Fix Me Up: Repairing Access-Control Bugs in Web Applications

Sooel Son† Kathryn S. McKinley†∗ Vitaly Shmatikov†
†The University of Texas at Austin ∗Microsoft Research
{samuel,mckinley,shmat}@cs.utexas.edu

Abstract

Access-control policies in Web applications ensure that only authorized users can perform security-sensitive operations. These policies usually check user credentials before executing actions such as writing to the database or navigating to privileged pages. Typically, every Web application uses its own, hand-crafted program logic to enforce access control. Within a single application, this logic can vary between different user roles, e.g., administrator or regular user. Unfortunately, developers forget to include proper access-control checks, a lot.

This paper presents the design and implementation of FIXMEUP, a static analysis and transformation tool that finds access-control errors of omission and produces candidate repairs. FIXMEUP starts with a high-level specification that indicates the conditional statement of a correct access-control check and automatically computes an interprocedural access-control template (ACT), which includes all program statements involved in this instance of access-control logic. The ACT serves as both a low-level policy specification and a program transformation template. FIXMEUP uses the ACT to find faulty access-control logic that misses some or all of these statements, inserts only the missing statements, and ensures that unintended dependences did not change the meaning of the access-control policy. FIXMEUP then presents the transformed program to the developer, who decides whether to accept the proposed repair.

Our evaluation on ten real-world PHP applications shows that FIXMEUP is capable of finding subtle access-control bugs and performing semantically correct repairs.

1 Introduction

Modern Web-based software, such as e-commerce applications, blogs, and wikis, typically consists of client-side scripts running in a Web browser and a server-side program that (1) converts clients’ requests into queries to a back-end database and (2) returns HTML content. Because any Internet user can invoke the server, application developers must ensure that unauthorized users cannot reach database queries, administrator functionality, pages with confidential or paid content, and other privileged operations.

Developers usually program access-control logic from scratch because there is no standard framework for implementing access control in Web applications. Access-control logic is often fairly sophisticated, spread over multiple functions, with different checks performed for different user roles [32, 36]. The scripting language of choice for server-side applications is PHP [27, 28]. In PHP, a network user can directly invoke any program file by providing its name as part of a URL. This feature introduces unintended entry points into programs and permits “forced browsing,” where a user navigates to pages without following the intended pattern and therefore bypasses access-control checks. As a consequence, incorrectly implemented access-control vulnerabilities occur prominently in the OWASP Top 10 Application Security Risks [24]. For example, all but one of the ten real-world PHP applications analyzed in this paper contain access-control vulnerabilities.

Whereas finding bugs is now a mature area, repairing them is a much harder problem and only recently has some progress been made on semi-automated methods for software repair. Static repair techniques can now fix violations of simple local patterns that need only one- or two-line edits [14, 25], or find one- or two-line changes that pass unit tests [41], or perform user-specified transformations within a single method [1, 19]. None of these techniques address interprocedural bugs. Several recent methods find access-control bugs using interprocedural analysis [32, 36] but how to repair them has been an open problem. A key issue for repairing these bugs is that many, but not all of the statements implementing the access-control logic are often already present in the vulnerable code. None of the prior patch, transformation, refactoring, or repair algorithms check if the statements are already present in the target of transformation.

We design and implement a static analysis and program transformation tool called FIXMEUP. FIXMEUP finds violations of access-control policies, produces candidate repairs, eliminates repairs that incorrectly implement the policy, and suggests the remaining repairs to developers.
As input, FixMeUp takes an access-control check, i.e., a conditional statement that determines whether or not some security-sensitive operation executes. These checks, marked by the developer or inferred by static analysis [32], serve as the high-level specification of the access-control policy. Our analysis computes interprocedural control and data dependences of the check, extracting an interprocedural slice containing all program statements that implement the access-control logic. FixMeUp creates an access-control template (ACT) from these statements. The ACT serves both as a low-level policy specification and a program transformation template. FixMeUp then uses the ACT to (1) find security-sensitive operations not protected by appropriate access-control logic; (2) transform the program by inserting only the missing logic into the vulnerable calling contexts, while preserving the statements and dependences already present; and (3) verify the transformation did not accidentally introduce unwanted dependences, changing the semantics of the inserted policy.

We evaluate FixMeUp on ten real-world Web applications varying in size from 1,500 to 100,000+ lines of PHP code. We chose these benchmarks because (i) prior work used them to specify and/or infer access-control policies [32, 36]; (ii) they contain known access-control bugs that FixMeUp finds and repairs; and (iii) they demonstrate the scalability of FixMeUp.

FixMeUp found 38 access-control bugs and correctly repaired 30 of them. We confirmed all bugs and repairs by hand and with experimental testing on attack inputs. In particular, FixMeUp found and repaired 5 bugs in two benchmarks that prior analysis of the same code missed [36]. In 7 cases, the inserted access-control check was added to an existing, alternative check. In one case, our repair validation procedure automatically detected an unwanted control dependence and issued a warning. In 28 cases, FixMeUp detected that vulnerable code already contained one or more, but not all, of the statements prescribed by the access-control template and adjusted the repair accordingly. This result shows that detecting which parts of the access-control logic are already present and correct is critical to repairing access-control vulnerabilities. No prior program repair or transformation approach detects whether the desired logic is already present in the program [1, 14, 19, 25, 41].

FixMeUp guarantees that the repaired code implements the same access-control policy as the template, but it cannot guarantee that the resulting program is “correct.” For example, FixMeUp may apply the policy to a context where the developer did not intend to use it, or the repair may introduce an unwanted dependence into the program (adding an access-control check always changes the program’s control flow). Static analysis in FixMeUp is neither sound, nor complete because it does not consider language features such as dynamic class loading, some external side effects, or eval. The developer should examine the errors found by FixMeUp and the suggested repairs.

Using automated program analysis tools for verification and bug finding is now a well-established approach that helps programmers discover errors and improve code quality in large software systems. No prior tool, however, can repair access-control errors of omission. These errors may appear relatively simple, but our analysis shows that they are common in Web applications. FixMeUp is a new tool that can help Web developers repair common access-control vulnerabilities in their applications.

2 Overview of our approach

FixMeUp starts with a high-level specification of the access-control policy. A policy prescribes one or more access-control checks on execution paths leading to sensitive operations, such as database queries, links to privileged pages, operations that rewrite cookies and delete files. Sensitive operations must be specified in advance. If the checks fail, the program does not execute the sensitive operations. Because access-control logic varies between different user roles and entry points within the same application [32, 36], different paths may require different checks or no checks at all. Access-control logic in Web applications is often interprocedural and context-sensitive.

