RC24243 (W0704-095) April 18, 2007
Computer Science

IBM Research Report

Experiences with Building Security Checking and Understanding Tool

Ted Habeck, Larry Koved, Orlando Marquez, Vugranam C. Sreedhar, Michael Steiner, Wietse Venema, Samuel Weber
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Gabriela Cretu
Columbia University
New York, NY 10027

Krishnaprasad Vikram
Cornell University
Ithaca, NY 14850

Experiences With Building Security Checking and Understanding Tool

Ted Habeck
Larry Koved
Orlando Marquez
IBM TJ Watson Research Center
Hawthrone, NY 10532
{habeck,koved,omarquez}@us.ibm.com

Vugranam C. Sreedhar
Michael Steiner
Wietse Venema
Samuel Weber
IBM TJ Watson Research Center
Hawthrone, NY 10532
{vugranam,msteiner,wietse}@us.ibm.com,
samweber@watson.ibm.com

Gabriela Cretu [*]
Columbia University
New York, NY 10027
gcretu@cs.columbia.edu

Krishnaprasad Vikram[*]
Cornell University
Ithaca, NY 14850
kvikram@cs.cornell.edu

Abstract

In this paper we present our experience in building security checking and understanding tools for Java, PHP, and JavaScript languages. The main theme of our work is how to make security accessible to application developers, who are typically not well versed in the nuances of secure software development. Therefore, from a tooling perspective, we provide extensions to an integrated development environment (IDE) based on the Eclipse Platform that has the ability to fix and address security problems and issues in a manner consistent with that is currently expected for syntax errors. We provide "easy buttons" and "quick fixes" requiring as few clicks in the IDE as possible to perform reasonable security problem fixes. We rely heavily on static and dynamic analysis, and a repetoire of security policies and coding practices to drive the usability of our tools. We also discuss some of the technical and non-technical challenges that we encountered during the development of our tools.

1. Introduction

For several years, members of IBM Research's Security and Privacy Department have been prototyping and building security checking and understanding tools. Our ultimate goal is to encourage, and improve the ability of, code developers to consider security issues as the software is being developed, rather than as an afterthought. Application developers are typically unaware of the nuances of secure software development. We strive to provide active assistance to this large and growing community. It is here that we feel that we can have the greatest impact.

Static analysis techniques have been in existence over the past few decades and have been successfully applied to many applications, including code optimization and bug finding. We have implemented several sophisticated static analyses, including context sensitive analysis and typestate analysis. We have found, however, significant technical and non-technical challenges in creating "usable" security tools. We observe that these sophisticated analyses need to become transparent in order to be acceptable to our end users. The consistent feedback we have received from our customers is that an analysis tool will be only useful if it is well integrated into the user's existing processes, providing security-relevant information without interfering with normal development. Careful consideration must be given to the organizational and of the social aspects of security development, how it is deployed and integrated into development environments and the build and testing processes.

One typical approach to enabling security in an application is illustrated below in the context of defining the security policies for a Java library so that the library can be run when the security authorization subsystem [14] is turned on.

```
Write the Java library
Enable Java security subsystem (the SecurityManager)
Write test cases
repeat {
  Run test cases
  Review SecurityExceptions thrown
  Inspect the stack trace
  Identify the CodeBase missing the Permission
  Insert privileged code if appropriate and/or
  Grant permissions ? use policy tool or text editor
} until(test cases run without SecurityExceptions)
```

Clearly, the above approach for adding permissions for large Java applications is not practical. Furthermore, adding appropriate Java permissions is just one of a number of steps that developers have to perform.

[*] The work described in this paper was partly done during the author's summer internship at IBM TJ Watson Research Center

The main theme of our work is how to make security accessible to application developers, who are typically not well versed in the nuances of secure software development. Therefore from a tooling perspective, we want our tools to be part of an integrated development environment (IDE) which has the ability to fix and address security problems and issues in a manner consistent with that is currently expected for syntax errors. We want to provide "easy buttons" and "quick fixes" requiring as few clicks in the IDE as possible to perform reasonable security problem fixes. Figure 1 illustrates this aspect in our current SWORD4J tool (see Section 2). In the figure we highlight a security problem in dealing with privileged instructions (similar to highlighting syntax errors), and quick fixes are provided to fix the problem.

Providing such easy buttons and quick fixes for security errors is quite challenging. In contrast to syntax and type checking, which are driven by well defined language grammars and typing rules, security policies and coding rules are not universally accepted and rarely formalized. Over the past few years we have developed and implemented many security and coding rules, and it is these rules that drive the usability of our tools.

Our current security checking and understanding tools (SCUT) are implemented as plug-ins to the Eclipse IDE. Eclipse was chosen as our target because of its user community acceptance and support for extensions. In particular, it includes assets such as Java Development Tool (JDT), CodeReview and Rational Application Analysis, and PHP Development Tool (PDT). Our experience with building and deploying such tools has been quite promising technically, resulting in novel analyses and influencing the development community.

In the rest of paper we present our past and present experience in building security checking and understanding tools for the Java language, the PHP language, and the JavaScript language. We will discuss some of the pain points that we encountered, both in terms of using analysis frameworks, and in terms of deploying and integrating the analysis framework with our Eclipse Platform. In Section 2 we will describe our experience with building a security analysis tool for Java, called SWORD4J. In Section 3 we will discuss our current effort with building a security analysis tool for PHP. In Section 4 we will discuss security for JavaScript language. In Section 5 we will discuss our current and future plan for building a robust and a usable security checking and understanding tool that is based on Eclipse development model. In Section 6 we discuss related work and conclude in Section 7.

2. SWORD4J: Security for Java

The Security Workbench Development Environment for Java (SWORD4J) is a security checking and understanding tool that we are developing for addressing security issues and problems in Java applications.[1] SWORD4J makes use of novel, state-of-the-art analysis techniques for detecting security errors. When a security error is detected by our analysis engine, we inform the end user (1) the nature of the security error identified, (2) the implications of the issues if not fixed (including extensive documentation), and (3) where possible, simple actions in the UI to fix the security error, including automatically updating the security policy or code refactorings.

Consider the problem of setting Java permission that we briefly discussed in the Introduction. When Java's security authorization is enabled, before any potentially sensitive operations are performed, such as file writes or network accesses, authorization checks are made to ensure that the code ultimately responsible for the action has the appropriate authority to do so. Authority is granted or denied via Permission objects. Authorization is transitive, as code

