Static Detection of Resource Contention Problems in Server-Side Scripts

Yunhui Zheng Xiangyu Zhang
Department of Computer Science
Purdue University
{zheng16,xyzhang}@cs.purdue.edu

***Abstract*—With modern multi-core architectures, web applications are usually configured to serve multiple requests simultaneously by spawning multiple instances. These instances may access the same external resources such as database tables and files. Such contentions may become severe during peak time, leading to violations of atomic business logic. In this paper, we propose a novel static analysis that detects atomicity violations of external operations for server side scripts. The analysis differs from traditional atomicity violation detection techniques by focusing on external resources instead of shared memory. It consists of three components. The first one is an interprocedural and path-sensitive resource identity analysis that determines whether multiple operations access the same external resource, which is critical to identifying contentions. The second component infers pairs of external operations that should be executed atomically. Finally, violations are detected by reasoning about serializability of interleaved atomic pairs. Experimental results show that the analysis is highly effective in detecting atomicity violations in real-world web apps.**

***Keywords*-php; resource contention; static analysis; constraint solving**

I. Introduction

Web is becoming an important computation platform. Many daily tasks such as online shopping, social networking, data storage, and document processing are carried out by web applications (apps). An important trend of web apps is to leverage modern multi-/many-core architectures. However, the incompatibility between the sequential programming model of server side scripts and the concurrent execution model configured in servers to leverage multiple cores may lead to harmful contentions on external resources.

Particularly, for design simplicity, popular server side scripting languages such as PHP do not support threading. As a result, web app developers tend to think that they are performing sequential programming, and they are at a safe distance from troubles caused by concurrency. However, it is not true under the most popular LAMP (Linux, Apache, MySQL and PHP) web app environment, in which Apache is the web server and server side scripting support is provided through PHP. PHP scripts communicate with a database system (e.g. MySQL) or the file system to provide services. To optimize server performance, Apache is usually configured to run in the MPM (Multi-Processing Modules) mode, which allows multiple instances of a server side script being spawned, each serving an independent client request. Although these instances do not share variables in memory as variables are by-default local to a thread, they do share external resources such as database tables and files. Such sharings nonetheless cause race conditions.

Many web application failures are essentially due to such race conditions. For example, in ProductCart[1] and OpenCart[2], they led to products being over-sold, resulting in back-orders. In Drupal[3] and Joolam BibTeX[4], they caused critical runtime exceptions and file corruptions.

Unfortunately, there are mis-perceptions that induce developers to overlook such problems. One of such is that race conditions on external resources are well guarded by the internal protection in the database engine and the file system, otherwise any regular sequential programs would be vulnerable as multiple instances of the same program may share the same resources. The observation is that while it is unlikely that multiple instances of a regular sequential program are running simultaneously, the likelihood is much higher for a web app as it aims to deliver the same service to many users. In other words, we argue that the problem is substantially exacerbated in the context of web apps.

Existing solutions for detecting concurrency errors fall short. Most race detection [27], [24], [22], [23] and atomicity violation detection [14], [19], [26], [18] techniques rely on analyzing synchronization primitives. However such primitives are simply not provided by server side scripting languages. Most existing techniques work by reasoning about the interleavings of accesses from different threads on shared memory. However, scripts usually do not share memory. Sharing is on external resources instead. To reason about the effect of interleaving external resource accesses, one needs to model these external operations, e.g. queries to the database engine. Furthermore, these operations cannot be simply modeled as reads or writes on shared objects. For example, the DELETE and INSERT database queries have more complex semantics that cannot be described as simple reads/writes. Identifying the resources being accessed also poses a unique challenge as they are usually denoted in the program as concatenations of strings and variables. One has to reason about the equivalence of such representations to determine if the same resources are being accessed.

There has been some recent work specifically focusing on concurrency errors in web apps. In [37], researchers proposed a technique to detect race conditions caused by the asynchronous AJAX requests. The technique is unfortu-

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

nately not applicable as it only works on client side through JavaScript and does not reason about external resources.

A dynamic analysis was proposed to detect race conditions caused by interactions between a web app and the database [25]. It collects SQL query traces and checks whether a specific interleaving pattern occurs in traces. It demands a proper input and a proper interleaving to expose a bug, which are in general hard to acquire due to non-determinism. The technique also completely neglects program semantics, leading to false positives.

In this paper, we propose a whole program interprocedural static analysis that detects atomicity violations regarding external resources in PHP scripts. The overview of our technique is presented in Fig. 1. It takes the PHP code of a web app and translates it to C. The C code is analyzed in three phases. The first phase is the resource identity analysis that determines whether multiple operations access the same external resource, which is critical to identifying contentions. The second phase infers pairs of external operations that should be executed atomically. In phase three, violations are detected by reasoning about serializability of interleaved atomic pairs.

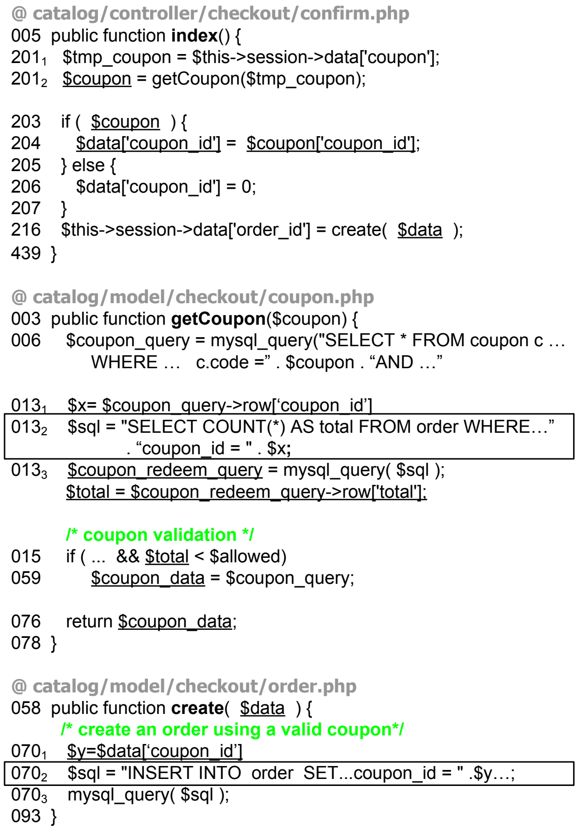
We make the following contributions.

- We develop a context- and path-sensitive interprocedural static analysis to automatically detect atomicity violations on shared external resources in PHP code.
- We develop the *resource identity analysis*, a technique to reason about the equality of resources being accessed by external operations. It is interprocedural and path sensitive, leveraging a SMT solver.
- We propose a novel way of statically inferring atomic regions in PHP code, which avoids demanding the developer to annotate such regions, or guessing them from synchronization primitives or dynamic traces [14], [19], [34]. It leverages the observation that atomicity properties in PHP programs are more amenable for automatic inference compared to general concurrent programs in C++ and Java, as PHP developers usually follow a sequential programming paradigm.
- We develop expressive abstractions for external operations. They go beyond the read and write abstractions for shared memory accesses. We define atomicity violations based on these abstractions.
- We evaluate the technique on real world web apps. The results show that it is highly effective, detecting 113 real bugs. Some of them have financial impacts.