FixMeUp is agnostic about the source of the policy and works equally well with user-specified policies and with policies inferred by program analysis. Our focus in this paper is on program repair and not on the orthogonal problem of policy specification or inference.

For simplicity, assume that the high-level policy is specified explicitly by the developer who adds annotations to the PHP source code marking (1) access-control checks, (2) the protected sensitive operation, and (3) a tag indicating the user role to which the policy applies (e.g., root, admin, or blog poster). Section 3 presents examples of specifications and policies. FixMeUp assumes that each high-level policy applies throughout the indicated user role.

FixMeUp uses this specification to compute an access-control template (ACT). FixMeUp starts with the conditional statement performing the correct access-control check and computes all methods and statements in its backward, interprocedural slice. Given this slice, FixMeUp builds an interprocedural, hierarchical representation of all statements in the check’s calling context on which the check depends. The ACT is both a low-level policy specification and a program transformation template.

To find missing access-control checks, FixMeUp looks at every calling context in which a sensitive operation may be executed and verifies whether the access-control logic present in this context matches the ACT for the corresponding role. Of course, FixMeUp cannot decide general semantic equivalence of arbitrary code fragments. In practice,
the access-control logic of Web applications is usually very stylized and located close to the program entry points. The resulting templates are loop-free, consist of relatively few statements, and have simple control and data dependences (see Table 2). A few statements may have side effects on the global variables, such as opening database connections and initializing session state. For example, a typical Web application may open the database once and then permit only the authorized users to store into the database; these stores may be sprinkled throughout the application.

FixMeUp generates candidate repairs by replicating the access-control logic in program contexts where some or all of it is missing. If FixMeUp finds a vulnerable context that permits execution of some sensitive operation without an access-control check, it transforms the context using the access-control template. This transformation finds and reuses statements already present in the vulnerable code and only inserts the statements from the template that are missing. The repair procedure uses and respects all control and data dependences between statements.

To ensure that the reused statements do not change the meaning of the inserted policy, FixMeUp composes a fresh template starting from the access-control check and matches it against the original template. If the templates do not match, FixMeUp issues a warning. If they match, FixMeUp provides the transformed code to the developer as the suggested repair.

3 Access-Control Policies

Access control is the cornerstone of Web-application security. Several of the OWASP Top 10 Application Security Risks [24] are access-control bugs: broken authentication and session management, insecure direct object references, and failure to restrict URL accesses. Access-control bugs can expose other types of vulnerabilities, too.

3.1 Examples of correct policies and bugs

In general, an access-control policy requires some checks prior to executing security-sensitive operations. Web applications frequently implement multiple user roles. For example, an online store may have customers and administrators, while a blogging site may have blog owners, publishers, and commenters. Access-control policies are thus role-sensitive. Different calling contexts associated with different user roles often require different checks.

Figures 1 and 2 show examples of access-control checks in real-world PHP applications. Figure 1 shows a correct check (line 4) in Add.php from minibloggie. Add.php invokes a dedicated verifyuser function that queries the user database with the username and password. If verification fails, the application returns the user to the login page. Figure 2 shows a correct check (line 3) performed by AcceptBid.php in the DNscript application. It reads the hash table containing the session state and checks the member flag. Both access-control policies protect the same operation—a mysql_query call site that updates the back-end database—but with very different logic.

The access-control checks are role-specific. For example, the DNscript application has two roles. Figure 2 shows the check for the “regular user” role and Figure 3 shows the check for the “administrator” role. DelCb.php in Figure 2 shows an access-control bug in the DNscript application: the check on $SESSION for the “regular user” role is present in AcceptBid.php, but missing in DelCb.php. The developer either forgot the check or did not realize that any network user can directly invoke DelCb.php. The bottom of Figure 2 shows how FixMeUp repairs DelCb.php by replicating the correct access-control logic from AcceptBid.php (associated with the “regular user” role). Similarly, Figure 3 shows how FixMeUp repairs an access-control bug in AddCat2.php (associated with the “administrator” role) by replicating the access-control check from Del.php.

Invalid control flow distinguishes access-control vulnerabilities from data-flow vulnerabilities, such as cross-site scripting and SQL injection studied in prior work [13, 15, 17, 39, 42]. The access-control policy determines if the user is authorized to perform a particular operation, regardless of whether or not there are tainted data flows into the arguments of the operation.

3.2 Design patterns for access control

There is no standard access-control library or framework for Web applications, thus each application implements access-
control policies in its own, idiosyncratic way. The variables that hold users’ credentials and authorization information, as well as the semantics of access-control checks, vary significantly from application to application. Fortunately, they tend to follow a stylized code design pattern.

Access control is typically enforced near the program’s entry point. First, the program collects relevant information. For example, the SELECT query returns the user’s record from the administrative database in minibloggie in Figure 1, while the session state variable holds user data in DNscript in Figure 2. Typically, only a few security-critical variables hold access-control information—for example, variables $user, $pwd, and $result in minibloggie—and they are updated in a very small number of places. The corresponding program slice is thus relatively small. All of our benchmark applications exhibit these features (see Table 2).

Second, the application executes one or more conditional statements that evaluate a predicate over security-critical variables. These statements implement the actual access-control checks, e.g., line 4 of Figure 1 and line 3 of acceptBid.php in Figure 2. If the check fails, the program terminates or returns to the login page. Otherwise, it continues execution, eventually reaching the sensitive operation protected by the check. In many applications, these steps are distributed over multiple functions and files, e.g., the verifyuser function in Figure 1.

3.3 Specifying access-control policies

FIXMEUp takes as input an explicitly specified or inferred access-control policy. An access-control policy is a set of role-specific mappings from program statements executing security-sensitive operations—such as SQL queries and file operations—to one or more conditional statements that must be executed prior to these operations. Because this paper focuses on program repair and not on policy specification or inference (see Section 7 for a discussion of policy sources), we limit our attention to policies specified by ex-
explicit annotation.

The developer marks the access-control checks and the security-sensitive operations and assigns them a user-role tag. This high-level specification informs FixMEUp that the marked check must be performed before the marked operation in all calling contexts associated with the indicated user role. In Figure 4, line 8 of admin.php shows an annotation that marks the access-control check with the “admin” role tag. Lines 22 and 26 show the annotations for security-sensitive operations. FixMEUp does not currently support disjointive policies where operations may be protected by either check A or check B.

