An Empirical Study of Multi-Language Security Patches in Open Source Software

Shiyu Sun[1], Yunlong Xing[1], Grant Zou[2], Xinda Wang[3], and Kun Sun[1]

[1] George Mason University, Fairfax, VA, USA
[2] University of Virginia, Charlottesville, VA, USA
[3] University of Texas at Dallas, Richardson, TX, USA

Abstract. Vulnerabilities in software repositories written in multiple programming languages present a major challenge to modern software quality assurance, especially those resulting from interactions between different languages. Existing static and dynamic program analysis tools are generally constrained to single-language analysis, while current deep-learning models lack the capability to process cross-language interactions effectively. To gain deeper insights into vulnerability patterns and patching behaviors in multi-language code, we conduct a measurement study on commits associated with multi-language security patches. We first collect a large-scale dataset of multi-language security patches from the MITRE corporation. We then analyze trends in language combinations, assess their proneness to vulnerabilities, and compare the severity of these vulnerabilities to those in single-language patches. Additionally, we classify patch patterns based on the types of language interactions to support automated program repair. To encourage further research, we release our dataset to the community, fostering deeper investigation into multi-language security patch development and enhancement.

Keywords: Multi-Languages Open Source Software · Security Patches.

1 Introduction

Multi-language development is increasingly prevalent and essential in modern software deployment, as applications often integrate multiple technologies to meet performance, scalability, and functionality demands [6]. Among the millions of commits involving modifications across multiple languages [5], approximately 6-18% [31,8] are intended to fix cross-language vulnerabilities, remedying critical risks caused by interactions between different languages.

Several efforts have been made to analyze multi-language software. Li et al. [17] studied multi-language software and revealed a statistical connection between language selection, vulnerability proneness, and interfacing mechanisms of multilingual projects. However, it cannot represent the association at the commit level since not all commits in the multilingual projects involve multiple programming languages. Moreover, other papers have worked on detecting multi-language vulnerabilities [23,22,33,12,32]. However, they either only focus on one

language combination (*e.g.,* Python-C or Java-C) or neglect the fixes for the bug. Additionally, they fail to explain why and how vulnerabilities are triggered or fixed across multiple languages. Furthermore, large language models (LLMs) can analyze code and have different domain knowledge to understand multiple programming languages, but they struggle to capture the full context when the input is large. Therefore, we can conclude that there is no existing labeled multi-language security patch dataset and the existing tools cannot be transferred to analyze different language combinations.

In this paper, we fill this gap by constructing a multi-language security patch dataset to reveal why different programming languages have been committed together and how they work together to fix vulnerabilities. Since different languages can interact with other programming languages across files and in the same file, we consider two types of commits: (1) commits that modify multiple files written in different programming languages, and (2) commits that modify a single file containing inline code from another language. First, we perform a statistical analysis of the patches, examining their language combinations, patch complexity, vulnerability categories, and severity levels. Second, we manually review each security patch to understand how vulnerabilities are fixed based on interfacing mechanisms. This analysis provides insight into the critical context needed for vulnerability remediation and its relationship with different types of vulnerabilities. Finally, we propose a taxonomy summarizing common fixing patterns, which can help advance automated program repair research.

According to the statistics from IEEE Spectrum [11] and the available data from MITRE, we focus on 16 popular programming languages, including C, C#, C++, Go, HTML, Java, JavaScript, Kotlin, Perl, PHP, Python, Ruby, Rust, Shell, SQL, and TypeScript. We collect 2,798 multi-language security patches, consisting of 1,253 multi-language security patches across multiple files and 1,545 in a single file. We summarize 11 fixing patterns, including eight for multi-file fixing and three patterns for single-file fixing.

Among our findings, we identify that multi-language security patches are more complex than single-language security patches in terms of lines of modification, branch structure changes, and dependency updates. Multi-language security patches are also more prone to injection-related vulnerabilities (*e.g.*, OS command injection and SQL injection), web-related vulnerabilities (*e.g.*, cross-site scripting (XSS) and cross-site request forgery (CSRF)), and authentication vulnerabilities. The severity of multi-language vulnerabilities is similar to that of single-language vulnerabilities. From our manual analysis of patches, we further observe that multi-language security patch patterns are associated with their interfacing mechanisms. Single-language security patch patterns can combine to form part of a multi-language security patch pattern (*e.g.*, one language sanitizes the input while another validates it). Additionally, non-fixing changes are often entangled with multi-language security fixes, making code review more difficult. The findings of our analysis provide insights that suggest paths forward for the security community to improve vulnerability management.

Our work provides several contributions. First, we collect the first dataset of multi-language security patches across 16 languages. Second, we analyze the characteristics of multi-language security patches. Moreover, we summarize the fixing patterns of multi-language security patches, revealing the reasons why different languages have been committed together and highlighting the need for automated tools to analyze multi-language programs. We release the dataset at https://figshare.com/Multi-language_Security_Patch_Dataset.

2 Background

2.1 Language Interoperability and Interfacing Mechanisms

Multilingual software leverages the distinct capabilities of various programming languages. To enable these languages to work together smoothly, they require effective inter-language interaction or interoperability mechanisms to ensure cohesive functionality [24]. We categorize four common interaction mechanisms with code-level evidence: client-server, foreign function interface (FFI), database queries, and subprocess execution.

Client-Server. The interaction occurs in three ways: (1) from the client to the server, (2) from the server to the client, and (3) bidirectionally between both. Client-side code is typically implemented using HTML for structure, JavaScript or TypeScript for dynamic behavior, whereas server-side code leverages languages and frameworks such as JavaScript/Node.js, Python, Java, PHP, C#, Rust, Go, or Ruby. These components interact bidirectionally, with the client sending requests to the server and the server processing and returning data or actions. There are several commonly used methods, often via HTTP requests.

Foreign Function Interfaces (FFI). An FFI is a mechanism that allows code written in one programming language to call functions or use services written in another language. FFI is crucial in scenarios where developers leverage existing libraries, optimize performance, or integrate systems written in multiple languages. For example, a Python application might use FFI to call a high-performance C library for numerical computations, or a Rust program might expose functions via FFI to be used in a Node.js application.

Database Queries. Python, Java, JavaScript, PHP, Ruby, Go, Rust, and C#, have libraries or frameworks that enable them to access databases and execute SQL commands. Some of these libraries provide direct SQL execution (*e.g.*, psycopg2 in Python), while others offer object-relational mapping (ORM) (*e.g.*, Entity Framework in C# or ActiveRecord in Ruby) to simplify database interactions by using an object-oriented approach. In this scenario, SQL statements are often composed as strings in the function call.

Subprocess Execution. Similar to database queries, many programming languages offer libraries or built-in methods to execute shell commands, allowing developers to run system commands directly from their code, including Python, JavaScript, Java, C/C++, Ruby, PHP, Go, Perl, and Rust.