II. Motivating Examples

A. Atomicity Violation in OpenCart 1.4.9.4

We first use a bug in a recent version of OpenCart (v1.4.9.4) to motivate our technique. OpenCart is an open source shopping cart application. It was reviewed as one of the best open source e-commerce platforms [5] in 2010. A user of OpenCart can be either a normal user or an administrator. The administrator can add, modify or delete products and coupons. A normal user can place orders with the option of applying coupons. The problem to be demonstrated allows illegal coupon usage. In particular, when multiple users place orders concurrently, which is very likely to happen in peak time, a coupon can be applied for arbitrary number of times, ignoring its use limit.



```
@ catalog/controller/checkout/confirm.php
005  public function index() {
201_1   $tmp_coupon = $this->session->data['coupon'];
201_2   $coupon = getCoupon($tmp_coupon);

203    if (  $coupon  ) {
204       $data['coupon_id'] =  $coupon['coupon_id'];
205    } else {
206       $data['coupon_id'] = 0;
207    }
216    $this->session->data['order_id'] = create(  $data  );
439  }

@ catalog/model/checkout/coupon.php
003  public function getCoupon($coupon) {
006     $coupon_query = mysql_query("SELECT * FROM coupon c …
            WHERE …   c.code =” . $coupon . “AND …”

013_1    $x= $coupon_query->row[‘coupon_id’]
013_2    $sql = "SELECT COUNT(*) AS total FROM order WHERE…”
                    . “coupon_id = " . $x;
013_3    $coupon_redeem_query = mysql_query( $sql );
         $total = $coupon_redeem_query->row['total'];

         /* coupon validation */
015      if ( ...  && $total < $allowed)
059          $coupon_data = $coupon_query;

076      return $coupon_data;
078  }

@ catalog/model/checkout/order.php
058  public function create(  $data  ) {
        /* create an order using a valid coupon*/
070_1    $y=$data[‘coupon_id’]
070_2    $sql = "INSERT INTO  order  SET...coupon_id = " .$y…;
070_3    mysql_query( $sql );
093  }
```

Figure 2. Code snippet from OpenCart v1.4.9.4. The two rectangled database queries may not execute atomically while they should. The update query is (transitively) dependent on the select query through the underlined variables. For readability, we normalize the code snippet by breaking some statements into sub-statements, described by the subscripts.

Fig. 2 shows the relevant code snippet. Function *index*() in *confirm.php* validates a coupon by calling *getCoupon()* at line 201 and then places the order by calling *create()*.

In *coupon.php*, function *getCoupon*() dispatches two queries at lines 6 and 13. The first query is to retrieve the coupon information from database. The second query is to determine the number of coupon uses by accessing table *order*. If the coupon has reached its limit, variable *$coupon_data* holds the *FALSE* value at line 76, which is returned to method *index()*, indicating expiration of the coupon. Otherwise, the coupon details such as discount are loaded to *$coupon_data* and returned. Finally, at line 70 in function *create*() in *order.php*, an order is placed by inserting a record to table *order*, with the same coupon id.

The bug manifests itself when multiple users apply the same coupon in the mean time. To simplify discussion,

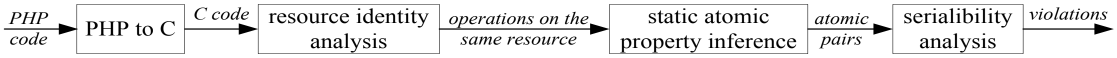
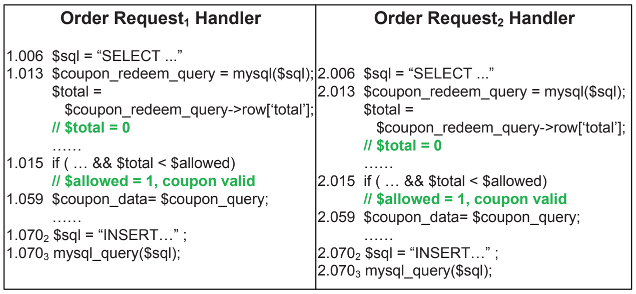
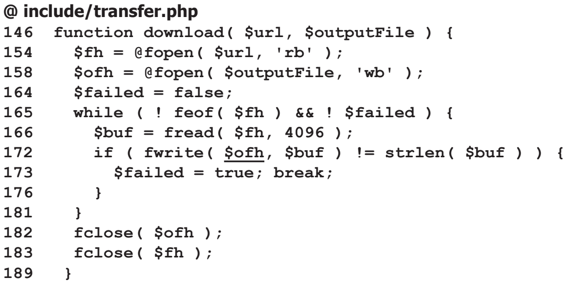

PHP code
PHP to C
C code
resource identity analysis
operations on the same resource
static atomic property inference
atomic pairs
serialibility analysis
violations

Figure 1. Overview of our technique.



Order Request$_1$ Handler	Order Request$_2$ Handler
1.006 `$sql = "SELECT ..."`	
1.013 `$coupon_redeem_query = mysql($sql);`	2.006 `$sql = "SELECT ..."`
`$total =`	2.013 `$coupon_redeem_query = mysql($sql);`
`$coupon_redeem_query->row['total'];`	`$total =`
`// $total = 0`	`$coupon_redeem_query->row['total'];`
……	**`// $total = 0`**
1.015 `if ( … && $total < $allowed)`	……
`// $allowed = 1, coupon valid`	2.015 `if ( … && $total < $allowed)`
1.059 `$coupon_data= $coupon_query;`	**`// $allowed = 1, coupon valid`**
……	2.059 `$coupon_data= $coupon_query;`
1.070_2 `$sql = "INSERT…" ;`	……
1.070_3 `mysql_query($sql);`	2.070_2 `$sql = "INSERT…" ;`
	2.070_3 `mysql_query($sql);`

Figure 3. The buggy interleaving for the OpenCart example. Symbol 1.013 means line 13 in the 1st thread.



```
@ include/transfer.php
146  function download( $url, $outputFile ) {
154    $fh = @fopen( $url, 'rb' );
158    $ofh = @fopen( $outputFile, 'wb' );
164    $failed = false;
165    while ( ! feof( $fh ) && ! $failed ) {
166      $buf = fread( $fh, 4096 );
172      if ( fwrite( $ofh, $buf ) != strlen( $buf ) ) {
173        $failed = true; break;
176      }
181    }
182    fclose( $ofh );
183    fclose( $fh );
189  }
```

Figure 4. Code snippet from eXtplorer File Manager v2.1.0-RC3

suppose there are two concurrent order requests with the same coupon that is supposed to be redeemed only once. They can be processed concurrently by two threads on two respective CPUs, leading to arbitrary interleaving of the entailed database operations. Fig. 3 shows a failure inducing interleaving, in which the coupon usage selection query of the second user (line 2.013) happens in between the selection query (line 1.013) and the order insertion query (line 1.070) of the first user. As a result, both users observe a valid coupon and are allowed to apply the coupon. OpenCart has also other similar bugs, leading to various undesirable effects such as products being over-ordered.

Although the essence of such bugs is typical atomicity violation, static detection of such violations for web applications is challenging, due to the following reasons.