Unlike GuardRails [3], FixMEUp does not require an external specification of all statements involved in access-control enforcement. Instead, FixMEUp automatically computes access-control policies from the annotations marking the checks and the protected operations.

4 Access-Control Templates

This section describes how FixMEUp computes access-control templates. We implemented this analysis in PHC, an open-source PHP compiler [26], and analyze PHC-generated abstract syntax trees (AST). We started by adding standard call graph, calling context, data dependence, and control dependence analyses to PHC.

FixMEUp takes as input an explicit mapping from sensitive operations to correct access-control checks. FixMEUp then performs interprocedural program slicing on the call graph and on the data- and control-dependence graphs of each method to identify the program statements on which each access-control check is data- or control-dependent. FixMEUp converts each slice into a template, which serves as a low-level specification of the correct policy logic and a blueprint for repair. Informally, the template contains all statements in the check’s calling context that are relevant to the check: (1) statements on which the check is data- or control-dependent, and (2) calls to methods that return before the check is executed but contain some statements on which the check is dependent.

4.1 Computing access-control slices

Given a conditional access-control check, FixMEUp picks an entry which has the shortest call depth to check. FixMEUp iteratively computes the transitive closure of the statements on which check is control- or data-dependent. This analysis requires the call graph, control-flow graphs, intraprocedural aliases, and intraprocedural def-use chains. For each call site, FixMEUp computes an interprocedural summary of side effects, representing the def-use information for every parameter, member variable, and base variable at this site. These analyses are standard compiler fare and we do not describe them further.

In general, a slice may execute an arbitrary computation, but as we pointed out in Section 3, slices that perform access-control enforcement are typically loop-free computations that first acquire or retrieve user credentials or session state, and then check them. All of our benchmarks follow this pattern. Statements in these slices update only a local variable that has the shortest call depth to check. In general, a slice may execute an arbitrary computation, but as we pointed out in Section 3, slices that perform access-control enforcement are typically loop-free computations that first acquire or retrieve user credentials or session state, and then check them. All of our benchmarks follow this pattern. Statements in these slices update only a local variable that has the shortest call depth to check.
4.2 Computing access-control templates

Statements in a slice may be spread across multiple methods and thus do not directly yield an executable code sequence for inserting elsewhere. Therefore, FixMeUp converts slices into templates.

An access-control template (ACT) is a hierarchical data structure whose hierarchy mirrors the calling context of the access-control check. Each level of the ACT corresponds to a method in the context. For each method, the ACT records the statements in that method that are part of the slice. These statements may include calls to methods that return before the access-control check is executed, but only if the call subgraphs rooted in these methods contain statements that are part of the slice.

The last level of the ACT contains the access-control check and the failed-authorization code that executes if the check fails (e.g., termination or redirection). The developer optionally specifies the failed-authorization branch. Without such specification, FixMeUp uses the branch that contains a program exit call, such as die or exit. We label each ACT with the programmer-specified user role from the check’s annotation.

Formally, \( \text{ACT}_{\text{role}} \) is an ordered list of \((m_i, S_i)\) pairs, where \(m_i\) are method names and \(S_i \subseteq m_i\) are ordered lists of statements. Each \(m_i\) is in the calling context of check, i.e., it will be on the stack when check executes. Each statement \(s \in S_i\) is part of the access-control logic because (1) the check is data- or control-dependent on \(s\), or (2) \(s\) is a call to a method that contains such a statement somewhere in its call graph, but \(n\) returns before the check executes, or (3) \(s\) is a statement in the failed-authorization branch of check. Consider the following example:

```plaintext
main() {
  a = b;
  c = credentials(a);
  if (c) then fail(...);
  perform security-sensitive operation
}
```

The conditional statement if \((c)\) is the access-control check and its calling context is simply main. The computed template \(\text{ACT}_{\text{role}}\) includes the call to credentials, as well as fail(...) in the branch corresponding to the failed check. We add the following pair to the \(\text{ACT}_{\text{role}}\): \(\text{main}, \{a=b, c=credentials(a), if (c) then fail(...)\}\).

Figure 5 shows the algorithm that, given a calling context and a slice, builds an ACT. The algorithm also constructs data- and control-dependence maps, \(DD_{ACT}\) and \(CD_{ACT}\), which represent all dependences between statements in the ACT. FixMeUp uses them to (1) preserve dependences between statements when inserting repair code, and (2) match templates to each other when validating repairs. Figure 4 gives an example of an access-control slice and the corresponding ACT from Newscript 1.3.

5 Finding and Repairing Vulnerabilities

We first give a high-level overview of how FixMeUp finds vulnerabilities, repairs them, and validates the repairs, and then we describe each step in more detail.

FixMeUp considers all security-sensitive operations in the program. Recall that each sensitive operation is associated with a particular user role (see Section 3.3). For each operation, FixMeUp computes all of its calling contexts. For each context, it considers all candidate checks, computes the corresponding access-control template \(ACT'\), and compares it with the role’s access-control template \(ACT_{\text{role}}\). If some context \(CC_{tgt}\) is missing the check, its \(ACT'\) will not match \(ACT_{\text{role}}\). This context has an access-control vulnerability and FixMeUp targets it for repair.

To repair \(CC_{tgt}\), FixMeUp inserts the code from \(ACT_{\text{role}}\) into the methods of \(CC_{tgt}\). \(ACT_{\text{role}}\) contains the calling context \(CC_{src}\) of a correct access-control check and FixMeUp uses it to guide its interprocedural repair of \(CC_{tgt}\). FixMeUp matches \(CC_{src}\) method by method against \(CC_{tgt}\). At the last matching method \(m_{inline}\), FixMeUp inlines all statements from the methods deeper in \(CC_{src}\) than \(m_{inline}\) into \(m_{inline}\). We call this adapting the ACT to a target context. Adaptation produces a method map indicating, for each \(m_{src} \in ACT_{\text{role}}\), the method \(m_{tgt} \in CC_{tgt}\) where to insert statements from \(m_{src}\).

For each statement in \(ACT_{\text{role}}\), FixMeUp inserts state-
ments from $m_{src}$ into the corresponding $m_{tgt}$ only if they are missing from $m_{tgt}$. In the simplest case, if the vulnerable context has only the entry method and no code that corresponds to any code in $ACT_{role}$, FixMeUp inserts the entire template into the entry method.

A repair can potentially introduce two types of undesired semantic changes to the target code. First, statements already present in the target may affect statements inserted from the template. We call these unintended changes to the inserted policy. Second, inserted statements may affect statements already present in the target. We call these unintended changes to the program. Because our analysis keeps track of all data and control dependences and our repair procedure carefully renames all variables, we prevent most of these errors. As we show in Section 6, FixMeUp detects when template statements with side effects are already present in the target. We call these unintended changes to the program.