2.2 Multi-language Security Patch

A software security patch is a set of changes between two versions of source code to address specific vulnerabilities. Listing 1.1 shows a security patch that sanitizes user input using `htmlentities()` to prevent XSS attacks.

```
diff --git a/web/edit/web/index.php b/web/edit/web/index.php
--- a/web/edit/web/index.php
+++ b/web/edit/web/index.php
+  $user_plain=htmlentities($_GET['user'])
diff --git a/.../edit_server.html b/web/templates/pages/edit_server.html
--- a/web/templates/pages/edit_server.html
+++ b/web/templates/pages/edit_server.html
- <span id="generate-csr"> / <a class="generate" target="_blank" href="/generate/ssl/?
      domain=<?=\$v_hostname?>"><?=_('Generate CSR');?></a></span>
+ <span id="generate-csr"> / <a class="generate" target="_blank" href="/generate/ssl/?
      domain=<?=htmlentities(trim($v_hostname,'"'));?>"><?=_('Generate CSR');?></a></
      span>
```

Listing 1.1. Security Patch for CVE-2022-0986 on PHP and HTML files.

We refer to security patches that contain more than one language as multi-language security patches. Language interaction can happen across multiple files or within a single file, depending on the language, environment, and tools used. In this paper, we consider both situations and use *multi-file multi-language* security patches and *single-file multi-language* security patches to refer to them.

Multi-file Multi-language Patch. Listing 1.1 shows a security patch fixing the vulnerability CVE-2022-0986 by modifying the PHP file and the HTML file. The `index.php` file escapes the HTML entities of the `user`, and `edit_server.html` escapes the HTML entities of the `v_hostname`. These two files work together to resolve the XSS vulnerability.

Single-file Multi-language Patch. Languages can interact in a single-file setup, which is referred to as embedding or interfacing between languages. Embedding is a program written in one language (like C) that uses another language (like a shell script) to handle specific tasks. For example, a C program might execute shell commands for file manipulation using `system()`. Interfacing happens when different languages work together through defined boundaries or protocols, such as using APIs, libraries, or inter-process communication, which is similar to multi-file interaction. As shown in Listing 1.2, the patch fixes the SQL injection by adding a Python sanity check to the field of the vulnerable SQL query.

```
diff --git a/gamespy/gs_database.py b/gamespy/gs_database.py
--- a/gamespy/gs_database.py
+++ b/gamespy/gs_database.py
@@ -367,12 +367,12 @@ def update_profile(self, profileid, field):
-    with Transaction(self.conn) as tx:
-        q = "UPDATE users SET \"%s\" = ? WHERE profileid = ?"
-            tx.nonquery(q % field[0], (field[1], profileid))
+        if field[0] in ["firstname", "lastname"]:
+            with Transaction(self.conn) as tx:
+                q = "UPDATE users SET \"%s\" = ? WHERE profileid = ?"
+                tx.nonquery(q % field[0], (field[1], profileid))
```

Listing 1.2. Security Patch for CVE-2020-36631 on SQL in Python file.

3 Patch Collection

To explore multi-language vulnerabilities and their fixes, we collect security patches from open-source software. According to the statistics from IEEE Spectrum [11] and the available data from MITRE [2], we focus on 16 popular programming languages, including C, C#, C++, Go, HTML, Java, JavaScript, Kotlin, Perl, PHP, Python, Ruby, Rust, Shell, SQL, and TypeScript. We summarize the language type, usage, and language interaction in Table 1.

Table 1. Language Selection.

Languages	Type	Level	Application	Language Interaction
C	Compiled	Low	Operating system	Assembly and high-level languages
C#	Compiled	High	Game	.NET languages and native code
C++	Compiled	Low	System and game	C and other system languages
Go	Compiled	High	Cloud and system	C and statically typed languages
HTML	Markup	-	Web page structure	Web languages
Java	Compiled	High	Mobile apps	JVM and native interfaces
JavaScript	Interpreted	High	Web	Web and server-side languages
Kotlin	Compiled	High	Mobile and web	Java and other JVM languages
Perl	Interpreted	High	System and web	C/C++ and web technologies
PHP	Interpreted	High	Web	HTML, JavaScript, and SQL
Python	Interpreted	High	Web and data science	C/C++ and web languages
Ruby	Interpreted	High	Web and scripting	Scripting languages
Rust	Compiled	Low	System and web	C and other system languages
Shell	Scripting	-	System and automation	Shell and command line tools
SQL	Domain-specific	-	Database management	Database languages
TypeScript	Interpreted	High	Web	JavaScript and other frameworks

Our data collection process begins by mining commits associated with CVE records indexed by MITRE [2]. First, we retrieve vulnerabilities that have been assigned CVE IDs. Next, we parse the vulnerability reports to extract patch hyperlinks and save the corresponding commits. We then download all security patches. To keep only the security patches written in the specified languages, we analyze the modified file extensions and select those that match the chosen languages, e.g., .c for C. If there is more than one type of file extension from different languages, we classify it as a multi-file, multi-language security patch. If only one programming language is involved, we include it for further analysis. For multi-file multi-language security patches, we retain all related files and manually analyze their interactions. For single-file multi-language security patches, we first review the official documentation to identify the possible languages within the file. Then, we develop a Python script to check for the presence of embedded language keywords in the host language syntax, e.g., `<?php>` as an embedded language inside a Java program. We match the presence of languages that are not listed in their documentation and flag their existence. After that, we manually check the matched single-file patches to identify the single-file multi-language security patches and analyze their interactions.

4 DATA CHARACTERIZATION

Based on the method in § 3, we collect a multi-language security patch dataset consisting of publicly indexed vulnerability fixes. We conduct a set of experimental studies to investigate the quantitative characteristics of the dataset.

4.1 Dataset Composition

We collect 14,453 security patches containing at least one of the selected languages (from the 1990s to September 24, 2024). Among them, 4,567 security patches modify more than one file type, and 1,253 of those modify at least two of the selected languages (which we consider as *multi-file multi-language* security patches), accounting for 27.44%. 9,886 security patches only modify one file type, and 1,545 of them embed other languages (which we call *single-file multi-language* security patches), making up 15.63%. In total, we collect 2,798 multi-language patches.

4.2 Language Combinations

Among 1,253 multi-file multi-language security patches, there are 111 combinations. Of these, 62 combinations include two languages, 34 combinations include three languages, 11 combinations include four languages, three combinations include five languages, and one combination includes six languages (*i.e.,* the patch for CVE-2016-10096 modifies Shell, C, HTML, PHP, SQL, and JavaScript files in the same commit). We use a Python script to count the existence of different languages. Table 2 shows the top-20 combinations. The table reveals that JavaScript, HTML, and Python stand out as the most popular languages for interacting with other languages. This popularity can be attributed to three key reasons. First, Python's versatility allows it to be applied across various domains and seamlessly integrated with other languages. Second, JavaScript's ability to function in both front-end and back-end development increases its likelihood of collaborating with other programming languages. Finally, HTML, as the backbone of most front-end websites, is inherently supported by back-end technologies, creating more opportunities for its use alongside other languages.