[1] Available for download at http://alphaworks.ibm.com/tech/sword4j

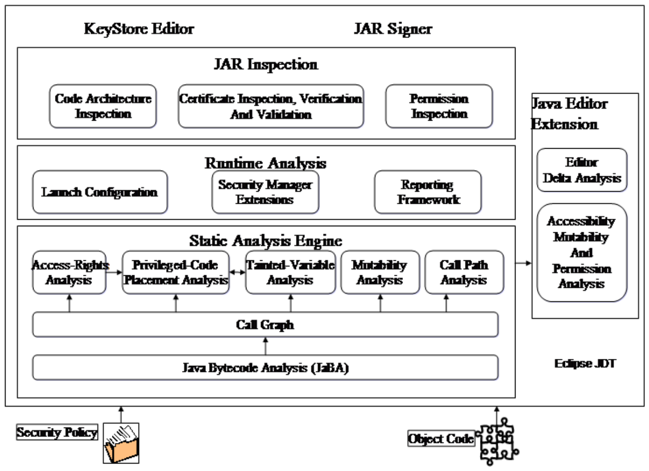


Figure 2. Architecture of SWORD4J

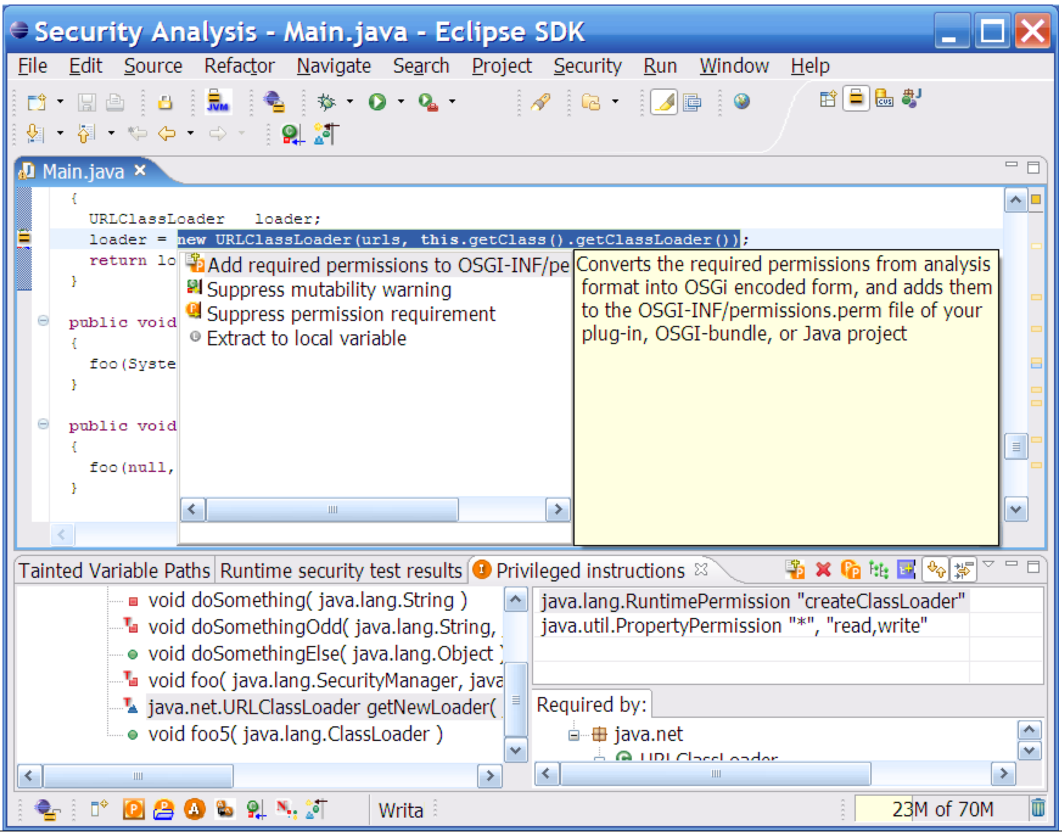
that isn't allowed network access shouldn't be able to gain access merely by calling innocent third-party code which does such access. Complicating matters is a mechanism whereby certain code can be deemed "privileged", and able to do operations that its callers cannot necessarily perform [14].

Security practices call for code being granted only the minimum authority necessary to perform its function. However, the transitive nature of authorization makes this extremely difficult for most modern applications, which are built on top of multiple framework layers: sufficient and necessary permissions must be determined for not only those operations the application specifically intends to do under all possible execution paths, but also those done by all the underlying framework and libraries. A small application code update might invoke one additional framework method, but cause a cascade of permission requirements. Recall from the Introduction, developers often determine permission requirements by repeatedly testing the application, inspecting security exceptions that occur, tweaking permissions in response, and then repeating the tests until the application appears to run successfully. This process is so burdensome that most developers simply disable authorization or grant their code universal access. In SWORD4J we use both static and dynamic analysis techniques to ease the process of defining the security policy for the code.

SWORD4J uses a static analysis framework called Java Bytecode Analyzer (JaBA) as its core analysis engine. JaBA supports several of the classical analysis techniques such as inter- and intra-procedural data flow and control flow analyses, and this framework also underlies SWORD4J. The overall architecture of SWORD4J is shown in Figure 2.

Naturally, an analysis engine for determining Java 2 Permission requirements (security policies) of code [19] is key. There is support for determining code that could be be refactored and made privileged [26]. Modifications have been made to support the commonly used and complex frameworks of Eclipse and OSGi [2]. SWORD4J also implements two tainted data detection algorithms, one based on mutability analysis [27], and the other an interprocedural tainted data flow analysis, details of which are beyond the scope of this paper. Yet another component is responsible for dynamic analysis, launching the inspected application and monitoring its behaviour.

Some notable SWORD4J features, motivated by usabilty issues and supported by the above analyses, are:


Security Analysis - Main.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Security Run Window Help
Main.java
Tainted Variable Paths
Runtime security test results
Privileged instructions
Writa
23M of 70M

```
{
  URLClassLoader   loader;
  loader = new URLClassLoader(urls, this.getClass().getClassLoader());
  return lo
}

public void
{
  foo(Syste
}

public void
{
  foo(null,
}
```

Add required permissions to OSGI-INF/pe
Suppress mutability warning
Suppress permission requirement
Extract to local variable

Converts the required permissions from analysis format into OSGi encoded form, and adds them to the OSGI-INF/permissions.perm file of your plug-in, OSGi-bundle, or Java project

void doSomething(java.lang.String)
void doSomethingOdd(java.lang.String,
void doSomethingElse(java.lang.Object)
void foo(java.lang.SecurityManager, java
java.net.URLClassLoader getNewLoader(
void foo5(java.lang.ClassLoader)

java.lang.RuntimePermission "createClassLoader"
java.util.PropertyPermission "*", "read,write"

Required by:
java.net
URLClassLoader

Figure 1. A screen shot of SWORD4J illustrating "easy button" and "quick fixes".

- Our target developers work with complex frameworks, such as Eclipse and OSGi [2]. These frameworks have security considerations that go beyond basic Java security. Our initial Permission analysis implementation showed that the these frameworks complicated permission determination sufficiently that we needed to build special analysis support in order to produce understandable feedback in such environments. Additionally, we automatically compute the analysis scope to so as to include exactly the classes which the user intends to inspect. Path sensitive filtering reduces the false positive rate. The end result is that, according to our user feedback, programmers are usually able to define Java security policies for their code and insert them into the appropriate policy files with only a few mouse clicks. In future work, we plan to add support for automatic refactoring of the code that the programmer has determined can be made privileged [26, 14].
- Support for enabling the Java authorization security subsystem. We found that determining how to enable the security subsystem was sufficiently time consuming for intended audience that it would pose a barrier for adoption. By simplifying this process, we allow developers to direct their time into the more important tasks.
- Native code and reflection cause issues for static analysis. This was the motivation behind our dynamic analysis engine implementation. If a permission was missed by our static analysis, this can be caught during runtime, localized to the specific code responsible and reported accurately to the user. For some runtime environments, such as OSGi, there are multiple policy files to be updated. The tooling is better able to accurately determine which of the many policy files needs to be updated, at a much reduced time for the developer.
- SWORD4J implements detection of best practice violations, such as not making public fields final. These are reported to the user in a similar manner to syntax errors, as the code is developed.
- One of the key mechanisms to protect Java code is to digitally sign it and verify the signature at runtime. This signature is part of the mechanism used to locate appropriate security policies for the code. SWORD4J includes a graphical user interface for digitally signing JAR files. This eliminates the need for developers to learn and switch to a separate signing tool. Just as with providing the means for enabling the Java runtime security, we wanted to minimize the learning required for developers to secure their code and deploy it.
- Managing the digital certificates used by the code signing feature described above. This enables developers to view and edit keystore entries, including changing certificate aliases, removing certificates, copying certificates between certificate stores, and importing certificates from the file system.