To validate that there are no unintended changes to an inserted policy, FixMeUp computes a fresh ACT from the repaired code and compares it with the adapted unin-

5.1 Matching templates
To find vulnerabilities and validate repairs, FixMeUp matches templates. In general, it is impossible to decide whether two arbitrary code sequences are semantically equivalent. Matching templates is tractable, however, because ACTs of real-world applications are loop-free and consist of a small number of assignments, method invocations, and conditional statements. Furthermore, when developers implement the same access-control policy in multiple places in the program, they tend to use structurally identical code which simplifies the matching process.

Figure 6 shows our template matching algorithm and the statement matching algorithm that it uses. The latter algorithm compares statements based on their data and control dependences, and therefore the syntactic order of statements does not matter. Matching is conservative: two matching templates are guaranteed to implement the same logic.

Let $ACT_x$ and $ACT_y$ be two templates. For every $s_x \in ACT_x$, FixMeUp determines if there exists only one matching statement $s_y \in ACT_y$, and vice versa. The developers may use different names for equivalent variables in different contexts, thus syntactic equivalence is too strict. Given statements $s_x \in ACT_x$ and $s_y \in ACT_y$, FixMeUp first checks whether the abstract syntax tree structures and operations of $s_x$ and $s_y$ are equivalent. If so, $s_x$ and $s_y$ are syntactically isomorphic, but can still compute different results. FixMeUp next considers the data dependences of $s_x$ and $s_y$. If the dependences also match, FixMeUp declares that the statements match. Table 1 shows the matching rules when neither statement has any dependences.

```plaintext
isMatchingACT (ACT_x, ACT_y) {
  // INPUT: two ACTs to be compared
  // OUTPUT: true if ACT_x and ACT_y match, false otherwise
  if (|ACT_x| ≠ |ACT_y|) return false;
  for (s_x ∈ ACT_x in order) {
    if (s_x ∈ ACT_y) return true
    if (s_x ∈ ACT_y) return false;
  } return true;
}

isMatching (s_x, s_y) {
  // INPUT: statements $s_x$ and $s_y$ to be compared
  // OUTPUT: true if $s_x$ and $s_y$ match, false otherwise
  VarMap ← φ
  StatementMap ← φ
  for (s_x ∈ ACT_x) {
    if (s_x ∈ ACT_y and isMatching(s_x, s_y)) {
      StatementMap ← StatementMap ∪ {(s_x, y)}
    } else {
      return false;
    }
  } return true;
}

// no data dependences
isMatching (s_x, s_y) {
  // INPUT: two ACTs to be compared
  // OUTPUT: true if $s_x$ and $s_y$ match, false otherwise
  VarMap ← φ
  StatementMap ← φ
  if (s_x ∈ ACT_x and s_y ∈ ACT_y) return true;
  if (DD_x ↦ DD_y) return true;
  if (DD_y ↦ DD_x) return true;
  if (DD_x ↦ DD_y) return true;
  if (DD_y ↦ DD_x) return true;
  return false;
}
```

Figure 6: Matching access-control templates

5.2 Finding access-control vulnerabilities
For each security-sensitive operation ($sso$), FixMeUp computes the tree of all calling contexts in which it may execute by (1) identifying all methods that may directly invoke $sso$ and (2) performing a backward, depth-first pass over the call graph from each such method to all possible program entries. FixMeUp adds each method to the calling context once, ignoring cyclic contexts, because it only needs to verify that the access-control policy is enforced once before $sso$ is executed.

For each calling context $CC$ in which $sso$ may be executed, FixMeUp first finds candidate access-control checks. A conditional statement $b$ is a candidate check if it (1) controls whether $sso$ executes or not, and (2) is syntactically equivalent to the correct check given by the $ACT_{role}$. For each such $b$, FixMeUp computes its slice, converts it into $ACT_b$ using the algorithms in Figure 5, and checks
whether \( ACT_{t} \) matches \( ACT_{t}^{\text{role}} \). If so, this context already implements correct access-control logic. Otherwise, if there are no candidate checks in the context or if none of the checks match the correct check, the context is vulnerable and FixMeUp performs the repair.

F

\[
\text{Table 1: Matching statements without dependences}
\]

<table>
<thead>
<tr>
<th>method_1(C_0, ..., C_i)</th>
<th>method_2(C'_0, ..., C'_i)</th>
<th>Match if (1) ( \text{method}_1 = \text{method}_2 ) and (2) all constants ( C_k = C'_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>localvar_a = C ∈ method_1</td>
<td>localvar_b = C' ∈ method_2</td>
<td>Match if (1) ( \text{method}_1 = \text{method}_2 ) or both methods are entry methods and (2) constants ( C = C' )</td>
</tr>
<tr>
<td>globalvar_a = C ∈ method_1</td>
<td>globalvar_b = C' ∈ method_2</td>
<td>Match if (1) ( \text{globalvar}_a = \text{globalvar}_b ) and (2) constants ( C = C' )</td>
</tr>
</tbody>
</table>

Figure 7: Repairing vulnerable code and validating the repair

5.3 Applying the template

Formally, \( CC_{src} = \{ (cs_1, m_0) \ldots (check, m_n) \} \), \( CC_{tgt} = \{ (cs'_1, m'_0) \ldots (ssos, m'_l) \} \), where \( cs_{i+1} \in m_i \) and \( cs'_{i+1} \in m'_i \) are the call sites of \( m_{i+1} \) and \( m'_{i+1} \) respectively. For simplicity, we omit the subscript from \( ACT_{t} \).

FixMeUp uses DoRepair in Figure 7 to carry out a repair. It starts by adapting \( ACT_{t} \) to the vulnerable calling context \( CC_{tgt} \). If \( CC_{tgt} \) already invokes some or all of the methods in \( ACT_{t} \), we do not want to repeat these calls because the policy specifies that they should be invoked only once in a particular order. After eliminating redundant method invocations, FixMeUp essentially inlines the remaining logic from \( ACT_{t} \) into \( ACT_{t}^{\text{adapted}} \).

Formally, the algorithm finds common method invocations in \( CC_{src} \) and \( CC_{tgt} \) by computing the deepest \( m_{lin} \in CC_{src} \) such that for all \( i \leq \text{inline} \) \( m_i \) matches \( m'_i \). For \( i = 0 \), \( m_0 \) and \( m'_0 \) match if they are both entry methods. For \( i \geq 1 \), \( m_i \) and \( m'_i \) match if they are invocations of exactly the same method. The first for loop in AdaptACT from Figure 8 performs this process.