Table 2. The Language Combination of Multi-file Multi-language Security Patch.

Language Combination	Count	Language Combination	Count
JavaScript & PHP	178	C++ & PHP	34
C++ & Python	163	Java & JavaScript	27
JavaScript & TypeScript	89	C++ & JavaScript	27
HTML & JavaScript	64	HTML & Java	25
C & Python	55	JavaScript & Ruby	23
HTML & Python	54	C & Perl	20
C & Shell	43	HTML & PHP	18
C & C++	41	HTML & JavaScript & Python	17
JavaScript & Python	36	SQL & JavaScript & PHP	15
HTML & Ruby	35	SQL & PHP	15

Table 3. The Language Combination of Single-file Multi-language Security Patch.

Host Language	Embedded Language	Host Language	Embedded Language
JavaScript	SQL, HTML, Shell	C#	SQL, HTML, JavaScript
Kotlin	Java	C++	Shell, SQL, PHP
Perl	Shell, SQL, HTML	Go	SQL, HTML, Shell, C
Python	HTML, Shell, SQL	HTML	SQL, Java, JavaScript, PHP
Ruby	SQL, Shell, C	Java	SQL, PHP, HTML, Shell
Rust	C, Shell		

Among 1,545 single-file multi-language security patches, there are 50 combinations. We list languages that embed with one of the selected 16 programming languages in Table 3. The results show that Shell, HTML, and SQL are the most popular languages for direct use in other languages. They often serve as embedded languages because they excel in tasks that complement general-purpose languages like Python, Java, or C++. Shell scripting automates system tasks and manages files, HTML structures web pages, and SQL handles database queries. When embedded in frameworks like Django (Python) or Rails (Ruby), HTML defines web page templates, enabling dynamic content generation. Similarly, embedding SQL allows direct database interaction for efficient data management. This combination leverages the strengths of both host and embedded languages—host languages handle application logic, while embedded languages optimize specific tasks like database operations or system management.

4.3 Security Patch Complexity

Security patch complexity reveals the difficulty in creating, applying, and verifying a security patch. We assess the complexity of security patches from three perspectives: the modification complexity, the branch structural complexity, and the dependency complexity. We directly count the lines of code that have changed, identifying additions and deletions marked with '+' and '-' for modification complexity. Branch structural complexity is evaluated by counting updated conditions, loops, or branches introduced by the patch. Dependency complexity changes are measured by noting any added or removed libraries or frameworks. We develop a Python script to count the added and deleted lines, identify branch statements by matching keywords and grammar, *e.g.,* `for a in A:` in Python, and identify library usage with keywords and grammar matching, *e.g.,* `#include<A>` in C.

As shown in Figure 1, we uncover that multi-language security patches (in red) are more complex compared to single-language ones (in blue). The complexity of the multi-language security patches mainly comes from three aspects. First, the scope and nature of the vulnerability in multi-language security patches are more complex, affecting multiple components or systems and necessitating solutions that span several languages. Second, the software architecture of multi-language software is more sophisticated. In a multi-language environment, patches might need to address cross-language interactions, which can introduce additional modifications and branch structure complexity. For example, if a web

application uses JavaScript for the front-end and Python for the back-end, a security vulnerability that affects both might require coordinated patches across these technologies, considering how data is handled and passed between them. Third, languages like Python can interact with multiple languages and often have more libraries/APIs to facilitate this interaction. The availability and reliability of these libraries can affect the dependency complexity of implementing security measures.

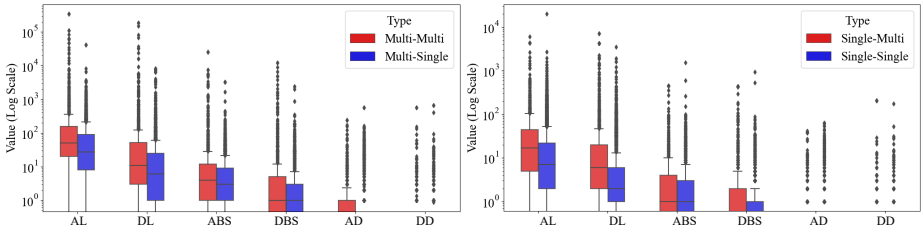


Fig. 1. Patch Complexity. AL (added lines) and DL (deleted lines) represent the modification complexity. ABS (added branch structure) and DBS (deleted branch structure) represent branch structural complexity. AD (added dependency) and DD (deleted dependency) represent dependency complexity.

4.4 Vulnerability Categories Proneness

To measure the vulnerability categories' proneness of multi-language security patches, we use CWE (Common Weakness Enumeration) [4] identifiers. We first collect the CWE identifiers associated with each patch. Next, we count and visualize the number of patches that fall into each CWE category. We get 7,605 security patches associated with 329 CWE-IDs, including 1,776 multi-language patches with 185 CWE-IDs and 5,829 single-language patches with 311 CWE-IDs. Multi-language and single-language security patches share 167 CWE-IDs. We plot the top-10 CWE distribution for multi-file and single-file patches in Figure 2.

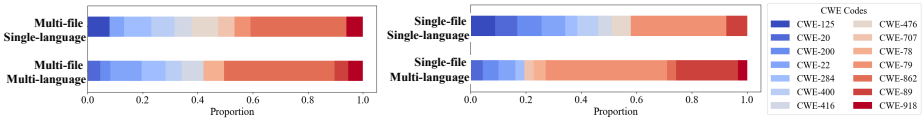


Fig. 2. CWE Distribution of Security Patches

Multi-file Single-language. The common vulnerabilities are XSS, information leakage, DoS, path traversal, heap-based buffer overflow, SSRF, and use-after-free. The distribution indicates a focus on memory management and DoS issues. **Multi-file Multi-language**. The common vulnerabilities are XSS, improper input validation, path traversal, CSRF, SQL injection, SSRF, and CSRF. The

distribution shows a mix of both client-side and server-side vulnerabilities, indicating a broad surface area due to the complex interactions between multiple file types and programming languages.

Single-file Single-language. The common vulnerabilities are XSS, out-of-bounds read, path traversal, NULL pointer dereference, use after free, and improper access control. The distribution emphasis on direct memory manipulation and access vulnerabilities, common in environments where a single language is used without the added complexity of multiple languages.

Single-file Multi-language. The common vulnerabilities are XSS, SQL injection, path traversal, OS command injection, improper neutralization, SSRF, and missing authorization. This category heavily features vulnerabilities that involve embedded languages in a single programming language.