- Compared to traditional atomicity violations, which are caused by shared memory accesses not being sufficiently protected, web app atomicity violations have different characteristics. They usually don't involve shared memory, but rather external resources. PHP does not have any build-in synchronization primitives either. To reason about external resources, we *have to model the semantics of the operations on these resources* as part of the PHP program. Some apps leverage external synchronizations (e.g. database transactions), which need to be properly modeled in the analysis too.
- The operations involved in an atomicity violation have to access the same external resource. Otherwise, there are no real contentions. However, determining if multiple operations access the same resource is highly challenging. Consider the OpenCart example. We need to determine that the select query at line 13_2 and the insert query at line 70_2 access the same tuples so that a concurrent execution of the select query may observe a stale value (about the same coupon). It requires proving that x at 13_2 is equivalent to y at 70_2, demanding a non-trivial interprocedural analysis.
- A necessary condition for the technique is the availability of atomic region definitions as violations are identified by reasoning about serializability of these regions. There are often a lot of operations accessing the same external resources, and only some of them need to be atomic. Traditional ways to defining atomic regions include user annotations [32], leveraging existing critical sections [14] (e.g. regions delimited by lock acquisitions and their corresponding releases), and inference from dynamic runs [19]. However, none of these solutions are applicable in our context. Furthermore, atomic regions in web apps may not be lexical (i.e. they do not form a lexical region such as a branch or a block of straight line code). In many cases, the operations that ought to execute atomically distribute in different functions. For example in Fig. 2, atomicity is present in the two rectangled SQL queries that are in different functions and the region (i.e. the path between them) is non-lexical.

The goal of our work is to develop a static analysis that overcomes the aforementioned challenges.

B. *Atomicity Violation in eXtplorer File Manager*

Besides databases, file system is another external shared resource that can suffer from similar problems. Next, we use the popular eXtplorer file manager v2.1.0-RC3 web app to explain the atomicity violation problem in file system. EXtplorer is a PHP based file management system. According to the stats of Sourceforge, eXtplorer has been downloaded for more than 300,000 times since its initial release in July 2007. The version we use is the latest release. Its UI is similar to that of a PC file manager. After logging in, the user can create, browse, edit, upload, download, and archive files. While operations are executed on the server via server side scripts, the user controls through a web-based interface on the client side. Some operations may entail downloading files from other remote servers to the local server.

Fig. 4 shows a code snippet from eXtplorer, in which function *download* () is used to download files from other remote servers. It takes two parameters: the address of the source file *$url* and the local path (on the local server) *$outputFile* to save the downloaded file. At line 166, it firstly reads from a remote server to a local buffer and then writes the buffer to the local path at line 172. The read and write are inside a while loop. Depending on the size of the file, *fwrite*() may be invoked several times.

Since a file is incrementally written in the loop, between invocations to *fwrite*() in consecutive iterations, a concurrent *fwrite* () to the same file through a different user request may happen, corrupting the file.

Detecting such file system related atomicity violations requires addressing the same set of challenges as for database related violations.

III. Resource Identity Analysis

In order to reason about atomicity violations regarding external resources, we have to determine if multiple operations are accessing the same external resource. We develop a *resource identity analysis* for this purpose. The analysis encodes the semantics of external operations into bit-vector logic constraints. These constraints, together with those generated from the regular PHP statements, are resolved by a SMT solver to determine if two given operations are accessing the same external resource. We will focus on analyzing database operations, which is more challenging than analyzing file system operations.

A. Analyzing Database Queries

We assume that all SQL queries in a program have their keywords, table names and tuple field names as constants[1].

We preprocess the program to its SSA form so that one variable represents one value in the rules in this section. Moreover, we preprocess the program by introducing dummy variables to represent non-keyword constant strings and values. This is to simplify the description of the analysis as we don't need to distinguish cases operating on constants from those operating on variables.

Modeling Simple Queries. We first discuss SELECT queries that retrieve a set of tuple fields from a single table with a where-clause. Such a select query is modeled as a set of conditional assignments of tuple fields to the result data structure. The assignments are guarded by the conditions described by the where-clause.

$$\begin{aligned}&\$r =^{l} \mathit{mysql\_query}(\text{“}\mathit{SELECT}\ f_1, ..., f_n\ \mathit{FROM}\ T\ \mathit{WHERE}\text{”}.\ C)\\ &\overset{\rho}{\Longrightarrow}\ \rho^t(C) \longrightarrow\ (\mathit{tuple}(l) = t\ \wedge\ r.f_1 = t.f_1 \wedge ... \wedge\ r.f_n = t.f_n)\\ &\qquad \bigwedge (\neg\rho^t(C) \longrightarrow\ \mathit{tuple}(l) \neq t)\end{aligned} \qquad (\textbf{SELECT})$$

$$C_1.\ \text{“}\ \mathit{AND}\ \text{”}.\ C_2 \quad \overset{\rho^t}{\Longrightarrow} \quad \rho^t(C_1) \wedge \rho^t(C_2) \qquad (\textbf{COND-AND})$$

$$C_1.\ \text{“}\ \mathit{OR}\ \text{”}.\ C_2 \quad \overset{\rho^t}{\Longrightarrow} \quad \rho^t(C_1) \vee \rho^t(C_2) \qquad (\textbf{COND-OR})$$

$$\text{“}f \bowtie \text{”}.\ \$x \quad \overset{\rho^t}{\Longrightarrow} \quad t.f \bowtie x \qquad (\textbf{CLAUSE})$$

The above rule (SELECT) presents the encoding. Rules (COND-AND), (COND-OR), and (CLAUSE) describe the where-clause encoding. We use '.' to denote string concatenation. Program statements are on the left and the corresponding encodings are on the right. The arrow in the middle labeled with ρ represents the transformation, which is sometimes also denoted as a function $\rho()$, with an optional superscript representing its context. For example, symbol ρ^t represents that the transformation is in the context of abstract tuple t. We introduce an abstract tuple t to represent the tuple being selected. Observe that t is not constrained. Later, we will show how to properly constrain the abstract tuple based on the information from other queries. Also, while at runtime multiple concrete tuples may be retrieved, we statically represent them with the same abstract tuple. The supporting function `tuple()` in rule (SELECT) maps a program point, denoted by label l, to an abstract tuple. It denotes the query at l entails reading tuple t. We support regular comparison operations in the where-clause, denoted by $\bowtie$ in rule (CLAUSE).

$$\begin{aligned}&\mathit{mysql\_query}^l(\text{“}\mathit{INSERT\ INTO}\ T(f_1, ..., f_n)\,\mathit{VALUES}(\text{”}.\\ &\$x_1.\ ...\ \$x_n)\ \overset{\rho}{\Longrightarrow}\ t_l.f_1 = x_1\ \wedge\ ... \wedge\ t_l.f_n = x_n\end{aligned} \quad (\textbf{INSERT})$$

Rule (INSERT) shows the encoding for an insert query. Note that we introduce an abstract tuple t_l specifically for the query at label l, denoting the tuple inserted by that query. The encoding process models an insert query as a sequence of assignments to the fields of the abstract tuple.

$$\begin{aligned}&\mathit{mysql\_query}^l(\text{“}\mathit{UPDATE}\ T\ \mathit{SET}(f_1 = \$x_1, ..., f_n = \$x_n)\\ &\qquad\qquad \mathit{WHERE}\text{”}.\ C)\\ &\overset{\rho}{\Longrightarrow}\ \rho^t(C) \longrightarrow\ (\mathit{tuple}(l) = t\ \wedge\ t_l.f_1 = x_1\ \wedge\ ... \wedge\ t_l.f_n = x_n)\\ &\quad \bigwedge (\neg\rho^t(C) \longrightarrow\ \mathit{tuple}(l) \neq t)\end{aligned} \qquad (\textbf{UPDATE})$$

An update-query at label l is encoded to conditional field assignments to tuple t_l, guarded by the conditions in the where-clause. Since an update-query entails reading tuples, we use $\mathit{tuple}(l) = t$ to denote the tuple to be read at l. Note that we use different abstract variables t and t_l to denote that the tuple has different values before and after the update.