The algorithm then adapts \( ACT_{t} \) to \( CC_{tgt} \) by inlining the remaining statements—those from the methods deeper than \( m_{lin} \) in \( ACT_{t} \)—into \( m_{lin} \). The second for loop in AdaptACT from Figure 8 performs this process and produces \( ACT_{t}^{\text{adapted}} \). While matching methods and inlining statements, FixMeUp records all matching method pairs \( (m_i, m'_i) \), including \( m_{lin} \), in MethodMap.

In the simplest case, the entry \( m'_0 \in CC_{tgt} \) is the only method matching \( m_{lin} = m_0 \). In this case, FixMeUp inlines every statement in \( ACT_{t} \) below \( m_0 \) and produces a flattened \( ACT_{t}^{\text{adapted}} \).

Otherwise, consider the longest matching method sequence \( (m_0 \ldots m_{lin}) \) and \( (m'_0 \ldots m'_{lin}) \) in \( CC_{src} \) and \( CC_{tgt} \), respectively. For \( 1 \leq i \leq \text{inline} - 1 \), \( m_i \) and \( m'_i \) are exactly the same; only \( m_0 \) and \( m_{lin} \) are distinct from \( m'_0 \) and \( m'_{lin} \), respectively. AdaptACT stores the \( (m_0, m'_0) \) and \( (m_{lin}, m'_{lin}) \) mappings in MethodMap.

FixMeUp uses the resulting template \( ACT_{t}^{\text{adapted}} \) to repair the target context using the ApplyACT algorithm in Figure 9. This algorithm respects the statement order, control dependences, and data dependences in the template. Furthermore, it avoids duplicating statements that are already present in the target methods.

The algorithm iterates \( m_{src} \) over \( m_0 \) and \( m_{lin} \) in \( ACT_{t}^{\text{adapted}} \) because, by construction, these are the only methods that differ between the template and the target. It first initializes the insertion point \( ip_{tgt} \) in \( m_{tgt} \) corresponding to \( m_{src} \) in MethodMap. The algorithm only inserts statements between the beginning of \( m_{tgt} \) and the sensitive operation \( sso \), or—if \( m_{tgt} \) calls other methods to reach \( sso \)—the call site of the next method in the calling context of \( sso \). Intuitively, the algorithm only considers potential insertion points and matching statements that precede \( sso \).

Before FixMeUp inserts a statement \( s \), it checks if there
AdaptACT \(ACT_{src}, CC_{tgt}\) \{
  1 // Adapt \(ACT_{src}\) to the target context \(CC_{tgt}\)
  2 \(ACT \leftarrow \) clone \(ACT_{src}\)
  3 \(CC_{src} = ACT.CC_{src}\)
  4 \(l \leftarrow 0\)
  5 for \((i = 0; i < \lceil CC_{src}.size() \rceil); i++\) \{
    6 // iterate from the entry to the bottom method in \(CC_{src}\)
    7 \(m_i \leftarrow i^{th}\) method in \(CC_{src}\)
    8 \(m_{tgt} \leftarrow i^{th}\) method in \(CC_{tgt}\)
    9 if \((m_i \text{ and } m_{tgt} \text{ are entries or } m_i \equiv m_{tgt})\) \{
      10 \(\text{MethodMap} \leftarrow \text{MethodMap} \cup \{(m_i, m_{tgt})\}\)
      11 \(l \leftarrow i\)
      12 } else break;
      13 \}
  14 \(m_{mini} \leftarrow l^{th}\) method in \(CC_{tgt}\)
  15 for \((k = l+1; k < \lceil CC_{tgt}.size() \rceil); k++\) \{
    16 inline method \(m_k\) from \(CC_{src}\) into \(m_{mini}\) in \(ACT\)
    17 \(\text{MethodMap} \leftarrow \text{MethodMap} \cup \{(m_k, m_{mini})\}\)
    18 \(l \leftarrow k\)
    19 return \(ACT\)
  20 \}
\}

Figure 8: Adapting ACT to a particular calling context

already exists a matching statement \(s' \in m_{tgt}\). If so, FixMeUp adds \(s\) and \(s'\) to StatementMap, sets the current insertion point \(ip_{tgt}\) to \(s'\), and moves on to the next statement. Otherwise, it inserts \(s\) as follows:

1. Transform \(s\) into \(s'\) by renaming variables.
2. If \(s\) is a conditional, insert empty statements on the true and false branches of \(s'\).
3. If \(ip_{tgt}\) has not been set yet, insert \(s'\) at the top of \(m_{tgt}\).
4. Otherwise, if \(s\) is immediately control-dependent on some conditional statement \(t\), insert \(s'\) as the last statement on the statement list of the matching branch of the corresponding conditional \(t' \in m_{tgt}\).
5. Otherwise, insert \(s'\) after \(ip_{tgt}\), i.e., as the next statement on the statement list containing \(ip_{tgt}\). For example, if \(ip_{tgt}\) is an assignment, insert \(s'\) as the next statement. If \(ip_{tgt}\) is a conditional, insert \(s'\) after the true and false clauses, at the same nesting level as \(ip_{tgt}\).
6. Add \((s, s')\) to StatementMap and set \(ip_{tgt}\) to \(s'\).

ApplyACT returns the repaired AST, the inserted check, and the number of reused statements.

**Variable renaming.** When FixMeUp inserts statements into a method, it must create new variable names that do not conflict with those that already exist in the target method. Furthermore, because FixMeUp, when possible, reuses existing statements that match statements from the ACT semantically (rather than syntactically), it must rename variables. Lastly, as the algorithm establishes new names and matches, it must rewrite subsequent dependent statements to use the new names. The isMatching function in Figure 6 establishes a mapping between a variable name from the template and a variable name from the target method when it matches assignment statements.

As FixMeUp inserts subsequent statements, it uses the variable map to replace the names from the template. Before ApplyACT inserts a statement, it calls RenameVars to remap all variable names to the names used by the target method. For unmapped variables, RenameVars creates fresh names that do not conflict with the existing names.