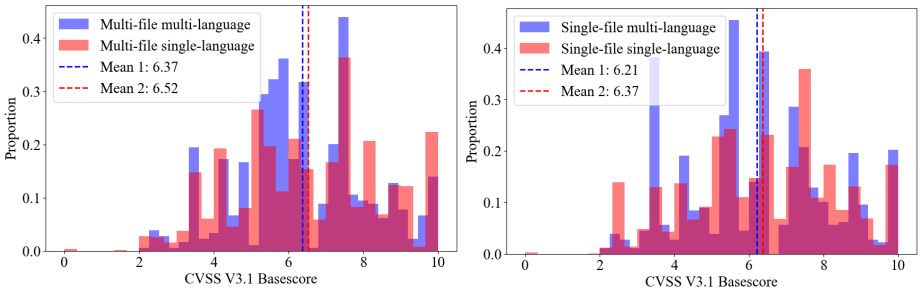
The figure reveals that both multi-language and single-language environments exhibit a higher prevalence of XSS and CSRF vulnerabilities, indicating challenges in user data handling and session management. Compared with single-language environments that tend to show more severe memory and resource handling issues like buffer overflows and use-after-free, multi-language environments are more prone to injection-related vulnerabilities, including command injection and SQL injection. Therefore, we conclude that the different combinations of programming languages lead to more concentrated types of vulnerabilities. This language combination tendency is also influenced by the type of application and the architecture used in multi-language combinations. For example, client-server architectures are common in the open-source ecosystem, and web technology stacks often include a database in the backend, shaping the types of SQL injection vulnerabilities.

4.5 Vulnerability Severity

We choose the score calculated by Common Vulnerability Scoring System 3.1 [3] to measure the severity of the vulnerability. Among the patched vulnerabilities, there are 2,200 multi-file and 3,508 single-file security patches associated with the CVSS V3.1 score, including 673 and 668 multi-language patches, respectively. We plot the histogram of the scores and compare them with single-language patched vulnerabilities. As shown in Figure 3, we can tell that the average score does not vary a lot between single-language and multiple-language patched vulnerabilities.

5 Data Analysis

We manually analyze the patches to answer two questions: (1) Why are various languages committed together? and (2) How do these languages work together to fix the vulnerability? We then showcase the fixing patterns for each language combination of multiple-file security patches and single-file security patches.



(a) Multi-file Security Patch (b) Single-file Security Patch

Fig. 3. CVSS Distribution of Security Patches

5.1 Patterns in Multiple Files

Among all multi-file multi-language security patches, we identify eight patterns. Six of these involve interoperability, which includes: (1) string sanitization to ensure input strings are properly sanitized from the user and server sides; (2) attribute validation to check and validate attributes to ensure they meet security requirements; (3) attribute or token addition, *e.g.,* the backend adds a token to support frontend validation and prevent CSRF attacks; (4) HTTP method update; (5) function and usage update; and (6) testing. Additionally, we observe two other patterns: (7) consistent changes across languages and (8) code and documentation sync that fixes the issue in one language while updating documentation in another language. The patterns cover 93% of examples, and we showcase each pattern along with the common language combinations and associated vulnerability types.

Programming Language Interoperability Patterns. Among all the multi-file multi-language security patches, the modified programming languages interact with each other with cross-language data/call flow.

String/Input Sanitization. Input sanitation is essential to prevent security vulnerabilities arising from untrusted user data. If input is not properly sanitized, attackers can exploit it through techniques such as SQL injection, XSS, and path traversal. In multi-language software systems, data shared across different languages is referred to as *global data* (*e.g.*, global variables), which is crucial for enabling interaction between different languages or system components. In web applications, global data can act as a bridge between server-generated content (*e.g.*, PHP) and client-side scripts (*e.g.*, JavaScript). In lower-level programming, global data is often used to share data between different languages or components. For instance, global data in C might be accessed directly in an assembly function. For multi-language systems where programs in different languages (*e.g.*, Python, Java, C++) communicate, environment variables can serve as global data accessible across languages. In more complex, distributed systems, databases or key-value stores (*e.g.*, Redis, Memcached) are often used to maintain a global state accessible to multiple applications or services, possibly written in different languages. As shown in Listing 1.3, `shortdesc` is global data accessible across different files in this system (PHP and JavaScript), likely as part of the page context or a globally defined JavaScript object. Both the PHP code and

JavaScript directly access `shortdesc`, so this patch secures `shortdesc` usage both in the server-rendered content and in any client-side updates. In summary, this patch adds sanitization to the `shortdesc` by ensuring it is consistently escaped in both the back-end and front-end, protecting against unintended script injection.

```
diff --git a/includes/Hooks/ActionsHooks.php b/includes/Hooks/ActionsHooks.php
--- a/includes/Hooks/ActionsHooks.php
+++ b/includes/Hooks/ActionsHooks.php
@@ -32,7 +32,7 @@ public function onInfoAction( $context, &$pageInfo ) {
            $context->msg( 'shortdescription-info-label' ),
-           $shortdesc
+           htmlspecialchars( $shortdesc )
diff --git a/modules/ext.shortDescription.js b/modules/ext.shortDescription.js
--- a/modules/ext.shortDescription.js
+++ b/modules/ext.shortDescription.js
@@ -7,7 +7,7 @@ function main() {
        tagline.classList.add( 'ext-shortdesc', 'shortdescription' );
-       tagline.innerHTML = shortdesc;
+       tagline.innerHTML = mw.html.escape( shortdesc );
```

Listing 1.3. Input Sanitization (CVE-2022-21710).

Attribute Validation. Adding a sanity check to validate security attributes, such as size and value, is a common way to fix vulnerabilities. This pattern applies across all languages and frameworks. A sanity check ensures that an attribute (or input) follows expected rules before being processed, preventing vulnerabilities such as XSS, SQL injection, and out-of-bounds access. As shown in Listing 1.4, the patch for CVE-2024-47227 is applied to enforce strict validation of the `order_name` parameter. The update ensures that only predefined values ("`name`" and "`quota`") are allowed, preventing unexpected or malicious inputs. A similar validation check is added to the template logic to prevent incorrect values from affecting UI behavior. By doing so, the patch improves code consistency, prevents invalid input from being processed, and reduces risks.

```
diff --git a/controllers/sql/user.py b/controllers/sql/user.py
--- a/controllers/sql/user.py
+++ b/controllers/sql/user.py
@@ -23,7 +23,13 @@ def GET(self, domain, cur_page=1, disabled_only=False):
+   # Currently only sorting by 'name' and 'quota' are supported.
+   if order_name not in ["name", "quota"]:
+       order_name = "name"
diff --git a/templates/default/sql/list.html b/templates/default/sql/list.html
--- a/templates/default/sql/list.html
+++ b/templates/default/sql/list.html
@@ -230,10 +230,19 @@ <h2>
         {% else %}
+           {% set url_suffix = "" %}
+           {% if order_name in ["name", "quota"] %}
+               {% set url_suffix = "?order_name=" + order_name %}
+               {% if order_by_desc %}
+                   {% set url_suffix = url_suffix + "&order_by=desc" %}
+               {% endif %}
+           {% endif %}
```

Listing 1.4. Attribute Validation (CVE-2024-47227).