To summarize SWORD4J is a collection of Eclipse plugins designed to aid developers in performing security related tasks. We are continuing to expand our current repertoire of security related analysis for Java, including integrating it with a Rational application analysis tool called *Code Review*, which contain many best practices and correctness rules for Java programs.[2]

[2] http://www-128.ibm.com/developerworks/rational/library/05/higgins/

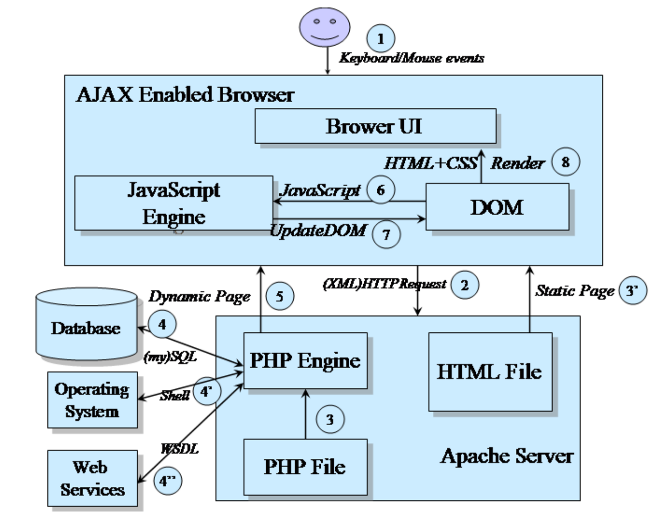
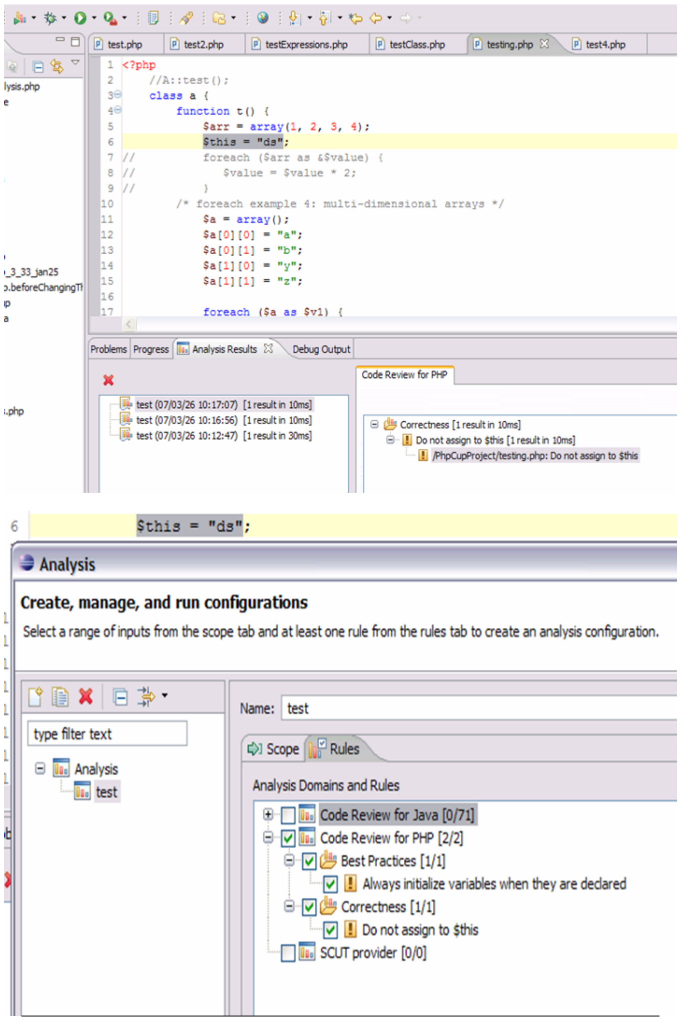


Figure 3. A Web application framework



```
<?php
    //A::test();
    class a {
        function t() {
            $arr = array(1, 2, 3, 4);
            $this = "ds";
//          foreach ($arr as &$value) {
//              $value = $value * 2;
//          }
        /* foreach example 4: multi-dimensional arrays */
            $a = array();
            $a[0][0] = "a";
            $a[0][1] = "b";
            $a[1][0] = "y";
            $a[1][1] = "z";

            foreach ($a as $v1) {
```

test (07/03/26 10:17:07) [1 result in 10ms]
test (07/03/26 10:16:56) [1 result in 10ms]
test (07/03/26 10:12:47) [1 result in 30ms]

Code Review for PHP

- Correctness [1 result in 10ms]
  - Do not assign to $this [1 result in 10ms]
    - /PhpCupProject/testing.php: Do not assign to $this

```
6            $this = "ds";
```

Analysis

Create, manage, and run configurations

Select a range of inputs from the scope tab and at least one rule from the rules tab to create an analysis configuration.

Name: test

Analysis Domains and Rules

- [ ] Code Review for Java [0/71]
- [✓] Code Review for PHP [2/2]
  - [✓] Best Practices [1/1]
    - [✓] Always initialize variables when they are declared
  - [✓] Correctness [1/1]
    - [✓] Do not assign to $this
- [ ] SCUT provider [0/0]

Figure 4. SCUT for PHP

3. PHP Analysis

PHP is an extremely popular server-side scripting language for building Web applications. In this section we will describe our experience with building tools for PHP applications. Before describing our experience let us briefly look at how a typical Web application scenario looks like.

3.1 Web Application

In a typical Web application scenario (see Figure 3), a user will interact with a browser via keyboard and mouse, generating keyboard and mouse events (step (1)). The events are intercepted by the browser environment and depending on the kind of events, the browser performs some action. Often, a user will enter a URL (Uniform Resource Locator) or click on a Web link to request a Web page from a (Apache) Web server using HTTP (Hyper Text Transport Protocol). The HTTP request (step 2) is sent over the Internet and arrives at the appropriate Apache Server (as determined by the URL). The Apache server directly handles static page requests (step 3'), and we will not be concerned with such static page requests. Often a page that is requested contains PHP scripts that may be stored in a disk on the server. The PHP script is loaded by PHP engine (step 3) and the script is then executed. The PHP engine may access database (step 4) or local file system (step 4') or invoke Web Services using WSDL (Web Service Description Language) interface (step 4''). The PHP engine then constructs a HTML page dynamically (step 5) and send the page back to the browser that requested the page. The browser constructs a DOM (Document Object Model) for the new page, and may render the page on the browser User Interface (step 8). The new page may contain JavaScripts that this then interpreted on the browser by the JavaScript Engine (step 6) and the JavaScript engine may then update the DOM. Security problems could arise at any step during the life cycle of a Web application running on Web application framework shown in Figure 3.

Unlike Java, both PHP and JavaScript run for short durations when certain Web events happen, such as, user clicking a Web link or request arriving at a Web server. Both PHP and JavaScript have many similar characteristics, for instance, both languages are weak and dynamically typed, heavily use strings as a means to communicate with other language components or the environment, and due to some of the intrinsic characteristics of the languages, programs written in them tend to expose security vulnerabilities that can easily be attacked.

From a tooling perspective we maintain the importance of providing the same IDE experience for PHP and JavaScript as for Java that we have developed for SWORD4J and Code Review. Our experience with building SCUT for PHP and JavaScript is still in its early phases, and it is not as mature as SWORD4J. In this section we will discuss our experience with the PHP language and in Section 4 we will discuss our experience with the JavaScript language.

Our current prototype for PHP tool is shown in Figure 5. This screenshot shows our Code Review functionality, which checks code for compliance with best practice, correctness and other security rules, and can support both PHP and Java. Two PHP rules are shown: 1) a best practice rule that states that users should initialize all variables, which avoids certain well-known security PHP issues (described later), and 2) a correctness rule, which detects if `$this` is assigned an arbitrary value.

Our current focus in PHP space has been understanding the semantics of the language and developing coding practices for writing secure PHP code. PHP provides many analysis challenges, usually resulting from language attempts to provide convenient functionality. There is a close correlation between features which are difficult to analyze and security failures, which we do not believe is accidental. In the next section we will overview interesting PHP

```
<?php
$username = $_GET['uname'];
$query = "SELECT id FROM users WHERE uname='$username'"
$result = mysql_query($query);
echo $result ;
?>
```

Figure 5. A simple PHP program which queries a database

characteristics and their impact on both security and analysis. In Section 3.3 we will discuss some of the data flow analyses that we are currently implementing in our tool.

3.2 PHP Characteristics

Variables in PHP do not need to be explicitly typed, and the type of a variable is determined at run-time. The same variable might hold values of different types at different program points. When a value is used, it is converted automatically and silently to the "correct" type. For instance, the expression `1+"apple"` will evaluate to 1. Clearly, this hinders type analysis, and is responsible for many program correctness issues. In the following sections we will discuss some of the intrinsic features of PHP that impacts security.