**Dealing with multiple matching statements.** In theory, there may exist multiple statements in \(m_{tgt}\) that match \(s\).
FindMatchingStmt($s, ip_{tgt}, m_{tgt})$

```plaintext
1  // INPUT:
2  $s$: statement in ACT
3  ip_{tgt}, last inserted statement in m_{tgt}
4  if ($m_{tgt}$ contains the sensitive operation $sso$)
5    $SL = \{ \text{statements in } m_{tgt} \text{ after } ip_{tgt} \text{ that dominate } sso\}$
6  else
7    $SL = \{ \text{statements in } m_{tgt} \text{ after } ip_{tgt} \text{ that dominate the callsite of next method in } CCl_{tgt} \}$
8  for ($t \in SL$) {
9    if (isMatching($s, t$)) {
10     StatementMap $\leftarrow$ StatementMap $\cup \{(s, t)\}$
11     return $t$
12    }
13  }
14  // If multiple statements in $SL$ match $s$, they are handled as described in Section 5.3
15  return null
16 } 17 }
```

Figure 10: Matching statements

and thus multiple ways to insert $ACT_{adapted}$ into the target context. Should this happen, FixMeUp is designed to exhaustively explore all possible matches, generate the corresponding candidate repairs, and validate each candidate. FixMeUp picks the validated candidate that reuses the most statements already present in the target and suggests it to the developer.

### 5.4 Validating Repairs

As mentioned above, FixMeUp can potentially introduce two types of semantic errors into the repaired program: (1) unintended changes to the inserted policy, and (2) unintended changes to the program. Unintended changes to the inserted policy may occur when existing statements change the semantics of the inserted code. Unintended changes to the program may occur when the inserted code changes the semantics of existing statements.

To detect type (1) errors, FixMeUp computes afresh an ACT from the repaired code and compares it—using ValidateRepair from Figure 7—with the ACT on which the repair was based. An ACT captures all control and data dependences. Any interference from the existing statements that affects the inserted code must change the dependences of the inserted statements. For example, suppose the reused statement has dependent statements already in the program that are not part of the ACT. In this case, the ACTs will not match and FixMeUp will issue a warning. This validation procedure guarantees that reusing an existing statement is always safe. We examined all 38 repairs suggested by FixMeUp for our benchmarks (see Section 6) and in only one case did the insertion of the repair code change the ACT semantics. FixMeUp’s validation algorithm detected this inconsistency and issued a warning.

With respect to type (2) errors, unintended changes to the program, observe that the actual purpose of the repair is to change the program’s semantics by adding access-control logic. FixMeUp therefore cannot guarantee that the repaired program is free from type (2) errors because it cannot know the full intent of the programmer.

The purpose of repair is to introduce a new dependence: all statements after the inserted access-control check become control-dependent on the check, which is a desired semantic change. Because FixMeUp inserts the check along with the statements defining the values used in the check, the inserted access-control logic may change both control and data dependences of statements that appear after the check. Our repair procedure minimizes the risk of unintended dependences by reusing existing statements as much as possible and by renaming all variables defined in the template to fresh names, thus preventing unintended dependences with the variables already present in the program. In just one of the 38 repairs on our benchmarks (see Figure 14 in Section 6) did an incorrectly annotated role cause FixMeUp to “repair” a context that already implemented a different access-control policy and thus introduce unwanted changes to the program.

### 5.5 Discussion and limitations

Good program analysis and transformation tools help developers produce correct code. They are especially useful for subtle semantic bugs such as inconsistent enforcement of access-control policies, but developers must still be intimately involved in the process. The rest of this section discusses the general limitations of any automated repair tool and the specific limitations of our implementation.

**Programmer burden.** Suggesting a repair, or any program change, to developers requires some specification of correct behavior. We rely on developers to annotate access-control checks and security-sensitive operations in their applications and tag them with the corresponding user role. We believe that this specification burden is relatively light and, furthermore, it can be supported by policy inference tools [32]. We require that the specifications be consistent for all security-sensitive operations in a given role. If the programmer wants different checks in different contexts for the same operation, the specification won’t be consistent and our approach will attempt to conservatively over-protect the operation. For example, Figure 11 shows that FixMeUp inserts a credential check performed in one context into a different context that already performs a CAPTCHA check, in this case introducing an unwanted duplicate check. Developers should always examine suggested repairs for correctness.

We believe that the consequences of access-control errors are sufficiently dire to motivate the developers to bear this burden in exchange for suggested code repairs, since it is easier to reject or manually fix a suggested change than it is to find the error and write the entire repair by hand. The latter requires systematic, tedious, error-prone examination of the entire program and its call graph. Language features
of PHP, such as the absence of a proper module system, dynamic typing, and eval, further complicate this process for PHP developers. The number of errors found by FixMEUp in real-world PHP applications attests to the difficulty of correctly programming access control in PHP.

Static analysis. FixMEUp uses a standard static interprocedural data- and control-dependence analysis to extract the program slice representing the access-control logic. Because this analysis is conservative, the slice could contain extraneous statements and therefore would be hard to apply as a transformation. Program slicing for more general debugging purposes often produces large slices [34]. Fortunately, access policies are typically self-contained and much more constrained. They consist of retrieving stored values into local variables, checks on these variables, and code that exits or restarts the program after the check fails. Consequently, access-control templates tend to be short (see Table 2).

Our FixMEUp prototype does not handle the dynamic language features of PHP, nor does it precisely model all system calls with external side effects. In particular, the analysis resolves dynamic types conservatively to build the call graph, but does not model eval or dynamic class loading, which is unsound in general. In practice, only myBB uses eval and we manually verified that there are no call chains or def-use chains involving eval that lead to security-sensitive operations, thus eval does not affect the computed ACTs.

Static analysis can only analyze code that is present at analysis time. PHP supports dynamic class loading and thus potentially loads classes our code does not analyze. However, our benchmarks use dynamic class loading in only a few cases, and we do analyze the classes they load. To handle these cases, we annotated 18 method invocations with extraneous statements and therefore would be hard to apply as a transformation. Program slicing for more general debugging purposes often produces large slices [34]. Fortunately, access policies are typically self-contained and much more constrained. They consist of retrieving stored values into local variables, checks on these variables, and code that exits or restarts the program after the check fails. Consequently, access-control templates tend to be short (see Table 2).

Our analysis models database connections such as open, close, and write, file operations that return file descriptors, etc., but it does not perform symbolic string analysis on the arguments. This is a possible source of imprecision. For example, consider two statements: writeData("a.txt", $data) and $newdata = readData($b). If $b is “a.txt”, the second statement is data-dependent on the first. A more precise algorithm would perform symbolic analysis to determine if the two statements may depend on each other and conservatively insert a dependence edge. Not doing this makes our analysis unsound in general, but in practice, we never observed these types of dependences. Therefore, even a more conservative analysis would have produced the same results on our benchmarks.