Constraining Abstract Tuples. In our analysis, we aim to determine if multiple operations are accessing the same shared resource. For instance, we may need to know if a tuple inserted by query at l_2 can be accessed by a select-query at l_1 (assuming the same table and l_1 precedes l_2). If so, l_1 in a thread may contend with l_2 in another thread, leading to atomicity violations (recall the OpenCart example in Section II).

To leverage the SMT solver to determine resource identity, we constrain the abstract tuple in the select-query to those in the insert/update queries, guarded by the where-condition of the select-query. Intuitively, it means that if the tuples added/changed by the insert/update queries satisfy the where condition of the select-query, they may be retrieved by the select-query and thus the two queries access the same tuple. It is possible that multiple inserted/updated abstract

[1] In our experience, dynamic table/column names are not common. For example, OpenCart has 59 data tables, all static, and 780 query strings with only 1 using a variable as the column name.

tuples satisfy the where-condition in the select-query. In such a case, the retrieved tuple can be any of them, without assuming any specific order.

Without losing generality, we assume there are 2 write-queries (i.e. insert or update queries) to the same table T in the program, at labels $l1$ and $l2$. The revised encoding rule for a select query (SELECT-X) to T is presented as follows.

$$\$r =^{l} \mathit{mysql\_query}(\text{“}\mathit{SELECT}\ f_1, ..., f_n\ \mathit{FROM}\ T\ \mathit{WHERE}\text{”}.\ C) \xRightarrow{\rho} F_1 \wedge F_2 \wedge F_{both} \wedge F_{neither} \qquad (\textbf{SELECT-X})$$

$$\begin{array}{lll} F_1 & = & (\rho^{t_{l1}}(C) \wedge \neg\rho^{t_{l2}}(C)) \longrightarrow \\ & & (\mathit{tuple}(l) = t_{l1} \ \wedge \ r.f_1 = t_{l1}.f_1 \wedge ... \wedge r.f_n = t_{l1}.f_n) \\ F_2 & = & (\neg\rho^{t_{l1}}(C) \wedge \rho^{t_{l2}}(C)) \longrightarrow \\ & & (\mathit{tuple}(l) = t_{l2} \ \wedge \ r.f_1 = t_{l2}.f_1 \wedge ... \wedge r.f_n = t_{l2}.f_n) \end{array}$$

$$\begin{array}{lll} F_{both} & = & (\rho^{t_{l1}}(C) \wedge \rho^{t_{l2}}(C)) \longrightarrow \\ & & (\mathit{tuple}(l) = t_{l1} \ \wedge \ r.f_1 = t_{l1}.f_1 \wedge ... \wedge r.f_n = t_{l1}.f_n) \ \bigvee \\ & & (\mathit{tuple}(l) = t_{l2} \ \wedge \ r.f_1 = t_{l2}.f_1 \wedge ... \wedge r.f_n = t_{l2}.f_n) \end{array}$$

$$F_{neither} = (\neg\rho^{t_{l1}}(C) \wedge \neg\rho^{t_{l2}}(C)) \longrightarrow \ \mathit{tuple}(l) \neq t_{l1} \wedge \mathit{tuple}(l) \neq t_{l2}$$

The revised encoding of a select-query is the conjunction of four clauses F_1, F_2, F_{both}, and $F_{neither}$. F_1 constrains the selected tuple with the tuple at $l1$ but not the one at $l2$. F_2 is the opposite. F_{both} describes that if both tuples at $l1$ and $l2$ satisfy the where-condition, the selected tuple could be either of them.

Example. Consider a simple PHP program as follows. It performs three queries, the first two insert two tuples with the first field being 12 and 8, respectively, and the third query selects the tuples with the first field greater-than 10. Therefore, only the tuple inserted at 1 is selected.

```
1 mysql_query("INSERT TO T(f1,f2) VALUES(12,".$x);
2 mysql_query("INSERT TO T(f1,f2) VALUES(8,".$y);
3 $r=mysql_query("SELECT f2 FROM T WHERE f1>10");
```

The corresponding encoding is as follows. From the three formula, it is easy to infer $\mathit{tuple}(3) = t_1$.

$$\begin{array}{lll} \rho(1) & = & t_1.f_1 = 12 \ \wedge t_1.f_2 = x \\ \rho(2) & = & t_2.f_1 = 8 \ \wedge t_2.f_2 = y \\ \rho(3) & = & \\ & & (t_1.f_1 > 10 \wedge \neg t_2.f_1 > 10) \longrightarrow (\mathit{tuple}(3) = t_1 \wedge r.f_2 = t_1.f_2) \bigwedge \\ & & ... \end{array}$$

Aggregation queries are very common in PHP programs. Sample aggregation queries include those acquiring the count, sum, average of tuples/fields. They are usually accompanied by the groupby keyword. It is difficult to statically model values of aggregation queries. Fortunately in our context, we only need to model the correlation between queries, such as if an insert query affects the result of an aggregation query, which is mainly determined by the where-condition instead of the specific aggregation function. Hence our encoding is very similar to that for regular selects and thus elided. Our technique currently does not support sub-queries.

Modeling PHP statements. Modeling queries alone is not sufficient to reason about resource identity as queries may use variables that are computed by regular PHP statements. Therefore, we have to model regular PHP statements as part of the analysis, including assignments, conditionals, loops, and simple arithmetic operations.

Assignment statements are encoded as simple equivalence constraints. The encoding is field sensitive, supporting definitions and uses of data structure fields. It also models the access of a query result through keyword row. For instance, $\$x = \$r \rightarrow row['f_1']$ is modeled as $x = r.f_1$.

Similar to all constraint solving based static analysis [13], [35], loops need to be unrolled to conditional statements. In this work, we unroll the loop body twice. It allows us to reason about the operations in consecutive iterations. More unrollings are unlikely useful as it is unlikely for a resource identity property to manifest itself after a large number of iterations in PHP programs.

Conditional statements are encoded from their SSA form. The following example shows a typical SSA transformation.

$$\boxed{\begin{array}{ll} 1\ \mathit{if}\ (C) & \\ 2 & \$x = E_1 \\ 3\ \mathit{else} & \\ 4 & \$x = E_2 \end{array}} \xRightarrow{ssa} \boxed{\begin{array}{ll} 11\ \mathit{if}\ (C) & \\ 12 & \$x_1 = E_1 \\ 13\ \mathit{else} & \\ 14 & \$x_2 = E_2 \\ \multicolumn{2}{l}{15\ \$x_3 \ = \ (C)?\ \$x_1 :\ \$x_2;} \end{array}}$$

A conditional assignment such as the one at line 15 is further encoded to a constraint by the following rule (COND-ASSN).

$$\$x = C?\$y : \$z \quad \xRightarrow{\rho} \quad (\rho(C) \longrightarrow x = y) \wedge (\neg\rho(C) \longrightarrow x = z) \qquad (\textbf{COND-ASSN})$$

Condition C in the rule corresponds to the predicate in the original conditional statement. It is also called the guard of the assignment. Our analysis is hence path sensitive as by encoding predicates, we are able to leverage the solver to associate a resource identity property with a feasible path, determined by finding a satisfying solution to the related guards. Our analysis is also interprocedural. It is summary-based.