Attribute or Token Addition/Update. This is where attributes or tokens have been added or updated, commonly used for web applications to enhance protection

against common attacks, particularly cross-site request forgery (CSRF). This pattern involves injecting security tokens or attributes into backend logic and frontend templates to ensure that user actions are properly authenticated. It is frequently seen in applications built with PHP, Python, Java, JavaScript, and HTML. Listing 1.5 shows the fix of CVE-2016-9456 by generating a CSRF token in the backend logic and then adding it as a hidden input field in the frontend.

```
diff --git a/lib/max/Admin/TrackerAppend.php b/lib/max/Admin/TrackerAppend.php
--- a/lib/max/Admin/TrackerAppend.php
+++ b/lib/max/Admin/TrackerAppend.php
@@ -46,6 +46,7 @@ function __construct()
+        $this->csrf_token    = phpAds_SessionGetToken();
         $this->advertiser_id = MAX_getValue('clientid', 0);
diff --git a/lib/max/themes/TrackerAppend.html b/lib/max/themes/TrackerAppend.html
--- a/lib/max/themes/TrackerAppend.html
+++ b/lib/max/themes/TrackerAppend.html
@@ -26,6 +26,7 @@
+    <input type="hidden" name="token" value="{csrf_token}" />
     <input type="hidden" name="clientid" value="{advertiser_id}" />
```

Listing 1.5. Attribute Addition (CVE-2016-9456).

HTTP Method Update. This pattern replaces GET requests with POST requests when handling sensitive data, user authentication, or modifying application state. It is widely used in web applications built with frameworks like Python (Django/Flask), JavaScript (AJAX), PHP, and others. As demonstrated in Listing 1.6, the patch addresses CVE-2016-10766 by switching from GET to POST when retrieving user information, preventing sensitive user identifiers from being exposed in the URL and instead placing them securely in the request body. The test case has been updated to confirm that the API call uses POST rather than GET in JavaScript, ensuring that user data is sent correctly in the request body. This approach mitigates risks such as data exposure in URLs, CSRF, and cache-based data leakage. Furthermore, it is a crucial security measure across various frameworks, including Django, JavaScript, PHP, and Java-based environments, helping to safeguard user data throughout different application layers.

```
diff --git a/lms/djangoapps/instructor/api.py b/lms/djangoapps/instructor/api.py
--- a/lms/djangoapps/instructor/api.py
+++ b/lms/djangoapps/instructor/api.py
@@ -874,10 +841,10 @@ def modify_access(request, course_id):
-    user = get_student_from_identifier(request.GET.get('unique_student_identifier'))
+    user = get_student_from_identifier(request.POST.get('unique_student_identifier'))
     except User.DoesNotExist:
         response_payload = {
-            'unique_student_identifier': request.GET.get('unique_student_identifier'),
+            'unique_student_identifier': request.POST.get('unique_student_identifier'),
             'userDoesNotExist': True,
diff --git a/spec/staff_debug_actions_spec.js b/spec/staff_debug_actions_spec.js
--- a/spec/staff_debug_actions_spec.js
+++ b/spec/staff_debug_actions_spec.js
@@ -91,7 +91,7 @@ define([
                     StaffDebug.reset(locationName, location);
-                    expect($.ajax.calls.mostRecent().args[0].type).toEqual('GET');
+                    expect($.ajax.calls.mostRecent().args[0].type).toEqual('POST');
```

Listing 1.6. HTTP Method From GET to POST (CVE-2016-10766).

Function and Usage Update. Vulnerabilities can be patched by modifying function calls, updating method names, or replacing deprecated functions to enforce access control and validation. It is commonly used in applications that allow dynamic function execution, API calls, or remote procedure calls, which may otherwise be exploited for unauthorized access, privilege escalation, or injection attacks. This pattern appears across different software architectures, *e.g.*, web applications (Python (Frappe/Django/Flask) + JavaScript (AJAX) + REST APIs), and embedded systems (C/C++ + Python). FFI is the most common interfacing mechanism for this pattern, which is a way to call functions, use variables, or access features written in different languages. When the function definition is updated, *e.g.*, adding new parameters, the caller needs to update the function call. Therefore, two files should be patched together to complete a fix. As shown in Listing 1.7, the backend modifies the `execute_cmd` function to ensure that method validation and whitelisting are correctly applied, preventing unauthorized execution, and the frontend JavaScript code is updated to use the new function `run_doc_method`, ensuring that frontend requests comply with the updated security rules and do not trigger insecure function calls.

```
diff --git a/frappe/handler.py b/frappe/handler.py
--- a/frappe/handler.py
+++ b/frappe/handler.py
@@ -64,8 +69,9 @@ def execute_cmd(cmd, from_async=False):
-   is_whitelisted(method)
-   is_valid_http_method(method)
+   if method != run_doc_method:
+       is_whitelisted(method)
+       is_valid_http_method(method)
@@ -75,31 +81,10 @@ def is_valid_http_method(method):
+@frappe.whitelist()
+def run_doc_method(method, docs=None, dt=None, dn=None, arg=None, args=None):
+# for backwards compatibility
+runserverobj = run_doc_method
diff --git a/frappe/public/js/frappe/request.js b/frappe/public/js/frappe/request.js
--- a/frappe/public/js/frappe/request.js
+++ b/frappe/public/js/frappe/request.js
@@ -55,7 +55,7 @@ frappe.call = function(opts) {
            $.extend(args, {
-           cmd: "runserverobj",
+           cmd: "run_doc_method",
```

Listing 1.7. Updated Function Usage (CVE-2022-23057).

Testing. When a function is written in one language but needs to be compatible with or used in applications written in another language, testing from the second language ensures that the function behaves as expected when integrated. This is often seen in APIs or libraries designed to be accessible in multiple languages (*e.g.*, a C++ library tested with Python to ensure compatibility in Python applications). Moreover, when a function is written in a low-level language like C or C++ for performance reasons, it may still be easier to test it using a higher-level language like Python, which is more flexible and readable. This approach allows developers to test the function's logic, edge cases, and integration without sacrificing the performance benefits of the low-level implementation. As shown in Listing 1.8, C++ code adds checks when processing the matrix, and Python

code tests the newly added code. This is a common pattern among Python and C/C++, C and Shell, C and Perl, and C and PHP combinations.

```
diff --git a/tensorflow/matrix_solve_op.cc b/tensorflow/matrix_solve_op.cc
+    for (int dim = 0; dim < ndims - 2; dim++) {
+      OP_REQUIRES_ASYNC(
+          context, input.dim_size(dim) == rhs.dim_size(dim),
+          errors::InvalidArgument(
+              ""All input tensors must have the same outer dimensions.""), done);
+    }
diff --git a/tensorflow/matrix_solve_op_test.py b/tensorflow/matrix_solve_op_test.py
+    matrix = np.random.normal(size=(2, 6, 2, 2))
+    rhs = np.random.normal(size=(2, 3, 2, 2))
+    with self.assertRaises((ValueError, errors_impl.InvalidArgumentError)):
+      self.evaluate(linalg_ops.matrix_solve(matrix, rhs))
```

Listing 1.8. Testing.