Statement matching is weaker than semantic equivalence. For example, our matching algorithm does not capture that statements \( a = b + c \) and \( a = \text{add}(b, c) \) are equivalent. Another minor limitation of our matching algorithm is the use of coarse-grained statement dependences instead of variable def-use chains on the remapped variable names. A more precise algorithm would enforce consistency between the def-use information for each variable name \( \text{var}_x \) used in \( s_x \) and \( \text{var}_y \) used in \( s_y \), even if the names are not the same given the variable mapping produced thus far. The current algorithm may yield a match with an inconsistent variable mapping in distinct statements and thus change the def-use dependences at the statement level. We never encountered this problem in practice and, in any case, our validation procedure catches errors of this type.

6 Evaluation

We evaluate FixMEUp on ten open-source interactive PHP Web applications, listed in Table 2. We chose SCARF, YaPiG, AWCM, minibloggie, and DNscript because they were analyzed in prior work on detecting access-control vulnerabilities [32, 36]. Unlike FixMEUp, none of the previous techniques repair the bugs they find. In addition to repairing known vulnerabilities, FixMEUp found four new vulnerabilities in AWCM 2.2 and one new vulnerability in YaPiG that prior analysis [36] missed. We added Newscript and phpCommunityCal to our benchmarks because they have known access-control vulnerabilities, all of which FixMEUp repaired successfully. To test the scalability of FixMEUp, we included two relatively large applications, GRBoard and myBB. Table 2 lists the lines of code (LoC) and total analysis time for each application, measured on a Linux workstation with Intel dual core 2.66GHz CPU with 2 GB of RAM. Analysis time scales well with the number of lines in the program.

Our benchmarks are typical of server-side PHP applications: they store information in a database or local file and manage it based on requests from Web users. Table 2 shows that four applications have a single access-control policy that applies throughout the program. The other six have two user roles each and thus two role-specific policies. Policies were specified by manual annotation. They are universal, i.e., they prescribe an access-control check that must be performed in all contexts associated with the given role.

FixMEUp finds 38 access-control bugs, correctly repairs 30 instances, and issues one warning. Nine of the ten benchmarks contained bugs. Seven bugs were previously unknown. As mentioned above, five of the previously unknown bugs appeared in applications that had been analyzed in prior work which missed the bugs. Five of the ten applications implement seven correct, but alternative policies in some of their contexts (i.e., these policies differ from the policy in the template).

The fourth and fifth columns in Table 2 characterize the access-control templates; the third column lists the user role...
AddDn.php

```php
1 <?
2 session_start();
3 if ("$_SESSION[‘member’]"
4   header(‘Location: login.php’);
5   exit;
6   ) ...?
7 ?>
```

Process.php

```php
1 <?
2 session_start(); // existing statement
3 if ("$_SESSION[‘member’]"
4   header(‘Location: login.php’); // [FixMeUp repair]
5     exit; // [FixMeUp repair]
6   ) ...
7 ...
8 $number = $POST[‘image’];
9 if (md5($number) != "$_SESSION[‘image_random_value’]"
10   echo ‘Verification does not match.. Go back and refresh your browser and then retype your verification’;
11   exit();
12 }
13 ?>
```

Figure 11: DNscript: Different access-control checks within the same user role

to which each policy applies. Six applications have two policies, admin or normal. The fourth column shows the total instances of the template in the code, showing that developers often implement the same access-control logic in multiple places in the program. For example, the DNscript application has two roles and thus two role-specific access-control policies. Out of the 22 templates in DNscript, only 2 are unique. The “LoC” column shows the size of each template (in AST statements). The templates are relatively small, between 2 and 11 statements each.

The “missing checks” and “alternative policies” columns in Table 2 show that FixMeUp finds a total of 38 missing checks. The “alternative policies” column shows that in seven cases FixMeUp inserts an access-control policy, but that the target code already has a different check. Figure 11 shows a code example of this case, where process.php is repaired using the policy from AddDn.php. However, it already contained a different, CAPTCHA-based check.

The “inserted polices” columns shows that FixMeUp made 37 validated repairs with one warning, 30 of which fixed actual vulnerabilities. For the other 7, the program already contained alternative logic for the same role (e.g., CAPTCHA vs. login). The case that generated the warning is shown in Figure 12. FixMeUp only inserts statements that are missing from the target. In minibloggie, the statements `session_start()` and `dbConnect()` are both in the template and in Del.php, thus FixMeUp does not insert them. It only inserts the missing statement `if (!verifyuser())
{header (‘Location: ./login.php’);}. The access-control check at line 10, however, now depends on the conditional at line 7. This dependence did not exist in the original ACT and does not pass FixMeUp validation.

The “partial” and “full” columns shows that, in 28 of 38 attempted repairs, FixMeUp reused some of the existing statements in the target when performing the repair, and only in 9 cases did it insert the entire template. This reuse demonstrates that repairs performed by FixMeUp are not simple clone-and-patch insertions, and adapting the tem-
already present alternative policy. Line 13 shows an access-control check already present in slideshow.php. Because the policy implemented by the existing check does not match the ACT that prescribes additional checks for the administrator role, FixMeUp inserts Line 3-11. However, the function call on Line 8 has a side effect on $$_{SESSION}$$ and $$$_{COOKIE}$$ which are used in the function call at Line 13. This side effect is easy to detect with standard dependence analysis, but the reason it occurred is a faulty annotation: the access-control policy represented by the ACT should not have been applied to this context.

We reported the new vulnerabilities found by FixMeUp and they were assigned CVE candidate numbers: CVE-2012-2443, 2444, 2445, 2437 and 2438. We confirmed the correctness of our repairs by testing each program and verifying that it is no longer vulnerable. When an unauthorized user invokes the repaired applications through either an intended or unintended entry point and attempts to execute the sensitive operation, every repaired application rejects the attempt and executes the code corresponding to the failed check from the original ACT.

7 Related Work

Related work includes techniques for finding access-control bugs, since it is a necessary first step to repairing them, general bug finding, program repair, and transformation tools.
Static detection of access-control bugs. Prior work simply reports that certain statements are reachable without an access-control check. Sun et al. require the programmer to specify the intended check for each application role and then automatically find execution paths with unchecked access to the role’s privileged pages [36]. Chlipala finds security violations by statically determining whether the application’s behavior is consistent with a policy specified as a database query [5].