At last, we deliver analysis results through a primitive relation $\mathit{same\_resource}$, which can be queried by other analysis components.

$$\mathit{same\_resource}(l_1, l_2) \ :- \ \rho(\mathit{program}) \wedge \mathit{tuple}(l_1) = t_{l_2}$$

The meaning of the relation is that the program is accessing the same resource at program points l_1 and l_2 if the formula to the right of symbol $:-$ is satisfiable. Intuitively, it means that from the program encoding we can infer that the tuples accessed at l_1 and l_2 are the same.

Example. Consider the motivation example in Fig. 2. Part of the encoding is presented in Table I. From the conjunction of (1)-(14), we can infer $\mathit{tuple}(13_3) = 70_3$ so that we have $\mathit{same\_resource}(13_3, 70_3)$. In particular, the condition in (2), i.e. $t_{70_3}.\mathit{coupon\_id} = x$, can be inferred from (1) $x = \mathit{coupon\_query.coupon\_id}$ and the equivalence of y and $\mathit{coupon\_query.coupon\_id}$, which is established from the conjunction of (14), (13), (12), (10), (8), (7), (5) and (4). □

Table I
PART OF THE ENCODING OF THE CODE SNIPPET IN FIG. 2.

statement	encoding
13_1	(1) $x = \mathit{coupon\_query.coupon\_id}$
$13_2,\ 13_3$	(2) $t_{70_3}.\mathit{coupon\_id} = x \longrightarrow (\mathit{tuple}(13_3) = 70_3 \wedge \mathit{coupon\_redeem\_query.total} = fv_{13_3} \bigwedge \ldots$
14	(3) $\mathit{total} = \mathit{conpon\_redeem\_query.total}$
$15,\ 59$	(4) $\mathit{coupon\_data}_1 = \mathit{coupon\_query}$ (5) $\mathit{total} < \mathit{allowed} \longrightarrow \mathit{coupon\_data}_2 = \mathit{coupon\_data}_1$ (6) $\neg\ \mathit{total} < \mathit{allowed} \longrightarrow \mathit{coupon\_data}_2 = \mathit{false}$
$76,\ 201_2$	(7) $\mathit{coupon} = \mathit{coupon\_data}_2$
$203 - 206$	(8) $\mathit{data}_1.\mathit{coupon\_id} = \mathit{coupon.coupon\_id}$ (9) $\mathit{data}_2.\mathit{coupon\_id} = 0$ (10) $\neg \mathit{coupon} = \mathit{false} \longrightarrow \mathit{data}_3 = \mathit{data}_1$ (11) $\mathit{coupon} = \mathit{false} \longrightarrow \mathit{data}_3 = \mathit{data}_2$
$216,\ 58$	(12) $\mathit{data}_4 = \mathit{data}_3$
70_1	(13) $y = \mathit{data}_4.\mathit{coupon\_id}$
$70_2,\ 70_3$	(14) $t_{70_3}.\mathit{coupon\_id} = y$

fv_{13_3} in (2) is a free variable.

B. Analyzing File Operations

Determining if multiple file operations have the same subject file is relatively easier. In this work, we focus on file open, read and write. We do not consider other file operations (e.g. getting a file's path) as they are less likely to be error-prone due to their nature. The basic idea of our analysis is to use the label of a file open statement as the abstraction of the file and then use constraint solver to reason about if multiple operations are on the same abstract file. Analysis results are delivered through the same primitive relation $\mathit{same\_resource}(l_1, l_2)$.

IV. SERIALIZABILITY ANALYSIS

Atomicity violation detection is a process of reasoning about serializability of operations in atomic regions. Specifically, given an sequence of interleaved operations from multiple (usually two) atomic regions in different threads/processes, a violation is reported if the sequence is not serializable. Hence, atomic region definitions are critical. In this work, we do not require the developer to annotate atomic regions. A way to addressing the lack of atomic region definitions, as in [19], [34], [28], is to reason about *pair-wise atomicity* instead, that is, to reason about if two consecutive accesses of the same shared memory location inside a thread, called *local accesses*, should be atomic. For example, in [19], [34], if during training executions, two consecutive local accesses are never observed to interleave with another access from a different thread, called the *remote access*, they are considered to be atomic. The key observation is that inferring atomic pairs is a lot more tractable than inferring an arbitrary atomic region.

In this work, we adopt a similar solution by considering pair-wise atomicity of external operations. However, we do not rely on dynamic runs but rather infer statically, leveraging the aforementioned resource identity analysis and program dependences. With such atomic pairs, violations are detected by reasoning about serializability of interleaved pairs. We will discuss how to infer atomic pairs in the next section. In this section, we assume the availability of atomic pairs and discuss the serializability analysis.

Typical Serializability Analysis Not Applicable. In existing work [19], [34], it was shown that to detect pair-wise atomicity violations, it is sufficient to analyze serializability when an atomic pair of accesses are interleaved with a remote access. Because shared memory accesses are modeled as two kinds: *read* and *write*. There are totally 2^3 possible patterns. Table II presents some of these patterns. For example in case two, given an atomic pair that first reads (R) and then writes (W), if it interleaves with a remote read (R'), the resulting sequence is serializable as it is equivalent to the sequence of R' R W. In contrast, case one is not.

Table II
DETERMINING PAIR-WISE ATOMICITY VIOLATIONS FOR SHARED MEMORY. R DENOTES READ, W DENOTES WRITE. *Local*$_{(1/2)}$ ARE THE TWO LOCAL ACCESSES INVOLVED IN AN ATOMIC PAIR. EACH ENTRY SHOWS A PAIR INTERLEAVED WITH A REMOTE ACCESS.

case	*local*$_1$	*remote*	*local*$_2$	*serializable*
1	R	W'	W	no
2	R	R'	W	yes
3	W	W'	W	yes
	...		...	
8	W	W'	R	no

However, such analysis is not sufficient for our purpose.

First, *modeling external operations to merely reads and writes is problematic.* For example, intuitively, we should model a file write as W. According to case 3 in Table II, two local file writes interleaved with a remote file write are serializable, which is wrong. Consider another example. Intuitively, SQL selects should be modeled as R, and deletes as W. In case 1, a local select and a local delete interleaved with a remote delete (on the same tuple) are not serializable, which is incorrect as it has the same consequence as first performing the two local operations and then the remote deletion, which becomes a no-op. The reason of these problems is that the semantics of file writes and SQL deletes cannot be precisely described as low level writes.

Second, *while all the eight patterns are possible for shared memory accesses, a lot of them are rarely observed in external operations in our experience.* For instance, an atomic pair with the form of WR is common for shared memory accesses. But it may not be the case for external operations, especially in PHP code, because it is unlikely that a program first writes to a tuple/file, and then reads it in the same thread.

Table III
CATEGORIZING EXTERNAL OPERATIONS.

Category	Description	Operations
A	append	SQL inserts, file writes
D	delete	SQL deletes
W	write	SQL updates
R	read	SQL selects, file reads

Our Solution. We propose to model external operations to four access categories, as shown in Table III, covering

$$(1)\ R\ R'\ (A|W|D)\ (A'|W') \qquad (2)\ A\ (A'|R'|W')\ A$$
$$(3)\ W\ (A'|R')\ (W|A) \qquad (4)\ D\ D'\ A\ A'$$

Figure 5. Atomicity violation patterns for external operations. Remote operations are superscripted.

the most common external operations. Two new categories, *append* and *delete*, are introduced.

Note with the new categories, theoretically there are many more interleaving patterns. Fortunately, according to our discussion earlier, most of them are not feasible in practice. We hence only consider a small subset as listed in Fig. 5. We also preclude patterns that are serializable.