3.2.1 Dynamic Scoping

As in most programming languages, PHP variables can have local or global scope. However, in PHP variable reference can change scope from local to global. This is best illustrated by an example. Consider the following program:

```
foo();

function foo() {
  $a = true;
  while ($a)
  {
    echo "Hello.\n";
    GLOBAL $a;
  }
  echo "Goodbye.\n";
}
```

When executed, this program will output

```
Hello.
Goodbye.
```

The reason for this is that when the function `foo` is executed, the variable $a (by default local) is initialized to the value true. The while loop's test will read this value and execute its body. The statement `GLOBAL $a` states that further references to $a will be to the global variable $a, not the local. When the while loops test is again executed, it will obtain the uninitialized value of the global $a, and terminate the loop. In summary, the expression "$a" in the while loop's test will refer to two distinct variables during the program's execution.

3.2.2 Attacks

Although not an issue specific to PHP, in practice injection attacks are a major concern for PHP programmers. An injection attack essentially consists of malicious injection of strings into a PHP program in such a way that the injected string will induce an unintended or unexpected interpretation of the PHP program. To illustrate a SQL injection, consider the PHP program in Figure 5.

Consider the `$query` variable that contains the string `"SELECT id FROM users WHERE uname='$username'"` . The sub-string `'$username'` is interpreted as a variable by the PHP engine. Therefore, the PHP engine will replace `$username` with its value and reconstruct a new string. For instance, if the string value of `$username` is `"monkey OR 1=1"`, the string value of the variable `$query` is then `"SELECT id FROM users WHERE uname=monkey OR 1=1"`. This will result in the data for all users being extracted and returned, instead of just the single user's data, as the programmer intended.

One way to mitigate this attack is to "filter" or "sanitize" the input so that only "valid" data is accepted. Unfortunately it is difficult or even impossible to define sanitization functions that will be universally effective: each security sensitive operation needs a different sanitization functions. An important principle for defending against attacks is to ensure that "tainted data" or unsanitized data never reaches resources that can compromize the security of the application, which we will discuss later in the paper.

3.2.3 Initialization Issues

Early versions of PHP provided a feature whereby, when PHP was invoked from a Web server, the fields of any incoming HTML forms would be automatically extracted, and global variables with the field names would be assigned to the values of the fields. For instance, if a HTML form with fields "partNum" and "quantity" was received, then the global variables "$partNum" and "$quantity" would be set. This resulted in serious security problems in practice, because the fields of HTML forms are not controlled by the application developer, but can be set arbitrarily by the sender. For instance, an attacker could send a message containing the HTML form with fields "username" and "validated" with the result that those global variables will be given attacker-chosen values.

Unfortunately, many scripts are written assuming this behavior, so this behavior could not be removed. Instead, the language designers made this behavior optional, controlled by an initialization setting, "register_globals". However, this means that the language semantics is dependent upon a setting which is under installation control: the same script can be secure on one server but not another, due to this setting being different.

Another similar initialization issue is the "magic_quotes" functionality. In response to many security failures caused by SQL injection attacks, the designers of the language created the "magic_quotes" initialization property. When enabled, all input to a PHP program will be inspected and any "special" characters in it will be automatically quoted. The idea was that if special characters were quoted by the PHP runtime, then the PHP programmer would not have to do any validation of their own before passing the input as part of a SQL query. Unfortunately, this turned out to be overly optimistic: the quoting needed depends upon the particular use of the data, so that the automatic quoting not only was rarely sufficient, but prevented proper validation from being performed. The result is that some scripts will be secure only if magic quoting is disabled, while others will be somewhat secure only if it is enabled.

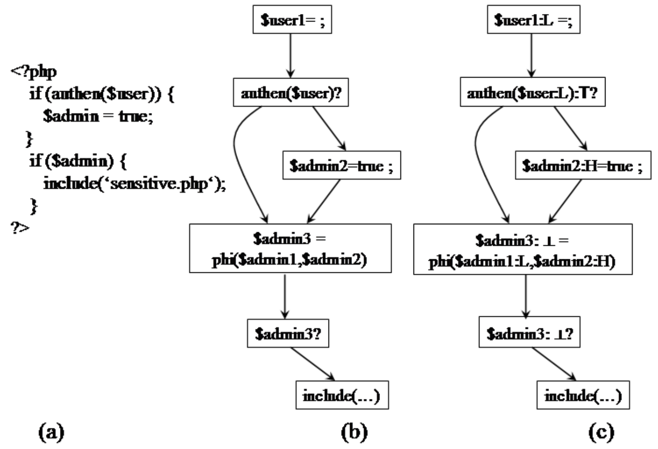
3.2.4 Dynamic Includes

As can be seen in Figure 7(a), include statements, like any other statement, can occur inside if-statements and functions, and their arguments can be arbitrary values, not necessarily constants. Programmers of most traditional languages might expect that this ability to include files determined dynamically at run-time will be little-used, since it might appear obviously beneficial to make it clear what code a script will execute. In reality, however, PHP actively encourages dynamically including files.

Many PHP scripts are intended for use by administrators or privileged users. If an attacker can cause one of these scripts to be executed unexpectedly, a security violation can result. If the target of an include statement can be affected by user input and this input is not properly validated, then such an attack can be mounted. From our perspective, dynamic includes also pose a problem for analyses.

```
s3: $ln = mysql_connect('localhost', 'user', 'password');
mysql_select("MYDATABASE", $ln) ;
while (?)  {
   s4 : mysql_query("CREATE DATABASE my_db",$ln)
   if (?) {
     mysql_close($ln) ;
    s5 : $ln := mysql_connect('myhost','my_user','my_password');
  }
}
              (a)
$ln1 = mysql_connect('localhost', 'user', 'password');
mysql_select("MYDATABASE", $l) ;
while (?)  {
   $ln2 = phi(ln1:o,ln4:o)
   s4 : mysql_query("CREATE DATABASE my_db",$ln2)
   if (?) {
     mysql_close($ln2) ;
     s5 : $ln3 := mysql_connect('myhost','my_user','my_password');
   }
  ln4:o = phi(ln2:o,ln3:o)
}
             (b)
```

Figure 6. (1) An example illustrating typestate and alias interaction, (b) corresponding TSSA form.



```
<?php
  if (authen($user)) {
    $admin = true;
  }
  if ($admin) {
    include('sensitive.php');
  }
?>
```

Figure 7. (a) A simple example (b) SSA form (c) TSSA form.

Whenever an include statement is encountered, all possible targets of the include have to be considered in order for the analysis to be sound. In the worst case, any file in the file system could be a potential target.

3.3 Data Flow Problems

In this section we highlight several data flow issues that affect security of PHP applications.

3.3.1 Resource Analysis

In PHP a *resource* variable holds a reference to an external resource. The type of a resource is called a resource type. Resource variable and type are ad hoc concept in PHP.[3] In PHP "resource" is a surface type and, because of PHP's dynamic typing, only type hints can be provided in code. One can use `is_resource($x)` to check if `$x` is a resource type, and `get_resource_type(resource $handle)` to returns the string representation of resource `$handle`. Since we are interested in protecting these resources, we focus many of our analyses on resources (e.g., typestate and taint analysis).

We use typestate verification techniques based on static single assignment (SSA) form for verifying correct usage of resources in PHP. Typestate verification consists of statically determining if a given program can execute operations on variables and objects that are not in a valid typestate. For instance, consider the example shown in Figure 6 (adopted from Field et al. [12]). The operation `f.read` is valid only if the typestate of `f` is $open$. Note that certain operations in a program can *alter* the typestate of a variable. The operation `f.close` changes the typestate of `f` from $open$ to $close$.

One of the hardest problems in precise typestate verification is the interaction between aliasing and typestate checking. Previous two-phase approaches, consisting of alias analysis followed by typestate analysis, can sometimes lead to imprecise typestate verification. To illustrate the imprecision, consider the example shown in Figure 6(a). Using alias analysis we can see that the reference `ln` at statement s4 can point to either database connection created program points s3 and s5. Using typestate analysis we can see that the connection created at s3 and s5 could be in a closed state at s4, indicating a possible typestate error. Unfortunately previous two-phase approaches are not be able to discover that `ln` can never point to a closed object at s4. Field et al. proposed a polynomial algorithm that integrates alias analysis and typestate analysis to derive a more precise typestate verification for certain classes of problems. The integrated approach consists of tracking both aliasing information and typestate information at every point in the program. We use a simpler sparse technique using typestate SSA (TSSA) form for typestate verification. The TSSA form for the above example is shown in Figure 6(b). The annotations such as `:o` indicate the "input" typestate at each statement. From the typestate annotation we can see that there are no typestate errors in the program. Our approach is inspired by optimistic sparse conditional constant propagation [34], and uses a two-phase approach for typestate verification.