One consequence of access-control bugs in Web applications is that attackers may perform unintended page navigation. Several approaches find these unintended navigation flows [2, 10]. They generally rely on heuristics and/or dynamic analysis to learn the intended flows and are thus incomplete. Furthermore, they cannot detect finer-grained access-control bugs. For example, a missing check on the same page as the protected operation will not manifest as an anomalous page navigation.

Without a programmer-provided specification, static analysis may infer the application’s access-control policies. Son and Shmatikov use consistency analysis to find variables in access-control logic [33]. Son et al. developed RoleCast, a tool that finds role-specific access-control checks without specification by exploiting software engineering conventions common in Web applications [32].

None of these approaches automatically repair the bugs they find, whereas FixMeUp (1) computes code templates that implement access-control logic, (2) finds calling contexts that implement this logic incorrectly, (3) transforms the code by inserting the template into one or more methods in the vulnerable contexts, and (4) validates that the transformed code implements the correct logic.

Code mining. A popular bug finding approach is to mine the program for patterns and looks for bugs as deviations or anomalies. This approach typically finds frequently occurring local, intraprocedural patterns [9]. Tan et al. showed how to find access-control bugs in SELinux using similar techniques, but with interprocedural analysis [37]. When applied to Web applications, heuristics based on finding deviations from common, program-wide patterns will likely generate an overwhelming number of false positives. As shown in [36] and [32], access-control logic in Web applications is significantly more sophisticated than simple “this check must always precede that operation” patterns. They are role- and context-sensitive, with different policies enforced on different execution paths. Simple pattern matching won’t find violations of such policies.

Verifying access control in Java libraries. Access-control checks are standardized in Java libraries and are simply calls to the SecurityManager class. A rich body of work developed techniques for verifying access control in Java class libraries [16, 29, 31, 35], but none of them attempt to repair access-control bugs.

Dynamic detection of access-control bugs. In the security domain, dynamic analysis finds security violations by tracking program execution [4, 7, 12, 43]. For example, Hallé et al. dynamically ensure that page navigation within the application conforms to the state machine specified by the programmer [12]. GuardRails requires the developers to provide explicit access-control specifications and enforces them dynamically within its framework for Ruby [3]. Alternatives to explicit specification include learning the state machine by observing benign runs and then relying on anomaly detection to find violations [6], or using static analysis of the server code to create a conservative model of legitimate request patterns and detecting deviations from these patterns at runtime [11]. Violations caused by missing access-control checks are an example of generic “execution omission” bugs. Zhang et al. presented a general dynamic approach to detecting such bugs [44].

In addition to the usual challenges of dynamic analysis, such as incomplete coverage, dynamic enforcement of access-control policies is limited in what it can do once it detects a violation. Typically, the runtime enforcement mechanism terminates the application since it does not know what the programmer intended for the application to do when an access-control check fails.

By contrast, our objective is to repair the original program. In particular, for the program branch corresponding to a failed access-control check, we insert the exact code used by the programmer as part of the correct checks (it may generate an error message and return to the initial page, terminate the program, etc.). The repaired program thus behaves as intended, does not require a special runtime environment, and can be executed anywhere.

Dynamic repair of software bugs. Dynamic program
repair fixes the symptom, but not the cause of the error [4, 7, 12, 22, 30, 43]. For example, dynamic repair allocates a new object on a null-pointer exception, or ignores out-of-bounds references instead of terminating the program. The dynamic fixes, however, are not reflected in the source code and require a special runtime.

Static detection of injection vulnerabilities. Many techniques detect data-flow vulnerabilities, such as cross-site scripting and SQL injection [13, 15, 17, 39, 42]. These bugs are characterized by tainted inputs flowing into database queries and HTML content generation. Access-control bugs are control-flow vulnerabilities: they enable the attacker to execute a sensitive operation, which may or may not be accompanied by illegitimate data flows. For example, if a constant query deletes the database, there is no tainted data flow into the operation.

Automatic remediation of software bugs. Much prior work finds code clones within the same application to help programmers refactor, fix bugs, and add features consistently [8, 18, 20, 23, 38]. These tools suggest where a bug fix may be needed, but do not transform the program. FixMeUp solves the dual of this problem: it inserts similar code where it is missing.

Several tools learn from a developer-provided fix and help apply similar fixes elsewhere. They perform the same syntactic edit on two clones [21], or suggest changes for API migration [1], or do not perform the edit [23], or ask users where to apply the edit [19]. These approaches only apply local edits and none of them consider the interprocedural edits that are required to repair access-control logic. In the more limited domain of access-control bugs, we automate both finding the missing logic and applying the fix.

Generating program fixes. A few approaches automatically generate a candidate patch and then check correctness with compilation and testing. For example, Perkins et al. generate patches to enforce invariants that are observed in correct executions but violated in erroneous ones [25]. They test several patched executions and select the most successful one. Weimer et al. [40, 41] generate candidate patches by randomly replicating, mutating, or deleting code from the program. Jin et al. automatically fix bugs by finding violations of pre-defined patterns encoded as finite-state machines, such as misplaced or missing lock and unlock pairs [14]. Their static analysis moves or inserts one or two lines of code to satisfy the correct pattern. All of these approaches focus on one- or two-line changes that satisfy some dynamic or static local predicate. By contrast, FixMeUp extracts and inserts multi-line code sequences responsible for enforcing the application’s context-sensitive access-control policy.

8 Conclusion
We presented FixMeUp, the first static analysis and program transformation tool for finding and repairing access-control bugs in server-side Web applications. FixMeUp starts with an access-control policy that maps security-sensitive operations—such as database queries and privileged file operations—to access-control checks that protect them from unauthorized execution. FixMeUp then automatically extracts the code responsible for access-control enforcement, uses it to create an access-control template, finds calling contexts where the check is missing or implemented incorrectly, repairs the vulnerability by applying the template, and validates the repair. The key to semantically correct repairs is the novel algorithm that finds and reuses existing statements that are part of the access-control logic. In particular, reuse of existing statements helps FixMeUp avoid duplicating statements that have side effects on the rest of the program. FixMeUp successfully repaired 30 access-control bugs in 10 real-world PHP applications, demonstrating its practical utility.

Acknowledgments. This research was partially supported by the NSF grants CNS-0746888, SHF-0910818, CCF-1018271, and CNS-1223396, a Google research award, the MURI program under AFOSR Grant No. FA9550-08-1-0352, and the Defense Advanced Research Agency (DARPA) and SPAWAR Systems Center Pacific, Contract No. N66001-11-C-4018.

References
[10] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. To-