The examples in Fig. 2 and 4 belong to the patterns R R' A A' and A A' A, respectively. One example of pattern (3) is that two update-queries update two disjoint sets of fields of the same tuple. A select query from a different thread is not supposed to see the partially updated tuple. Note that pattern (4) involves four operations, which are the interleaving of two DA atomic pairs. The code snippet in Fig. 6 shows an example. It corresponds to the coding pattern of cleaning stale data (maybe multiple tuples) before inserting a new tuple with the same key. Sequence D D' A A' is not serializable as two tuples will be inserted. This pattern also discloses that reasoning about only triples (two locals and one remote) as in shared memory serializability analysis is not sufficient for external resources. Observe that although pattern (4) is unserializable, its sub-patterns D D' A and D' A A' are serializable.

```
$s1 = "DELETE FROM book WHERE ID = '" . $bookID . "'";
$r1 = mysql_query($s1);
/*update information about the book*/
$s2 = "INSERT INTO book VALUES('". $bookID ."', .... )";
$r2 = mysql_query($s2);
```

Figure 6. Example for an atomic pair involving delete- and insert-queries.

V. Atomic Pair Inference

In this section, we discuss how to infer atomic pairs from PHP programs. Considering all operations in a PHP thread/process atomic is too simplistic to be useful. Many operations present in a program allow concurrency. Our experimental results (in Section VII) show that naively considering all operations on the same resource atomic leads to many false positives.

We propose to consider program dependence in order to infer pair-wise atomicity properties. The observation is that if two external operations are correlated through program dependences, the original intention of the developer was most likely to assume such program dependences are exercised in a way identical to a sequential execution. Recall that in the OpenCart example in Fig. 2. The insert-query is transitively dependent on the select-query as the execution of the insert-query is determined by if the value of the selected *total* number of coupon uses has not reached the *allowed* limit. The precise dependence path is indicated in the figure by underlining the involved variables. The assumption implied by the dependence path is that the total number of uses should remain constant from the select-query till the insert-query that places a new order.

We consider program dependences in two different ways depending on the category of an atomic pair.

- *A pair of operations with the first operation being a read is considered atomic if there is a dependence path, including both control and data dependences, from the first operation to the second.* It corresponds to that the result of the first operation is used in the second operation. The OpenCart example illustrates this case.
- *A pair of operations with neither being a read is considered atomic if both are data dependent on the same variable and there is a valid program path correlating the two operations.* Intuitively, it means that both operations are consuming/storing the same or correlated computation results. Such a process is often not intended to be interfered by other threads. The eXtplorer example in Fig. 4 illustrates this case. The file writes in two consecutive iterations are both dependent on the creation of the output file. Another example is presented in Fig. 6, in which the two operations are delete and insert. Both are dependent on the definition point of variable $bookID.

Datalog Rules for Atomic Pair Inference. The atomic pair inference process is described as datalog rules in Fig. 7. Datalog [9] uses a Prolog-like notation. It provides a neat representation for whole program analysis. Data flow facts can be formulated as relations. Analysis is represented as inference rules on these relations. Relations are in the form $P(x_1, x_2, ..., x_n)$ with P being a predicate and $x_1, ..., x_n$ representing program artifacts, such as labels. A predicate is a declarative statement on the variables. For example, $datadep(l_1, l_2)$ denotes that there is a data dependence path from l_1 to l_2.

The form of an inference rule is as follows.

$$P \text{ :- } B_1,\ B_2,\ ...,\ B_n$$

B_1, B_2, ..., and B_n are either relations or negated relations. The rule means that if B_1, B_2, ..., and B_n are true then P is true.

Relations can be either inferred or atoms. In program analysis, we often start with a set of atoms that describe basic facts of the program and then infer other more interesting relations.

In Fig. 7, we assume program dependence relation dep and data dependence relation $datadep$ as atoms. These relations are generated through standard program analysis.

Rules (D1) and (D2) present the rules that infer atomic pairs. Rule (D1) describes the process of inferring atomic pairs with the first operation being a read. It infers from 4 atoms, in which $same\_resource$ relation is described in Section III. In the OpenCart example, an entry $sql\_atomic\_RA(13_3, 70_3)$ is inferred.

Rule (D2) describes the inference of atomic pairs with neither operation being a read. Note that in (D1), we don't explicitly require l_2 is reachable from l_1 because $dep(l_1, l_2)$ implies that. In the example in Fig. 6, an entry $sql\_atomic\_DA(2, 4)$ is inferred.

Rules (D5) and (D6) describe the inference of file operation atomic pairs, which is very similar to database operation pairs. Note that the possible patterns for file operation atomic pairs are fewer. Other patterns are either impossible or rare.

VI. Violation Detection

Given the inferred atomic pairs, violations are detected by observing if the interleaving patterns presented in Fig. 5 can happen. Since the remote operation is from a different execution instance of the *same* PHP code, it is sufficient to analyze if the same program contains the specific offending remote operation. Although PHP does not provide any buildin synchronization support, developers can make use of external primitives such as database transactions, table locks, and file locks to ensure atomicity. Therefore, our technique also needs to detect if an atomic pair is well protected by those external primitives.

Rule (D3) determines if a given pair of program points (l_3, l_4) is nested in a database transaction. It requires the existence of a transaction that starts and ends at l_1 and l_2, respectively, and l_1 dominates l_3 and l_2 post-dominates l_4 such that all paths leading from program entry to l_3 must go through l_1 and all paths from l_4 to program exit must go through l_2.

Rule (D4) detects query atomicity violations of the pattern WR'W with R' being a remote select-query. It reports (l_1, l_2) as an atomic WW pair that could be violated if there exists a sql-select at l_3 that operates on the same abstract tuple, and (l_1, l_2) is not protected by a database transaction. Our analysis also models table locks. Due to the space limitation, we are not presenting the relevant rules.

Detection rules for other sql operation and file operation interleaving patterns can be similarly derived as Rule (D4).

VII. Evaluation

Our system is implemented on LLVM[7], an open source PHP compiler (phc) [6], and the STP solver [8]. Phc is used to translate PHP to C. The main analysis is implemented in LLVM. It takes the C program and transforms it to constraints, which are resolved by the solver. We translate PHP to C to leverage LLVM for call graph construction, points-to analysis, etc., as we are not aware of infrastructures that allow us to analyze PHP direclty.

Since phc aims to generate C code that is compilable and executable, the PHP features not directly supported by C are realized by chunks of C code. For example, array fields in PHP can be added dynmaically. In translated C, hash tables are used, substantially increasing difficulty for our analysis. We therefore modified the code generator of phc to generate simplified C code by replacing those hash table accesses with field accesses. The resulting code may not be executable, but reflecting the original semantics.

The LLVM component translates the generated C programs to their SSA forms, which are further encoded to constraints. In order to understand external operations and encode them properly, we implement a simple string analysis that tracks string concatenations so that we can acquire the the query strings. A string variable is mapped to a linked list of constants and variables denoting its value. Query strings can be parsed to identify table name and field names. Our system currently requires these names to be constant.

Since PHP files are largely independent modules,it is unnecessary to encode all the files of a web app. We use a demand-driven strategy. In particular, given a query about resource identity, we perform program slicing to identify the relevant PHP modules and functions and then only encode the slice.

We apply our technique to a set of real world web applications. The benchmarks are mainly from previous PHP analysis works[11], [31], [36], excluding those that have trivial external resource accesses or functionally overlap with the selected ones. In addition, the shopping, forum and wiki kinds of apps are often accessed concurrently, so we randomly pick OpenCart, phpBB, aphpkb for each kind.