Let T be the set of typestates. We use two distinguished typestates: $\top$ and $\bot$, that are ordered ($\sqsubseteq$) with respect to elements t of T as follows: $(1)\ \bot\ \sqsubseteq\ t$, $(2)\ t\ \sqsubseteq\ \top$, and $(3)\ \bot\ \sqsubseteq\ \top$. In other words, the set $T' = \top \cup \bot \cup T$ forms a lattice, with meet ($\wedge$) operation shown below (where $t, t' \in T$):

$$\begin{aligned} \top \wedge t &= t \\ \bot \wedge t &= \bot \\ \bot \wedge \top &= \bot \\ t \wedge t' &= \bot \text{ if } t \neq t' \\ &= t \text{ if } t = t' \end{aligned}$$

Intuitively, $\bot$ is an *undefined typestate* and it is an error to operate on variables whose typestate is $\bot$. The typestate $\top$ is an *undetermined typestate*. For simplicity, we assume that the set of typestate elements in T are not comparable with one another. However one can easily extend the framework described here by assuming a lattice structure for T. Notice that the typestate lattice defined above is very similar to a constant propagation lattice [34].

3.3.2 Vulnerabilities and Attacks

Understanding the flow of data with in a PHP program is a cornerstone for detecting security vulnerabilities. We are currently developing data flow analysis techniques to detect and prevent such security attacks. We have developed a new combined typestate and tainted flow analysis for detecting security vulnerabilities.

The best method that PHP programmers have of defending against attacks using register_globals is to ensure that they never make use of a possibly uninitialized variable. For this and other reasons, one of the major security problems is statically detecting the use of uninitialized values. This is made more difficult

[3] http://www.php.net/manual/en/resource.php gives the current resources supported in PHP.

by the fact that PHP variables can be explicitly uninitialized using the `unset` operation. Consider the example shown in Figure 7(a). The variable `$admin` is not initialized, and the input query `admin=1&user=guest` can trigger a security violation. We developed a sparse typestate analysis based on Static Single Assignment (SSA) form for detecting such uninitialized variables. The SSA form for the example is shown in Figure 7(b). We define two typestates `L` and `H` (for low security and high security labels). We associate typestates `L` to `$admin1` since it is not initialized, and associate typestate `H` to `$admin2` since it is initialized. We then propagate the two typestates to the ϕ-node and 'merge' them to get a new typestate $\perp$ for `$admin3`. We introduce a simple lattice structure for typestate propagation so that we can merge them at the ϕ-nodes. The lattice element $\perp$ essentially is an undefined typestate and a sensitive operation should not operate or depend on undefined typestates. Since `include('sensitive.php')` is control dependent on an 'undefined' predicate `$admin3?` we report a security error. In PHP a variable can be explicitly `unset`, and any further use of such unset variable can trigger a security vulnerability. Using our sparse type state analysis we can track the use of such unset variables.

3.3.3 PHP Runtime Analysis

We are implementing simple runtime taint analysis that is similar to the basic Perl/Ruby tainting support: when tainted data is used in specific contexts such as shell or sql commands, the run-time system raises an exception [**?**].[4] Programmers may turn on run-time taint checks to find out if they use tainted data in sensitive contexts. And when an application is written with taint support in mind, website operators may turn on run-time taint checks to provide an additional barrier against penetration. Our system can work with unmodified third-party extensions.

Previous approaches to PHP taint analysis explicitly permit the use of tainted data in, for example, html, shell, or sql commands. When such a command contains "forbidden" characters, these systems either "neuter" the command before execution, or they abort the command altogether. The main problems with automatic cleansing are not technical but psychological:

Education: Automatic cleansing systems don't make programmers aware that network data is inherently untrustworthy[15]. Instead, they teach the exact opposite: don't worry about data hygiene. This leads to a false sense of security, because automatic cleansing of shell or sql commands solves only part of the problem.

Expectation: Automatic cleansing systems have to be perfect. For example, if the safety net catches some but not all cross-site scripting or SQL injection attacks, then the system has a security hole.

These two problems are probably the major contributors to the demise of PHP's ill-fated "safe mode" feature. There are some technical problems too with automatic cleansing approaches: (1) The automatic cleansing safety net has to keep track of exactly which characters in a substring are derived from untrusted input, so that it can later recognize malicious content in, for example shell or sql commands. (2) Different contexts need different definitions of what could be "malicious" content. In particular, providers of PHP extensions need to implement their own special-purpose code to detect or neuter untrusted substrings in inputs, or to mark untrusted substrings in result values.

We have implemented a simple runtime taint analysis by modifying the Zend PHP runtime engine. The basic idea of runtime taint analysis is to mark certain external inputs as tainted, and to disallow the use of tainted data in certain operations that change PHP's own state (e.g., include, eval, etc.), or that access or modify external state (e.g., file access, network access, HTML output, shell or database command). Although the exact details of what is tainted and what is forbidden may evolve over time, the general mechanism is well understood.

The following is a high-level view of what would happen when taint checking is turned on at run-time:

- Each ZVAL (the PHP internal representation of a string, number or other data object) is marked tainted or not tainted. That is, we don't taint individual characters within substrings. In the future, we may explore the possibility of multiple shades of taint.
- Primitives and functions such as `echo`, `eval`, or `mysql_query()` detect whether they are given tainted input. Depending on run-time settings, either the program terminates with a run-time error, or it proceeds after logging a warning.
- The PHP run-time system propagates taintedness across expressions. If an input to an expression is tainted, then the result of that expression is tainted too. There are exceptions to this rule: some primitives and functions always produce untainted results, and some always produce tainted results.
- The PHP application programmer untaints data by explicit assignment with an untainted value. For example, the result from htmlentities() or `mysql_real_escape_string()` is not tainted. This is sufficient for our purpose: to help programmers, by pointing out that they need to explicitly sanitize user data.

Our approach is the result of trade-offs between usability, implementation cost, maintenance cost, run-time cost, and impact on third-party extensions [25]. For example, we chose against fine-grained tainting not only for performance reasons but also because it would require invasive changes to third-party extensions; and we chose against the use of conditional expressions to untaint variables because we can't reliably determine if the intention of a test is to sanitize input.

Our goal is to add run-time taint checking to PHP, not to provide a sandbox for the execution of hostile code. It is just a tool to help programmers find out what data needs to be sanitized. It avoids changes to third-party extensions, and is turned off by default. It is therefore completely backwards compatible with earlier versions of the PHP runtime.

4. JavaScript Analysis

With the advent of DHTML and Ajax, JavaScript has become an increasingly important programming language. Similar to popular server-side scripting languages such as PHP, JavaScript is dynamically typed. In addition, JavaScript allows types themselves to be changed at runtime. It also provides a variety of ways to evaluate arbitrary strings as code during runtime. This is further complicated by the strong coupling with the other components of DHTML, i.e., (X)HTML[5], CSS[6] and the underlying DOM[7] of the browser environment, These provide various ways for dynamic code evaluation and self-modification, i.e., using (dynamically generated) event handlers. Similarly, constructs such as `with` and closures makes the determination of scopes non-trivial. More detailed discussion on how to capture self-modification and general issues with related static analysis can be found, e.g., in Yu et al [36] and Dolby [10], respectively.

