though more work is expected to improve their effectiveness. We will also improve our design for language generation

to accommodate the features of binary configuration, which can lead to the solution to the second problem.

ConfigRE relies on control-flow based taint analysis to detect configuration fields' semantic relations. Control-flow taint analysis is well-known to be hard, as it could cause a large amount of irrelevant data to be mistakenly tainted. Our current mitigation of the problem is empirical, and could cause false positives, i.e., taint of unrelated data, or false negatives, missing of the data that should be tainted. Note that the false positives can be mitigated in the third step of our approach, where candidate permission fields undergo a rerun test. An improvement can be achieved through incorporating into our approach existing control-flow taint analysis techniques, such as Dytan [10].

In the presence of a large number of alternative values, the technique based upon rerun can become problematic, as far as performance is concerned. A potential solution is to utilize static analysis to check whether a field indeed affects the accomplishment of a transaction, and rerunning an application only in the cases static analysis fails.

An important problem worth intensive effort is the coverage of an analysis: if some authorization settings do not appear in the configuration files used in the analysis, their specifications may not be discovered by our current approach. A technique that extracts an entire authorization specification will be a step forward. Efforts on this direction can be assisted by the existing techniques for exploring multiple executions [20]. However, these techniques are pretty heavyweight, very time-consuming even for the programs of moderate sizes. How to take advantage of configuration features to improve their efficiency can be a nontrivial problem.

The current design of ConfigRE only considers the interactions between the remote client and the local file system. There are many other security-related configuration issues the follow-up research could embark on: an example is discovery of the security settings for downloading and executing a remote script. Another interesting question is how to automatically detect the hidden configurations of less than innocent programs such as Trojan horses. This can be a very challenging problem, because these programs may contain obfuscated code to discourage a binary analysis.

The prototype we implemented is not fully automatic. Human intervention is needed to check the configuration fields it extracts. This problem comes from the fact that existing field-extraction techniques [8, 18, 32], which we built our prototype upon, cannot guarantee to correctly identify all configuration fields: occasionally, they could miss some delimiters necessary for field identification. As a result, we had to manually examine the fields discovered by our prototype to eliminate the problems such as merger of two fields into one. Moreover, the test request used in our research was also manually adjusted to explore the security configurations for multiple subject/object pairs. Such an adjustment is based upon the information automatically discovered during an analysis, and can be fully automated, as discussed in Section 3.2.

7. CONCLUSION

Knowledge of configuration file formats can have significantly security implication. For example, it lays the foundation for automatic detection of security misconfigurations. However, such knowledge is not well-documented for many applications, and becomes increasingly inaccessible due to the use of graphic user interfaces for indirect configurations. Effective solution to this problem relies on automatic analysis of an application to reverse engineer its configuration specification. As a first step toward this end, we present in this paper a new technique called ConfigRE. Our approach first identifies individual configuration fields and their se-

mantic relations, and then detects the fields related to access control components such as subjects, objects and permissions. Using such information, ConfigRE automatically generates a specification language for access control configurations. The language is further converted into a scanner to check configuration files for the security policies specified in an application. We evaluated ConfigRE using real applications, which demonstrates the efficacy of our technique. The future research includes extending our current design to handle binary-based configurations and the policy specifications for more complicated activities such as executing a script.

8. ACKNOWLEDGEMENTS

9. REFERENCES

[1] flex: The Fast Lexical Analyzer. http://flex.sourceforge.net/, as of 2008.

[2] bftpd. http://freshmeat.net/projects/bftpd/, as of April, 2008.

[3] Microsoft baseline security analyzer. http://www.microsoft.com/technet/security/tools/MBSAHome.mspx, as of April, 2008.

[4] yacc. http://dinosaur.compilertools.net/yacc/, as of April, 2008.

[5] P. Anderson, P. Goldsack, and J. Paterson. Smartfrog meets lcfg: Autonomous reconfiguration with central policy control. In *LISA '03: Proceedings of the 17th USENIX conference on System administration*, pages 213–222, 2003.

[6] D. Brumley, J. Newsome, D. X. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *S&P*, pages 2–16, 2006.

[7] M. Burgess. Cfengine: A site configuration engine. *USENIX Computing systems*, 8(3):309–337, Summer 1995.

[8] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329, 2007.

[9] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *ISCC*, pages 749–754, 2006.

[10] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, 2007.

[11] J. R. Crandall, Z. Su, and S. F. Wu. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 235–248, New York, NY, USA, 2005. ACM Press.

[12] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of the 16th USENIX Security Symposium*, August 2007.

[13] C. Donnelly and R. Stallman. *Bison: The Yacc-Compatible Parser Generator*. Luniverse Inc, Bloomington, Indiana, 2000.

[14] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference (Usenix'07)*, June 2007.

[15] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[16] E. Kycyman and Y.-M. Wang. Discovering correctness constraints for self-management of system configuration. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, pages 28–35, 2004.

[17] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[18] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *NDSS*, 2008.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[20] A. Moser, C. Krügel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[21] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 5–5, 2004.

[22] J. Newsome, D. Brumley, J. Franklin, and D. X. Song. Replayer: automatic protocol replay by binary analysis. In *ACM Conference on Computer and Communications Security*, pages 311–321, 2006.

[23] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[24] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and System (USITS '03)*, 2003.

[25] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, pages 135–148, 2006.

[26] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[27] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *MICRO*, pages 243–254. IEEE Computer Society, 2004.

[28] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, pages 245–258, 2004.

[29] R. Wang, X. Wang, K. Zhang, and Z. Li. Towards automatics reverse engineering of software security configuration. Technical Report IUCS-TR667, Indiana University, 2008.

[30] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA*, pages 159–172, 2003.

[31] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 6–6, 2004.

[32] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *NDSS*, 2008.

[33] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[34] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic configuration of internet services. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 219–229, New York, NY, USA, 2007. ACM.