Other Patterns. We also discover two more patterns that involve more than one type of file being updated while they do not interact with each other.

Consistent Changes Across Languages. Developers may migrate the software from one language to another to adapt the new features, *e.g.*, Linux from C to Rust. In the middle of the migration, to ensure functionality and compatibility, the code repository may contain similar code in the two programming languages. Besides, to be compatible with different platforms or meet different expectations, a piece of software may be developed in different languages, *e.g.*, Mozilla Firefox, C++/Rust for Desktop, Java/Kotlin for Android. In such scenarios, although multiple different files have been modified, the semantics of the modified files stay the same to guarantee all versions are patched. Listing 1.9 shows C and C++ perform the same change to avoid buffer overflow.

```
diff --git a/dcraw/dcraw.c b/dcraw/dcraw.c
-  if(tiff_bps <= 8)
-      gamma_curve(1.0/imgdata.params.coolscan_nef_gamma,0.,1,255);
+  if(!image)
+      throw LIBRAW_EXCEPTION_IO_CORRUPT;
+  int bypp = tiff_bps <= 8 ? 1 : 2;
diff --git a/internal/dcraw_common.cpp b/internal/dcraw_common.cpp
-  if(tiff_bps <= 8)
-      gamma_curve(1.0/imgdata.params.coolscan_nef_gamma,0.,1,255);
+  if(!image)
+      throw LIBRAW_EXCEPTION_IO_CORRUPT;
+  int bypp = tiff_bps <= 8 ? 1 : 2;
```

Listing 1.9. Same Change (CVE-2018-5812).

Code and Documentation. Another common pattern in non-interactive multi-language security patches is that one language is responsible for fixing the vulnerability, while the other updates the version information or related documentation. As shown in Listing 1.10, the vulnerability is addressed through Java code changes, whereas the HTML update is limited to modifying the bug description. In this pattern, the language responsible for fixing the vulnerability can vary, but HTML is commonly used for documentation updates.

```
diff --git a/changelog.html b/changelog.html
--- a/changelog.html
+++ b/changelog.html
@@ -61,6 +61,10 @@
+  <li class=bug>
+    may refuse set <a href="https://www.owasp.org/index.php/SecureFlag">secure</a>.
+    Deal with it gracefully.
+    (<a href="https://issues.jenkins-ci.org/browse/JENKINS-25019">issue 25019</a>)
diff --git a/core/src/main/java/WebAppMain.java b/core/src/main/java/WebAppMain.java
--- a/core/src/main/java/WebAppMain.java
+++ b/core/src/main/java/WebAppMain.java
@@ -56,6 +56,7 @@
+       markCookieAsHttpOnly(context);
+       private void markCookieAsHttpOnly(ServletContext context) {
+        try {
+            LOGGER.log(Level.WARNING, "Failed to set HTTP-only cookie flag", e);
+        }
+    }
```

Listing 1.10. Documentaion Update (CVE-2014-9635).

5.2 Patterns in Single File

Among security patches, when multiple languages interact within a single file, embedding is a popular way to handle specific tasks. As shown in Table 3, SQL, Shell, and HTML are the three most popular programming languages that are embedded.

SQL Query. Many programs, *e.g.,* mobile apps and web apps, interact with databases, so embedding SQL allows developers to combine SQL's data-handling strengths with the computational capabilities of general-purpose languages like Python, Java, or C. The interaction workflow spans four phases: (1) Connection: Establish a connection to the database using credentials and connection strings; (2) Query Execution: SQL commands are sent as strings or method calls from the language to the database; (3) Result Handling: The database processes the query and sends results back, which are parsed and used in the program; (4) Error Handling: SQL errors (*e.g.*, syntax issues, connection failures) are handled by the host language's mechanisms. The fix patterns are associated with each workflow phase. Listing 1.11 shows a patch that modifies the `machine.py` file to improve how SQL queries are handled. Specifically, it replaces string interpolation with parameterized queries to enhance security and maintainability. The updated code uses a parameterized query and passes the `cc_number` as a parameter. This ensures that the database driver handles escaping, preventing SQL injection.

Subprocess Execution. Embedding shell scripts allows developers to combine the logic and features of high-level programming languages with the system-level capabilities of shell commands. Most programming languages provide a way to execute shell commands or scripts using system calls. These system calls create a child process to run the shell script and capture its output. As shown in Listing 1.12, this patch modifies the `apkleaks.py` script to improve the handling of command-line arguments when decompiling APK files. It addresses potential issues like command injection and ensures proper escaping of arguments passed to the `os.system` call. Besides, subprocess libraries offer more control over shell execution, such as capturing output, handling errors, and managing inputs. As shown in Listing 1.13, this patch modifies the `gluon/messageboxhandler.py` file to replace the use of `os.system` with `subprocess.run` for sending system notifications

```
diff --git a/machine.py b/machine.py
--- a/machine.py
+++ b/machine.py
@@ -151,8 +151,7 @@ def is_card_pin_at_session(request):
 def get_card(request, cc_number):
-    q = "select * from cards where cc_number = '%s'" % cc_number.replace('-', '')
-    row = request.db.execute(q).fetchone()
+    row = request.db.execute("select * from cards where cc_number = ?", (cc_number.
     replace('-', ''),)).fetchone()
@@ -172,7 +171,7 @@ def update_failed_attempts(request, failed_attempts):
 def block_card(request):
     card = request.session['card']
-    request.db.execute("update cards set status='blocked' where id=%s" % card['id'])
+    request.db.execute("update cards set status='blocked' where id=?", (card['id'],))
     request.db.commit()
```

Listing 1.11. SQL Query (CVE-2015-10069).

using the `notify-send` command. From these two examples, we can tell that the shell script can be constructed as a string, or the tokens that form the shell statement will be used as parameters for library methods.

```
diff --git a/apkleaks/apkleaks.py b/apkleaks/apkleaks.py
--- a/apkleaks/apkleaks.py
+++ b/apkleaks/apkleaks.py
@@ -2,6 +2,7 @@
+ from pipes import quote
@@ -84,8 +85,9 @@ def decompile(self):
-       dec = "%s %s -d %s --deobf" % (self.jadx, dex, self.tempdir)
-       os.system(dec)
+       args = [self.jadx, dex, "-d", self.tempdir, "--deobf"]
+       comm = "%s" % (" ".join(quote(arg) for arg in args))
+       os.system(comm)
```

Listing 1.12. Subprocess (CVE-2021-21386).

```
diff --git a/gluon/messageboxhandler.py b/gluon/messageboxhandler.py
--- a/gluon/messageboxhandler.py
+++ b/gluon/messageboxhandler.py
@@ -1,6 +1,6 @@
-import os
+import subprocess
@@ -36,4 +36,4 @@ def __init__(self):
     def emit(self, record):
         if tkinter:
             msg = self.format(record)
-            os.system("notify-send '%s'" % msg)
+            subprocess.run(["notify-send", msg], check=False, timeout=2)
```

Listing 1.13. Subprocess (CVE-2023-45158).