[4] `http://perldoc.perl.org/perlsec.html`

[5] http://www.w3.org/MarkUp/

[6] http://www.w3.org/Style/CSS/

[7] http://www.w3.org/DOM/

Our interest in analyzing JavaScript stems from the fact that JavaScript code exhibits considerable control over the browser. Due to weak security-models in browsers, this often leads to lack of separation and opens the door for a variety of attacks. This is in particular dangerous in situations such as portals or mashups where sensitive information from multiple and potentially mistrusting or malicious information providers is aggregated on a single page. For example, in a mashup providing a one-stop car purchase portal combining information from different dealers, insurance companies and the user's bank, dealers should not be able to modify each others car prices nor should they be able to spy on a user's bank account. Even for environments such as enterprise portals, where information comes arguably from the same trust domain, the sensitivity of salary data and the like makes isolation a necessity to provide security-in-the-depth and to protect against programming errors such as cross-site scripting (XSS) attacks.

Our approach currently focuses on portals and is roughly comprised of the following steps: (1) For each portlet fragment, we check a number of syntactic constraints[8] and mark each fragment with its corresponding security domain by wrapping it in a special `div` element `portlet-root`; (2) After aggregation of the portlet fragments into a whole HTML page, we convert the page into an equivalent JavaScript program, i.e., one which renders the exact same content; (3) Together with an object model of the browser's DOM, also defined in JavaScript, we perform a static analysis of isolation and integrity constraints using a predecessor of IBM Research's WALA[9] libraries; (4) Finally, we rewrite certain code constructs, e.g., to separate name spaces. Converting everything into JavaScript allows for a unified analysis approach. For instance, having converted the HTML into equivalent JavaScript, the analysis engine automatically constructs an object model for the DOM tree for the page, which is used to perform precise alias analysis of DOM objects. Uniformly using JavaScript also enables easy customizations to particular browsers which are usually not 100% standards-conformant and provide various security-sensitive extensions.

Two examples of constraints we perform in step (3) are the restriction of DOM tree walking of a portlet to its domain and the protection of the integrity of system code.

To restrict tree-walking, we perform a pointer analysis on all operations that climb up in the tree — descending is always safe – and make sure that the points-to set does not include the `portlet-root` element. Together with the constraints guaranteed by construction in step (1), the name space separation ensured by step (4), this will guarantee the invariant that a portlet can only access its own DOM elements.

Of course, above algorithm also relies on the integrity of the system libraries, which brings us to the second example of analysis. To maintain code integrity, we have to insure that no user code can redefine system code or objects. Furthermore, we have to make sure that system functions only receive objects as parameters which meet the expectation, i.e., the parameter to the method `appendChild` of `DOMNode` must be a proper `DOMNode` generated by `DOMDocument.createElement` or equivalent. This is necessary to prevent a rogue element from subverting the browser "inside-out". To achieve this, an information-flow lattice has to be enforced to prevent user information from flowing into system code. Obviously, given the multiple ways JavaScript allows to aliasing functions and variables, we have to be careful to do appropriate alias analysis.

So far we mostly side-stepped the issue of runtime code evaluation: We restrict executed code in event handlers to calls to statically fixed functions and ban `eval` in its various incarnations. `eval` is mostly unnecessary and indicates the presences of bad coding; even for handling of JSON[10] objects, a common use-case for `eval`, a JSON parser instead of `eval` is preferable from a security-in-the-depth perspective. One could, though, allow it by using code rewriting techniques such as in BrowserShield [30].

Of high importance, of course, is performance. The static analysis poses rather hefty computation cost on the server side. While various caching optimizations could improve the situation incrementally, it is an interesting research question whether the analysis could be done directly on the generation code, e.g., JSP or PHP, and be done correspondingly offline. Alternatively, the code generation could generate additional information facilitating the analysis akin to proof carrying code [22]. For example, the original code generation could be based on a language such as Java as used in GWT[11] and ensure partial guarantees such as static type-safety and correspondingly simplify runtime analysis. This would also be helpful if the component is generated remotely, e.g., via WSRP [20].

Our work currently focuses on verification in the deployment and runtime system. However, it also seems beneficial to expose similar JavaScript analysis directly to the developers to inform them early about violation of our imposed restrictions, help them in restructuring their code correspondingly and complement existing code assists and best practice verification such as [10] and jslint[12].

5. An Integrated Analysis Framework

Over the past few years we have used various static analysis framework as a means for checking and understanding security problems and issues in Java, PHP, and JavaScripts. We want to provide the same look and feel for security checking and understanding of code for Java, PHP, Javascript, and other languages. To this end, we are currently building an Integrated Analysis Framework (IAF) that is based on extending Eclipse Development Tool (EDT) models such as Java Development Tool (JDT), PHP Development Tool (PDT), and JavaScript Development Tool (JSDT). Currently, JDT model is quite mature, whereas PDT model and JDT models are still under development.

Before diving into our IAF, we will first present the current JDT Core Java Model (CJM) and Abstract Syntax Tree (AST) or Document Object Model (DOM). CJM is a hierarchical model, consisting of core elements such as IJavaProject, IType, IMethod, etc. Figure 8 illustrates the core elements of CJM. `IWorkspace` is the root of the heirarchy and from which we can drive other core elements. The following is a snippet of code to get handle to Java compilation unit model (keep in mind JDT CJM provides several ways to get an handle to a `ICompilationUnit`).

```
IWorkspaceRoot wsRoot =
          ResourcesPlugin.getWorkspace().getRoot();
IProject project = wsRoot.getProject("MyProject");
project.open(null);
IJavaProject myProject = JavaCore.create(project);
IType myType = javaProject.findType("MyClass");
ICompilationUnit myICU  = myType.getCompilationUnit();
```

Given a handle to `ICompilationUnit` one can start parsing it to construct an AST for that compilation unit as follows:

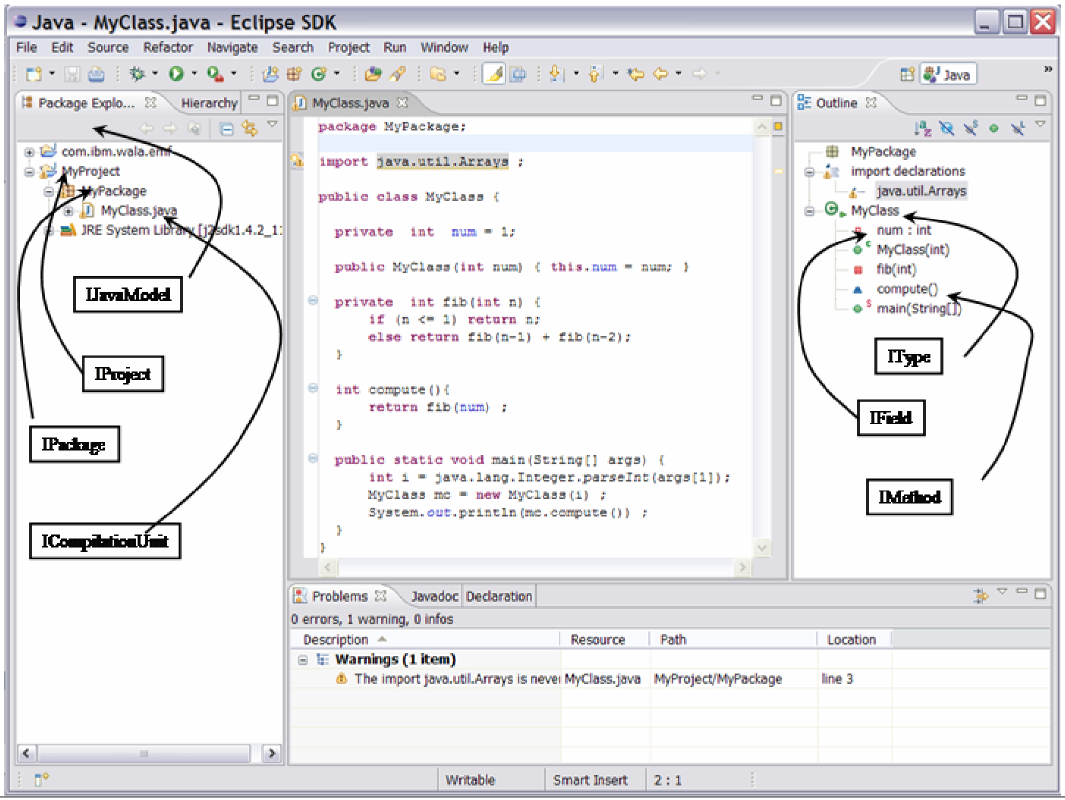
[8] For example, JSR 186 [1] would not prevent two malicious portlets to wrap a form element of a third good portlet with another form and hence hijack any information submitted from the good portlet even when JavaScript would be disabled; an attack which is possible with at least one commercially available portal server.