The characteristics of these programs are listed in Table IV. Observe that many of them are very large web apps, with a few hundred PHP files and over 100k LOC. All experiments are run on an Intel Dual Core 2.5GHz machine with 2GB memory. The OS is Linux-2.6.35.

Table IV
PROGRAM CHARACTERISTICS.

	PHP	PHP LOC			
Application	files	mean	stdev	max	total
openCart v1.4.9.4	535	99	152	1233	53025
phpBB v3.0.0	245	685	2933	45178	167797
ajallerix v0.1	16	402	1283	5207	6435
eXtplorer v2.1.0-RC3	277	291	397	4618	80598
scarf v2007-02-27	19	89	91	369	1686
phpoll v0.97 beta	27	167	122	494	4522
AWCM v2.2	187	79	85	483	14717
webChess v1.0.0_rc2	28	186	264	1243	5219
faqforge v1.3.2	19	90	73	218	1710
schoolMate v1.5.4	63	129	94	539	8120
timeclock v1.04	63	330	437	2832	20800
aphpkb v0.95.5	46	93	67	264	4283
news pro v1.4.0	30	231	183	811	6925
DCP-Portal v 6.1.1	362	335	428	5075	121410
Employee Scheduler v2.1beta	43	215	220	1231	9264

Table V presents the result of violation detection. "*PHP LOC w/ inclusion*" is the average LOC of PHP files after inlining the scripts indicated by the `include` keyword. "*Converted C LOC*" is the average LOC of the C programs translated from the expanded PHP. "*Constraint complexity*" presents the average number of *variables* and *constraints* in the formula. The last four columns present the number violations reported and the number of false positives for our technique and a simplified static analysis. "*SA w/o Dep.*"

Atoms

$dep(l_1, l_2)$: there is a program dependence path from l_1 to l_2, including both data and control dependences.
$datadep(l_1, l_2)$: there is a data dependence path from l_1 to l_2.
$sql\_R/W/A/D(l)$: there is a SQL select/update/insert/delete query at l.
$file\_R/A(l)$: there is a file read/write operation at l.
$reachable(l_1, l_2)$: l_2 is reachable from l_1.
$trans(l_1, l_2)$: a database transaction is created at l_1 and then released at l_2.
$dom(l_1, l_2)$: l_1 dominates l_2.
$pdom(l_1, l_2)$: l_1 post-dominates l_2.

Rules for Database Queries

/* (l_1, l_2) is a sql RW/RA/RD atomic pair*/
(D1) $sql\_atomic\_RW/RA/RD(l_1, l_2)$:- $sql\_R(l_1),\ sql\_W/A/D(l_2),\ same\_resource(l_1, l_2),\ dep(l_1, l_2)$

/* (l_1, l_2) is a sql DA/WW/WA atomic pair*/
(D2) $sql\_atomic\_DA/WW/WA(l_1, l_2)$:- $sql\_D/W/W(l_1),\ sql\_A/W/A(l_2),\ same\_resource(l_1, l_2),\ reachable(l_1, l_2),$
$datadep(l, l_1),\ datadep(l, l_2)$

/* The two operations at l_3 and l_4 are protected by the transaction in between l_1 and l_2*/
(D3) $in\_trans(l_1, l_2, l_3, l_4)$:- $trans(l_1, l_2),\ dom(l_1, l_3),\ pDom(l_2, l_4)$

/* WR'W atomicity violation, l_1 and l_2 should be atomic, but the interleaving with l_3 is not serializable*/
(D4) $sql\_violation\_wRw(l_1, l_2)$:- $sql\_atomic\_WW(l_1, l_2),\ sql\_R(l_3),\ same\_resource(l_3, l_1), \neg\ in\_trans(l_4, l_5, l_1, l_2)$

Rules for File Operations

/* (l_1, l_2) is a file RA atomic pair*/
(D5) $file\_atomic\_RA(l_1, l_2)$:- $file\_R(l_1),\ file\_A(l_2),\ same\_resource(l_1, l_2),\ dep(l_1, l_2)$
/* (l_1, l_2) is a file AA atomic pair*/
(D6) $file\_atomic\_AA(l_1, l_2)$:- $file\_A(l_1),\ file\_A(l_2),\ same\_resource(l_1, l_2),\ reachable(l_1, l_2),\ datadep(l, l_1),\ datadep(l, l_2)$

Figure 7. Datalog rules for atomicity violation detection for external operations. W, R, A, and D denote write, read, append, and delete.

Table V
ANALYSIS RESULT.

	PHP LOC w/	Converted	Complexity (avg)		Our method		SA w/o Dep.	
Application	inclusion (avg)	C LOC (avg)	Variable	Constraint	vio$_c$	FP$_c$	vio$_s$	FP$_s$
openCart v1.4.9.4	899	49068	1359	1972	32	0	32	0
phpBB v3.0.0	2490	83683	1054	1568	14	0	14	0
ajallerix v0.1	86	5170	247	762	1	0	1	0
eXtplorer v2.1.0-RC3	781	10559	326	598	2	0	2	0
scarf v2007-02-27	402	9313	76	165	5	0	12	7
phpoll v0.97 beta	208	5586	83	232	4	0	4	0
AWCM v2.2	2679	51607	124	227	2	0	15	13
webChess v1.0.0_rc2	1706	63616	82	174	7	0	11	4
faqforge v1.3.2	217	2993	52	89	3	0	3	0
schoolMate v1.5.4	421	14489	123	185	11	3	33	25
timeclock v1.04	1154	25746	147	194	3	0	27	24
aphpkb v0.95.5	842	17663	75	97	3	0	5	2
news pro v1.4.0	1129	28331	178	239	11	0	30	19
DCP-Portal v 6.1.1	423	20940	253	597	11	0	11	0
Employee Scheduler v2.1beta	2089	84659	169	271	7	0	8	1
Total	—	—	—	—	116	3	208	95

represents a technique that can be considered as a static version of the one used in [25] (which is dynamic). In particular, it does not infer atomicity properties from PHP code, but rather directly compares the query strings. Queries that access the same table and abstract tuple, and may form unserializable interleavings are reported. The analysis time is mostly within a few seconds and thus elided.

We make the following observations from the result.

- Our analysis is able to detect many violations in these real world web apps. We manually validate each bug by constructing a real test input and exercising the problematic interleaving pattern. These bugs could lead to problems such as coupon misuses, product being oversold, data corruption and runtime database exceptions.
- Our analysis produces very few false positives.
- The simplified approach produces many false positives. This illustrates the benefit of analyzing PHP code. It also suggests that even though many queries are accessing the same table and the same tuples, they are allowed to execute concurrently.
- Slicing is an effective optimization because even though the programs are large on average, the average numbers of constraints and symbolic variables are small.

False Positive. Our analysis sometimes reports false positives. A typical example is shown in Fig. 8. Variables such as *$_POST[...]* hold the values submitted by the client so that they are defined in the client-side. Our analysis is not able to make any assumptions about these values. Therefore, we treat them as free variables.

Based on such assumptions, the predicates at lines 1 and 5 are both satisfiable, rendering the path between the two queries (at lines 3 and 7) feasible. Since the queries access the same resource and both depend on the same `$_POST[total]` variable, according to our detection rules, the two operations are considered to be atomic. Interleaving "1,1',5" hence constitutes a violation. However, in practice, these two queries are not related.