Embedded in HTML. Embedding HTML introduces security risks such as XSS, HTML injection, Clickjacking, and CSRF. These vulnerabilities arise when user input is not properly escaped, sanitized, or validated, allowing attackers to inject malicious scripts, manipulate HTML structures, or execute unauthorized actions. To mitigate these risks, developers often use escaping functions, enforce Content Security Policies (CSP) to restrict inline scripts, sanitize user input using libraries like DOMPurify (JavaScript) or Bleach (Python), and implement CSRF tokens in forms. Additionally, avoiding direct execution of embedded HTML and using templating engines ensures structured and secure rendering, reducing the likelihood of security breaches while maintaining flexibility in content presenta-

tion. As shown in Listing 1.14, the patch shows HTML embedded in PHP using Laravel Blade templates. The code includes HTML elements (`a <form> tag`) while also using Laravel's Blade directives like `@csrf` and `route('admin.logout')` , which generate PHP code dynamically. This update ensures that logout requests are sent as POST requests with a CSRF token, preventing CSRF attacks where attackers might trick users into logging out unknowingly.

```
diff --git a/views/layouts/main.blade.php b/views/layouts/main.blade.php
--- a/views/layouts/main.blade.php
+++ b/views/layouts/main.blade.php
@@ -71,6 +71,11 @@
    @include('twill::partials.footer')   </section>  </div>
+   <form class="visually-hidden" method="POST" action="{{ route('admin.logout') }}"
      data-logout-form> @csrf </form>
```

Listing 1.14. Embedded HTML (CVE-2021-3932).

6 Discussion

We outline various usage scenarios for our work. Additionally, we discuss its limitations and propose directions for future work.

6.1 Usage Scenarios

We release the dataset, including patch patterns, and will expand it with multi-language non-security patches. This dataset will serve as a benchmark, training data for multi-language program analysis tools, and automatic program repairs.
Benchmark. To the best of our knowledge, the dataset we released is the largest multi-language security patch dataset, encompassing the most languages. It serves as a robust benchmark for evaluating the learning or transfer capabilities of target techniques, as it includes interactions across 16 programming languages and 185 vulnerability types.
Multi-language Program Analysis Tool. Previous program analysis tools, often focused on single languages, may miss vulnerabilities in embedded or interacting languages. Our dataset addresses this gap by offering language combinations and test cases to evaluate such tools. By analyzing existing vulnerability statistics, researchers can prioritize studying language pairs more prone to vulnerabilities, *e.g.*, JavaScript and PHP, to enhance tool effectiveness.
Auto Program Repair. The primary goal of releasing the multi-language security patch dataset is to advance automatic program repair. The dataset includes both vulnerable (pre-patch) and fixed (post-patch) code, enabling the detection of vulnerabilities and guiding their remediation. The pre-patch versions can be used to design static or dynamic analysis tools or train deep learning models to identify vulnerability patterns across languages. Meanwhile, the post-patch versions provide remediation examples, which can train language models to transform vulnerable code into secure code. Notably, our dataset covers single-file multi-language security patches, meaning it can address cases where embedded languages appear as strings within a host language.

6.2 Limitation and Future Work

Our dataset is sourced from MITRE, but the analysis and indexing of CVEs by CVE Numbering Authorities (CNAs) [1] may introduce biases, as the 421 diverse CNAs do not catalog every reported vulnerability. As a result, our dataset does not include all multi-language security patches found in open-source repositories. Additionally, not all vulnerability records include fixes, nor are all fixes publicly released, so our dataset lacks patches for these cases. Furthermore, some vulnerability reports omit CWE and CVSS scores, potentially limiting the statistical analysis of vulnerability types.

Due to differing advisory policies, some vendors release security patches without public disclosure. Research [31] shows that 6-10% of GitHub commits are security patches, prompting our future focus on identifying and analyzing silent multi-language patches. Furthermore, as our MITRE-based dataset lacks certain interaction types, we plan to generate synthetic examples to incorporate unseen language combinations and interaction patterns.

7 Related Work

Patch Dataset. Security commits offer valuable insights into existing vulnerabilities and their corresponding fixes, making them essential for creating datasets used in security commit detection and automated program repair. However, all [29,14,31,28,9,13,34] of the existing datasets are limited to patches written in one programming language. Besides, most of the existing datasets are limited to specific projects [29,35] or contain only a small number of security commits linked to CVEs [14,9]. While some researchers include commits indexed by NVD as well as silent fixes [31,28,13,30], their focus has been primarily on C/C++, Java, and Python, overlooking widespread multi-language repositories. Although Li et al. [17] have released a commit dataset of multilingual projects, the samples in this dataset are not in multi-languages nor indexed by the MITRE, and the interfacing mechanisms included are limited. Additionally, the quality of the data cannot be assured, as only less than 5% of the samples were manually verified. Furthermore, their method of using keyword matching to identify vulnerable commits does not guarantee the accuracy of the identified vulnerabilities.

Multi-language Software and Patch Analysis. Building software using multiple programming languages has been a normal practice for a long time. Researchers have measured the language selection [25,20,19], challenges [26], and bad practices [7]. However, it remains uncertain if multilingual code construction has significant security implications or leads to real security consequences. Li et al. [17,18] found statistically significant associations between the proneness of multilingual code to vulnerabilities, which is correlated with the language interfacing mechanism, not that of individual languages. They also introduce the first taxonomy of language interfacing. However, their analysis is limited to the project level. In multi-language software, not all commits involve multiple languages. As a result, their conclusions apply to multi-language software as a whole, not to multi-language patches. The former cannot fully represent the latter. Even within the same repository, interaction mechanisms can vary.

For the same language combination, both vulnerability patterns and interface mechanisms may differ. Moreover, they define only four types of interfacing and overlook interaction details that occur within single-file multi-language cases.