[9] http://wala.sourceforge.net/

[10] http://www.json.org/

[11] http://code.google.com/webtoolkit/

[12] www.jslint.com



```
package MyPackage;

import java.util.Arrays ;

public class MyClass {

  private  int  num = 1;

  public MyClass(int num) { this.num = num; }

  private  int fib(int n) {
      if (n <= 1) return n;
      else return fib(n-1) + fib(n-2);
  }

  int compute(){
      return fib(num) ;
  }

  public static void main(String[] args) {
      int i = java.lang.Integer.parseInt(args[1]);
      MyClass mc = new MyClass(i) ;
      System.out.println(mc.compute()) ;
  }
}
```

```
0 errors, 1 warning, 0 infos
Description                              Resource       Path                  Location
Warnings (1 item)
  The import java.util.Arrays is never   MyClass.java   MyProject/MyPackage   line 3
```

Figure 8. An example illustrating Eclipse Java model

```
ASTParser parser = ASTParser.newParser(AST.JLS3);
parser.setSource(unit);
parser.setResolveBindings(true);
CompilationUnit myCU = parser.createAST(null);
```

In the above code snippet, `CompilationUnit` is an AST node, and extends the root `org.eclipse.jdt.core.dom.ASTNode` of the AST/DOM model. One can use `ASTVisitor` interface to visit nodes of the AST (and perform operations on the AST nodes). `ASTParser`, `ASTNode`, and the `ASTVisitor` forms the core interfaces for manipulating Java sources in Eclipse, and can be used in many source code manipulation applications (such syntax highlighting). We are currently building our IAF using JDT core elements.

Our PHP model is deliberately designed to be as similar to Java's as possible, in order to ease development of language-independent code. As with Java's model, parser, node and visitor objects are provided and form the core of our manipulations. Significant differences, naturally, do arise. In Java type information is determined statically, and can be easily obtained and made available in the AST. Since in PHP not only is type information not static, but even the files which constitute the program not statically determinable, the PHP model cannot provide the same level of information initially. Instead, later analysis passes annotate the AST nodes with such information. Support for PHP has been integrated into Rational's Code Review functionality, so that when analyses discover possible flaws in PHP code, they can be easily displayed and referenced in the editor, as shown in Figure 4

Figure 9 illustrates the basic flow of our framework. Our PDT core model and PDT DOM/AST model `<<adopt>>` the core concepts of the JDT core model and JDT DOM/AST model, respectively.[13]

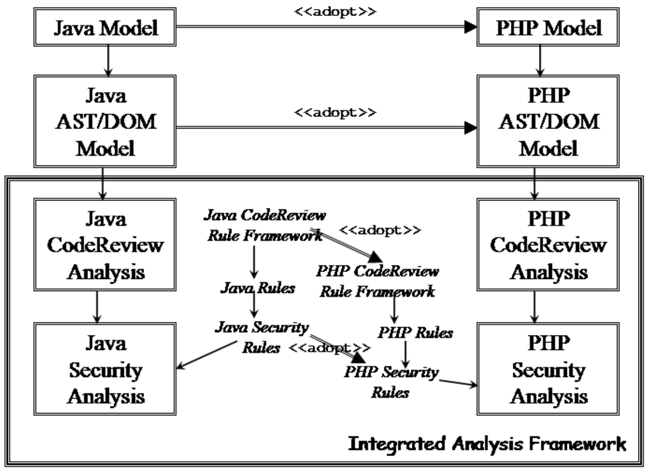


Figure 9. Basic elements of SCUT

We have made several design choices for implementing IAF based on the following observation of the Eclipse core models:

[13] At present we are mostly focusing on PHP and Java tools, and we hope to do the same for JavaScript analysis in the near future.

- Our current focus is on analyzing source code, and so we do not directly analyze bytecode (one can easily use a decompiler to reconstruct the source code from bytecodes).
- Conserving memory usage is very important in our framework since memory footprint is critical to adoption by the development community at large. Therefore it is very important not to create unnecessary data structure that could increase memory usage. Eclipse AST/DOM provides infrastructure for generating abstract syntax tree. We perform most of our intraprocedural analysis over AST, using concepts from attribute grammars that was developed over 30 years ago. In contrast, the WALA and JaBA frameworks create independent data structures, such as WALA and JaBA IRs, that exist independently of the Eclipse AST. This creates unnecessary duplication and so consume more memory than needed.
- From an end user or application developer perspective it is important that an analysis framework can map analysis output back to the source code and other artifacts structures. The Eclipse platform provides several other structures that relate various assets, including source code, requirement models, test cases, etc. By implementing IAF as part of Eclipse core extensions we benefit from having access to these other structures within Eclipse, and this allows us to create a more user friendly security tool.
- It is not practical to do whole program analysis, given that we are targeting to analyze extremely large programs. At any instance we can only load a part of the program, and so our analysis should be incremental and modular. Unfortunately, for many non-trivial program analyses such as typestate analysis and context sensitive analysis, devising modular and incremental analysis can often lead to unsoundness or imprecise results. For many libraries and plug-in modules that rarely change we can perform off-line analysis and store analysis summaries in a database repository. We are currently designing our analysis summary storage format that will be stored in Apache Derby database. From an end user perspective we want our IAF load and start time for these libraries to be no more than a moderate increment over the the base load and start time of the Eclipse platform.

Our IAF is still under active development and we hope to publish more on our experience with IAF in the near future.

6. Related Work

During recent years there has been an explosion of interest in both program analysis and security, and the current body of work is therefore too large to give more than a cursory overview. In this section we'll describe closely relevant work in the areas of security models, web application vulnerability detection, and program analysis.

6.1 Security Models

Before it is possible to detect possible security vulnerabilities, it is necessary to define an intended security policy. The classic works upon which such policies are based are the Bell-LaPadula model (for confidentiality) [4], and the Biba model (for integrity) [5].

These models classify both people (or users or process) and information (or objects or resources) into different levels of trust and sensitivity that represent security classifications. Typically users and processes are given *clearance level* that indicates the level of trust, and information have *classification level* that indicates the sensitivity of the information. Before a user (or a process) is allowed to access information at a particular security level, the user should have clearance to access the information at that level.

A simple security model with two security levels L and H, can be defined using a lattice with partial order $L \sqsubseteq H$. Parts of program elements can be labeled with either H or L. For confidentiality, information from H labeled elements cannot flow into L labeled elements [4], and for integrity information from L labeled elements cannot flow in H labeled elements [5].

Our analysis based on TSSA is a *hybrid analysis* that includes both typestate analysis and information flow analysis. It is important to remember that traditional typestate analysis do not use lattice more, and whereas traditional information flow analysis do not use typestate model.

6.2 Security Vulnerability Detection

Bisbey et. al, [6] proposed in 1978 to use what we would now call program analysis to detect security vulnerabilities in source code. Unfortunately, the technology available at the time was not adequate for this proposal to be realized. Since then, much work has been done in this area, including such work as [8, 3, 11, 37, 31].

In [29] and [28], Reimer et al. detect programming errors in J2EE Web applications, and apply their tool to large, commercial applications. Their system allows application-specific rules to be constructed from a set of general templates.

Huang et al. [15] use information flow based type systems for detecting flow insensitive security bugs. They also use typestate analysis for detecting flow sensitive bugs. The sections in the code that are considered by the static analysis to be vulnerable are instrumented at the runtime. The vulnerabilities that they consider are XSS and SQL injection. Huang et al. do not handle pointer reference, which simplifies much of their analysis.

Pixy [16, 17], is a tool for detecting security vulnerabilities that is based on information flow taint analysis. Pixy analysis is flow sensitive and limited context sensitive. Pixy analysis include literal analysis and alias analysis, but limited to non-object-oriented features of PHP.