The reason of the FP in Fig. 8 is that we missed the constraint that a client can never submit a single request that can add a new record and edit an existing one in the meantime. We speculate if we can model the client-side logic, such FPs can be eliminated. We leave it to our future work.

```
1  if($_POST["addassignment"] == 1) {
2    $s1 = "UPDATE courses SET ... $_POST[total] ... ";
3    $r1 = mysql_query($s1);
4  }
5  if($_POST["editassignment"] == 1) {
6    $s2 = "UPDATE courses SET ... $_POST[total] ... ";
7    $r2 = mysql_query($s2);
8  }
```

Figure 8. False Positive Example - simplified snippet from *schoolMate*.

False Negative. Our analysis is incomplete and may have false negatives. For example, the atomicity inference is heuristic based. The current string analysis handles string concatenation but doesn't support functions such as substr() or strrpos(). However, without an oracle, it's hard to determine false negatives automatically. We leave it to our future work.

VIII. Related Work

Data Race and Atomicity Violation Detections. There are many works on data race detection [27], [24], [22], [23] and atomicity violation detection[14], [19], [26], [18].They are mostly addressing problems caused by shared memory accesses. They often leverage synchronization primitives. In contrast, the problem in our scenario is caused by sharing external resources. Server side scripts provide no build-in threading or synchronization support. In other words, we have to address a largely different set of challenges.

Web app testing. Server side script testing is increasingly studied lately. Wassermann et al. [33] designed an automatic input generation algorithm for web apps based on concolic execution. They also models the semantics of string operations and solve constraints involving different types. Harman et al. [16] proposed a session data repair method for regression testing. The work by Halfond et al. [15] precisely identifies a web app's interface to improve test input generation via symbolic execution. Artzi et al. [11] proposed to combine concrete and symbolic executions to automatically generate test cases that expose faults by analyzing the server-side script. Sprenkle et al. [30] suggested that statistical model-based test generation can be adopted and applied for web app testing. Marchetto et al.[21] proposed a testability measurement that can be leveraged in automated testing of web apps. The vulnerability measurement proposed in [29] heuristically inspects SQL hotspots in server scripts to decide priority. Carzaniga et al. [12] proposed an automatic workaround of web app failures. Provided a failure, it tries to find a different execution sequence that achieves the same functionality while bypass the failure. The above approaches do not address problems caused by concurrent executions.

Web app comprehension. Since web app source code is usually not well organized, it is difficult for human developer/maintainer to understand the (complex) correlations between modules. Thus, many works have been proposed to help people get better understanding. Hassan et al. [17] proposed to extract code structure and display the interactions between components. *WANDA* [10] instruments web apps and combines dynamic and static informations to address the problem. Similarly, the integration of *WARE* and *WANDA*[20] combines static and dynamic analysis to enhance comprehension. These works are too general to solve our problem.

IX. Conclusion

We propose a static analysis that detects atomicity violations in web apps regarding external resources. The technique features a novel resource identity analysis that is interprocedural and path-sensitive. It models external operations to constraints and leverages a SMT solver to determine whether multiple operations are accessing the same external resource, which is a critical condition for contention. We also develop an automated approach to statically infer if a pair of operations that access the same resource demands atomicity. Violations are detected by reasoning about serializability of interleaved atomic pairs. Our results show that the technique is highly effective, capable of detecting many real atomicity violations in large web apps.

Acknowledgments

We would like to thank the reviewers for their substantial efforts. This research is supported, in part, by the National Science Foundation (NSF) under grants 0834529 and 0845870. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] ProductCart overselling. http://www.rotofugi.com/toyscart/pc/viewContent.asp?idpage=48.

[2] OpenCart overselling. http://forum.opencart.com/viewtopic.php?f=20&t=37073.

[3] Drupal critical runtime failure. http://drupal.org/node/566832.

[4] Joomla file corruption. http://forum.joomla.org/viewtopic.php?p=1795892.

[5] 5 Best Open Source Shopping Carts. http://www.stylomart.com/technology/5-best-open-source-shopping-carts/.

[6] Phc: open source PHP compiler. http://www.phpcompiler.org/.

[7] The LLVM Compiler Infrastructure Project. http://llvm.org/.

[8] The STP Constraint Solver. https://sites.google.com/site/stpfastprover/.

[9] A. Aho, M. Lam, R. Sethi, and J. Ullman. Compilers: principles, techniques, and tools (2nd Ed.). *Pearson Education, Inc, 2006*.

[10] G. Antoniol, M. Penta, M. Zazzara. Understanding Web Applications through Dynamic Analysis. *IWPC'04*.

[11] S. Artzi, J. Dolby, F. Tip and M. Pistoia. Practical fault localization for dynamic web applications. *ICSE'10*.

[12] A. Carzaniga, A. Gorla, N. Perino and M. Pezzè. Automatic workarounds for web applications. *FSE'10*.

[13] E. Clarke, D. Kroening and K. Yorav. Behavioral consistency of C and verilog programs using bounded model checking. *DAC'03*.

[14] C. Flanagan and S. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL'04*.

[15] W. Halfond, S. Anand and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *ISSTA'09*.

[16] M. Harman and N. Alshahwan. Automated Session Data Repair for Web Application Regression Testing. In *ICST'08*.

[17] A. Hassan and R. Holt. Architecture recovery of web applications. In *ICSE'02*.

[18] Z. Lai, S. C. Cheung and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing invariants. In *ICSE'10*.

[19] S. Lu, J. Tucek, F. Qin and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII*.

[20] G. Lucca and M. Penta. Integrating Static and Dynamic Analysis to improve the Comprehension of Existing Web Applications. *WSE'05*.

[21] A. Marchetto, R. Tiella, P. Tonella, N. Alshahwan, M. Harman. Crawlability metrics for automated web testing. *STTT 13(2) 2011*.

[22] M. Naik, A. Aiken and J. Whaley. Effective static race detection for Java. In *PLDI'06* .

[23] A. Nistor, D. Marinov and J. Torrellas. Light64: lightweight hardware support for data race detection during systematic testing of parallel programs. In *MICRO-42*.

[24] R. O'Callahan and J. Choi. Hybrid dynamic data race detection. In *PPoPP'03*.

[25] R. Paleari, D. Marrone, D. Bruschi and M. Monga. On Race Vulnerabilities in Web Applications. In *DIMVA'08*.

[26] C. Park, K.Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT'08/FSE-16* .

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. In *TOCS 15(4) 1997*.

[28] A. Singh, D. Marino, S. Narayanasamy, T. Millstein and M. Musuvathi. Efficient processor support for DRFx, a memory model with exceptions. In *ASPLOS'11*.

[29] B. Smith and L. Williams. Using SQL Hotspots in a Prioritization Heuristic for Detecting All Types of Web Application Vulnerabilities. In *ICST'11*.

[30] S. Sprenkle, L. Pollock and L. Simko. A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications. In *ICST'11*.

[31] F. Sun, L. Xu, and Z. Su. Static Detection of Access Control Vulnerabilities in Web Applications. In *USENIX Security 2011*.

[32] M. Vaziri, F. Tip and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL'06*.

[33] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura and Z. Su. Dynamic test input generation for web applications. In *ISSTA'08*.

[34] D. Weeratunge, X. Zhang, and S. Jagannathan. Accentuating the Positive: Atomicity Inference and Enforcement Using Correct Executions. In *OOPSLA'11*.

[35] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL'05*.

[36] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX'06*.

[37] Y. Zheng, T. Bao and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *WWW'11*.