Motivated by the security impact of the multi-language trend, researchers propose approaches to analyze the multi-language patches. However, these approaches either focus on limited language combinations or specific application types. Buro et al. [10] show formal properties of interest of multi-language abstractions. Li et al. [33] and Yang et al. [21] propose to locate and detect multilingual bugs in Python and C/C++ repositories, *i.e.*, TensorFlow and NumPy. Lei et al. [16] propose intermediate representations to detect vulnerabilities across C/++ and Java. Figueiredo et al. [15] detects multi-language vulnerabilities for web applications. Negrini [27] introduces a generic framework for multi-language analysis for Go, Java, and C++ in smart contract development. Yang et al. [32] characterize Python-C and Java-C patches into 5 aspects, including their symptoms, locations, manifestations, root causes, and fixes.

The scarcity of data on language combinations, insufficient explanations of language interoperability, and the lack of summarized fixing patterns have motivated us to create a comprehensive dataset of multi-language security patches from diverse projects. Furthermore, we seek to extract fixing patterns to encourage advancements in automated program repair.

8 Conclusion

In this paper, we investigate why commits in multi-language repositories involve updates across multiple languages and how these languages collaborate to address vulnerabilities. We analyze 2,798 multi-language security patches indexed by MITRE, comprising 1,253 multi-file and 1,545 single-file patches. Our findings reveal that multi-language patches are more complex than single-language ones, with greater modifications, intricate branch structures, and higher dependency complexity. We also find that multi-language security patches are more likely to involve injection-related and web-related vulnerabilities, mainly due to the general system design and choice of languages used in multi-language software systems. We summarize 11 fixing patterns and observe their correlation with interfacing mechanisms. To promote further research, we release our dataset, including fixing patterns, to support the development of automated multi-language program repairs.

Acknowledgments

We appreciate the helpful comments from our shepherd and the reviewers. This work was partially supported by the US Office of Naval Research grant N00014-23-1-2122.

References

1. CNAs (2024), https://www.cve.org/ProgramOrganization/CNAs
2. CVE. (2024), https://cve.mitre.org/index.html
3. CVSS_v3 (2024), https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator
4. CWE (2024), https://cwe.mitre.org/
5. Github commits language count (2025), âĂŒhttps://api.github.com/search/commits?q=language:{languagecombination}
6. Github repositories language count (2025), âĂŒhttps://api.github.com/search/repositories?q=language:{languagecombination}
7. Abidi, M., Grichi, M., Khomh, F., Guéhéneuc, Y.G.: Code smells for multi-language systems. In: Proceedings of the 24th European conference on pattern languages of programs. pp. 1–13 (2019)
8. Bandara, V., Rathnayake, T., Weerasekara, N., Elvitigala, C., Thilakarathna, K., Wijesekera, P., Keppitiyagama, C.: Fix that fix commit: A real-world remediation analysis of javascript projects. In: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE (2020)
9. Bhandari, G., Naseer, A., Moonen, L.: CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 30–39 (2021)
10. Buro, S., Crole, R., Mastroeni, I.: On multi-language abstraction: Towards a static analysis of multi-language programs. Formal Methods in System Design (2023)
11. Cass, S.: The top programming languages 2024 (Aug 2024), https://spectrum.ieee.org/top-programming-languages-2024
12. Chakraborty, P., Alfadel, M., Nagappan, M.: Blaze: Cross-language and cross-project bug localization via dynamic chunking and hard example learning. arXiv preprint arXiv:2407.17631 (2024)
13. Chen, Y., Ding, Z., Chen, X., Wagner, D.: DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. arXiv preprint arXiv:2304.00409 (2023)
14. Fan, J., Li, Y., Wang, S., Nguyen, T.N.: AC/C++ code vulnerability dataset with code changes and CVE summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. pp. 508–512 (2020)
15. Figueiredo, A., Lide, T., Correia, M.: Multi-language web vulnerability detection. In: 2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). pp. 153–154. IEEE (2020)
16. Lei, T., Xue, J., Wang, Y., Liu, Z.: Irc-clvul: Cross-programming-language vulnerability detection with intermediate representations and combined features. Electronics **12**(14), 3067 (2023)
17. Li, W., Li, L., Cai, H.: On the vulnerability proneness of multilingual code. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 847–859 (2022)
18. Li, W., Li, L., Cai, H.: Polyfax: a toolkit for characterizing multi-language software. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1662–1666 (2022)
19. Li, W., Marino, A., Yang, H., Meng, N., Li, L., Cai, H.: How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software. ACM Transactions on Software Engineering and Methodology **33**(3), 1–46 (2024)

20. Li, W., Meng, N., Li, L., Cai, H.: Understanding language selection in multi-language software projects on github. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 256–257. IEEE (2021)
21. Li, W., Ming, J., Luo, X., Cai, H.: {PolyCruise}: A {Cross-Language} dynamic information flow analysis. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2513–2530 (2022)
22. Li, Z., Ji, J., Liang, P., Mo, R., Liu, H.: An exploratory study on just-in-time multi-programming-language bug prediction. Information and Software Technology **175**, 107524 (2024)
23. Li, Z., Wang, W., Wang, S., Liang, P., Mo, R.: Understanding resolution of multi-language bugs: An empirical study on apache projects. In: 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). pp. 1–11. IEEE (2023)
24. Malone, T.: Interoperability in programming languages. Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal **1**(2), 3 (2014)
25. Mayer, P., Bauer, A.: An empirical analysis of the utilization of multiple programming languages in open source projects. In: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering. pp. 1–10 (2015)
26. Mushtaq, Z., Rasool, G.: Multilingual source code analysis: State of the art and challenges. In: 2015 International Conference on Open Source Systems & Technologies (ICOSST). pp. 170–175. IEEE (2015)
27. Negrini, L.: A generic framework for multilanguage analysis (2023)
28. Nikitopoulos, G., Dritsa, K., Louridas, P., Mitropoulos, D.: CrossVul: a cross-language vulnerability dataset with commit data. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1565–1569 (2021)
29. Ponta, S.E., Plate, H., Sabetta, A., Bezzi, M., Dangremont, C.: A manually-curated dataset of fixes to vulnerabilities of open-source software. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE (2019)
30. Sun, S., Wang, S., Wang, X., Xing, Y., Zhang, E., Sun, K.: Exploring security commits in python. In: 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 171–181. IEEE (2023)
31. Wang, X., Wang, S., Feng, P., Sun, K., Jajodia, S.: Patchdb: A large-scale security patch dataset. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 149–160. IEEE (2021)
32. YANG, H., CAI, H.: Dissecting real-world cross-language bugs (2025)
33. Yang, H., Nong, Y., Zhang, T., Luo, X., Cai, H.: Learning to detect and localize multilingual bugs. Proceedings of the ACM on Software Engineering (FSE) (2024)
34. Zheng, Y., Pujar, S., Lewis, B., Buratti, L., Epstein, E., Yang, B., Laredo, J., Morari, A., Su, Z.: D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) (2021)
35. Zhou, Y., Siow, J.K., Wang, C., Liu, S., Liu, Y.: SPI: Automated identification of security patches via commits. ACM Transactions on Software Engineering and Methodology (TOSEM) **31**(1), 1–27 (2021)