Nguyen et al. [23] and Pietraszek and Berghe [24] use fine grain dynamic taint analysis for tracking tainted data at substring level. They then use fine grain taint analysis to detect injection attacks. Both approaches modify the PHP interpreter to track taint information for string data.

Minamide [21] approximates the string output of PHP programs using a context-free grammar. The approach does static checking of properties and validation of the Web pages that are generated dynamically, and use the result to detect cross-site scripting vulnerability. Once again Minamide does not deal with objects and references and ignore the relationship between arguments and return value inside functions.

Su and Wassermann [33] first define a formal model of command injection attacks in web application. They then propose an algorithm for preventing these attacks based on context-free grammars and compiler parsing techniques. They also assume that the input that gets propagated into the database query or the output document changes the syntactic structure of the query or document.

Xie and Aiken [35] use a symbolic execution to model dynamic features of PHP inside basic blocks . Block summaries are then used to hide complexity of symbolic execution and from intra- and inter-procedural analysis.

SQLrand [7] avoid SQL injection attacks by separating commands encoded in the program code from user input data. It is based on the assumption that keywords of SQL statements are fixed as constants in the script code and should not be given by the user input. Source code is instrumented by replacing all keywords with encoded versions. The modified statements are intercepted by an SQL proxy, which filters illegal SQL encoded statements.

In [18], instructions are encrypted when they are stored in memory and decrypted before being loaded into the processor. Without knowing the key, code that is changed by malicious users would be incorrectly decrypted and the application would crash.

6.3 Typestate Analysis and SSA Form

The original work on typestate focused on finding flow-sensitive errors [32]. DeLine and Fahndrich extended the classical typestate theory to objects [9]. They use pre- and post-conditions to express allowed transition rules between the typestates of the object, and typestates to express predicates (or constraints) over the objects concrete state, which includes the field states. They handle aliasing by using adoption and focus operations with linear type system. The type checker assumes must-alias properties for a limited program scope, and so are able to handle strong updates during typestate transitions. Aiken et al. enable strong updates during typestate transition by using an inference algorithm for inferring restricted and confined pointers. Foster et al. present a mechanism to add type qualifiers as first class element in C language [13]. Type qualifier are similar to typestates and can be used for detecting flow-sensitive type errors. Fink et al. present a staged approach and combine some of the previous work on typestate analysis. The resulting framework for verification of typestate properties is a staged verification system in which faster verifiers run at earlier stages which reduce the workload for later, more precise, stages. Our work differs from most previous work on typestate analysis in that we combine sparse evaluation and typestate analysis to construct TSSA form and SPIG. Our typestate verification using TSSA form is inspired from Wegman and Zadeck optimistic constant propagation algorithm [34]. We introduce a lattice structure for typestates, and use properties of SSA form and optimistic constant propagation algorithm to obtain a faster and precise typestate verification in the presence aliasing. We are currently extending the sparse typestate analysis for non-shallow programs.

Our sparse property implication analysis and SPIG can be applied to solve many data flow problems that have property implication property. We are currently using SPIG for taint analysis and for interprocedural typestate verification.

7. Conclusions

Developing security checking and understanding tool is a challenging task. It is not only important to come up with the right analysis techniques to solve security problems, it is also important to ensure the analysis is well integrated into the development process and be made transparent to the application developer. In this paper we have only touched upon the “tip of the iceberg” in terms of our experience with building usable security tools. We are continuing to expand our repertoire of security related tasks that can help application developer to build secure software for Java, PHP, JavaScript, and other languages. Also, our objective is to give the same user experience when developing with multiple languages.

References

[1] Alejandro Abdelnur and Stefan Hepper. Java Portlet specification. Java Specification Requests 168, Java Community Process, October 2003.

[2] The OSGi Alliance. *OSGi Service Platform, Release 3*. Ios Pr, Inc, 2003.

[3] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes, May 2002. In IEEE Symposium on Security and Privacy, Oakland, California.

[4] E. D. Bell and L. J. LaPadula. ”Secure Computer Systems: Mathematical Foundations and Model”. In *Technical Report M74-244, Mitre Corporation*, 1973.

[5] K. J. Biba. ”Integrity Considerations for Secure Computer Systems”. In *Technical Report M74-244, Mitre Corporation*, 1975.

[6] Richard Bisbey and Dennis Hollingworth. Protection analysis: Final report. Technical Report ISI/SR-78-13, Information Sciences Institute, University of Southern California, Marina del Rey, CA, May 1978. `http://csrc.nist.gov/publications/history/bisb78.pdf`.

[7] S. Boyd and A. Keromytis. ”SQLrand: Preventing SQL injection attacks”. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, 2004.

[8] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, Washington, DC, November 18–22, 2002.

[9] R. DeLine and M. Fahndrich. Typestates for objects. In *18th European Conference on Object-Oriented Programming*, 2004.

[10] Julian Dolby. Using static analysis for IDE’s for dynamic languages. In *The Eclipse Languages Symposium*, October 2005.

[11] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, 2002.

[12] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *In Proceedings. of Static Analysis Symposium,*, volume 2694, pages 439–462. LNCS Springer, 2003.

[13] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 2002.

[14] Li Gong and Roland Schemers. Implementing protection domains in the java^{tm} development kit 1.2. In *NDSS*. The Internet Society, 1998.

[15] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. ”Securing Web Application Code by Static Analysis and Runtime Protection”. In *WWW ’04: Proceedings of the 13th International Conference on World Wide Web*, 2004.

[16] N. Jovanovic, C. Kruegel, and E. Kirda. ”Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities”. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[17] N. Jovanovic, C. Kruegel, and E. Kirda. ”Precise Alias Analysis for Static Detection of Web Application Vulnerabilities”. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, 2006.

[18] G. S. Kc, A. D. Keromytis, and V. Prevelakis. ”Countering Code-Injection Attacks with Instruction-Set Randomization”. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[19] Larry Koved, Marco Pistoia, and A Kershenbaum. ”Access Rights Analysis for Java”. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[20] Alan Kropp, Carsten Leue, and Rich Thompson. Web Services for Remote Portlets Specification. Oasis standard, OASIS, August 2003. Version 1.0.

[21] Y. Minamide. ”Static Approximation of Dynamically Generated Web Pages”. In *WWW ’05: Proceedings of the 14th International Conference on World Wide Web*, 2005.

[22] George C. Necula. Proof-carrying code. In *24th Symposium on Principles of Programming Languages (POPL)*, pages 106–119, Paris, France, January 1997. ACM Press.

[23] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. ”Automatically Hardening Web Applications Using Precise Tainting”. In *Proceedings of the 20th IFIP International Information Security Conference*, 2005.

[24] T. Pietraszek and C. V. Berghe. ”Defending against Injection Attacks through Context-Sensitive String Evaluation”. In *Proceedings of the Recent Advances in Intrusion Detection (RAID)*, 2005.

[25] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID*, pages 124–145, 2005.

[26] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. ”Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection”. In *Proceedings of 19th European Conference on Object-Oriented Programming*, 2005.

[27] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in java. In *Proceedings of CASCON 2000*, 2000.

[28] Darrel Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Julian Dolby, Aaron Kershenbaum, and Larry Koved. Validating structural properties of nested objects. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[29] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert Johnson, Aaron Kershenbaum, and Larry Koved. Saber: Smart analysis based error reduction. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2004.

[30] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, November 2006.

[31] Benjamin Schwarz, Hao Chen, David Wagner, Jeremy Lin, Wei Tu, Geoff Morrison, and Jacob West. Model checking an entire linux distribution for security violations. *acsac*, 0:13–22, 2005.

[32] R. Strom and S. Yemini. Typestate: a programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1), Jan 1986.

[33] Z. Su and G. Wassermann. ”The Essence of Command Injection Attacks in Web Applications”. In *33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2006.

[34] Dave Thomas. *Programming Ruby: The Pragmatic Programmer's Guide*. Pragmatic Bookshelf, 2004.

[35] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

[36] Y. Xie and A. Aiken. ”Static Detection of Security Vulnerabilities in Scripting Languages ”. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[37] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *34st Symposium on Principles of Programming Languages (POPL)*, pages 237–249. ACM Press, January 2007.

[38] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the USENIX Security Symposium*, August 2002.