ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

DISPATCH: Unraveling Security Patches from Entangled Code Changes

Shiyu Sun*
George Mason University

Yunlong Xing*
George Mason University

Xinda Wang
University of Texas at Dallas

Shu Wang
Palo Alto Networks, Inc.

Qi Li
Tsinghua University

Kun Sun
George Mason University

Abstract

Security patches are crucial for preserving the integrity, confidentiality, and availability of computing resources. However, their deployment can be significantly postponed when intertwined with non-security patches. Existing code change decomposition methods are primarily designed for code review, focusing on connecting related parts. However, they often include irrelevant statements in a bloated security patch, complicating security patch detection, verification, and deployment. In this paper, we develop a patch decomposition system named DISPATCH for unraveling individual security patches from entangled code changes. We first introduce a graph representation named PatchGraph to capture the fine-grained code modifications by retaining changed syntax and dependency. Next, we perform a two-stage patch dependency analysis to group the changed statements addressing the same vulnerability into individual security patches. The first stage focuses on the statement level, where boundaries are defined to exclude unrelated statements. The second stage analyzes the unvisited dependencies, ensuring the patch's applicability by maintaining syntactic correctness and function completeness. In the evaluation across four popular software repositories (i.e., OpenSSL, Linux Kernel, ImageMagick, and Nginx), DISPATCH can unravel individual security patches from entangled ones with over 91.9% recall, outperforming existing methods by at least 20% in accuracy.

1 Introduction

A software patch is a piece of code changes to update or fix issues in a computer program or system. These issues include security vulnerabilities, implementation bugs, performance enhancements, and compatibility improvements. Security patches are the critical updates addressing software vulnerabilities to maintain the integrity, confidentiality, and availability of computing resources [15]. Prompt adoption of these patches can significantly bolster overall security, safeguarding against data breaches and cyber-attacks. It is best practice to keep each patch focused on a specific purpose, thereby facilitating its understanding, testing, and deployment.

However, in practice, security patches are typically entangled with other patches. Based on the research across eight popular repositories [40], 11%–39% of security patches are intricately entangled with non-fixing changes. Moreover, we randomly sampled 500 security patches from NVD [7], 34.6% (173/500) of them (e.g., CVE-2019-13223 [12] and CVE-2019-11338 [13]) are entangled with at least one security patch or non-security patch. Directly applying these entangled patches may introduce unforeseen bugs or regressions, increasing the maintenance overhead due to the heightened complexity of testing and verification. Moreover, these entangled patches pose significant challenges for organizations by hindering their ability to prioritize, verify the efficacy, and ensure the compatibility of security updates. Consequently, this hampers the prompt securing of systems and prolongs exposure to known vulnerabilities. Therefore, it is vital to isolate and comprehend the impact of each security patch from the entangled patches.

Several efforts have been made to unravel entangled patches. With various types of program representations being extracted to cover syntactic structure, control flow, and data dependencies in the code changes, different clustering approaches, including heuristic-based solutions [20, 25, 27, 29, 30, 48, 50], slicing-based solutions [25, 39, 56], and graph clustering-based solutions [18, 22, 24, 32, 41, 44, 47, 51] are performed to cluster related statements into separate groups. However, since the main objective of existing solutions is to facilitate code review, they face two challenges when unraveling security patches from entangled ones. First, it is prone to include irrelevant statements in a security patch. When a security patch has data dependencies with other non-security patches, existing solutions often mistakenly treat such intertwined updates as a single patch. It will unnecessarily increase the efforts to test the bloated security patch. Second, it cannot guarantee the completeness of an individual security patch,

*These authors contributed equally to this work.

potentially leaving parts of the patch in another cluster. This incompleteness can result in two problems: (1) the security patch may fail to compile, and (2) even if it compiles, it may not fully address the target vulnerability. Therefore, it remains a challenge to correctly and accurately unravel security patches from the entangled ones.

In this paper, we propose a patch decomposition system named DISPATCH to unravel individual security patches from entangled code changes. Our work is motivated by the observation that existing approaches [22, 32, 47, 51, 56] only preserve the deleted/added syntax in the patch representation while neglecting the updated statements, which could provide more essential information. We consider the updated ones by identifying what has been changed and how those changes are made (*e.g.*, expanding the boundary of if conditions), and grouping changes with the same purpose. We first develop a graph-based patch representation called PatchGraph to capture the fine-grain syntax and dependency modification, *i.e.*, diff behaviors, serving as the prerequisite for individual patch generation. Then, we conduct a two-stage patch dependency analysis to determine the scope of individual patches and separate security patches from the mingled ones. Different from traditional dependency analysis [51, 56] that relies on control/data dependency and call relations, we focus on the diff behaviors, changes over control/data dependency, and the call relation modifications. Firstly, we conduct a statement dependency analysis to cluster modified statements into the patch segments without irrelevant changes. Secondly, we perform segment dependency analysis to generate individual security patches by utilizing the relationships among patch segments.

Given an entangled patch, we develop PatchGraph to represent the fine-grained changes (*e.g.*, deleted unused variable, new function call, or update return status), which are called *diff behavior*. We first distinguish the changed statements into three modification types, *i.e.*, deleted/added/updated, and then identify the deleted/added identifiers (*e.g.*, variable, function, and return status) from the deleted or added statements and compare the fine-grained tokens in updated statements to retain the modified components and discard the unchanged ones. By doing so, we can successfully constrain the diff behavior to the modified syntax and corresponding changed data dependency. Finally, we construct PatchGraph by overlaying the diff behaviors on classic program dependencies (*i.e.*, CDG, DDG, and call graph).

To ensure that only modification-dependent statements are grouped, we conduct a patch statement-level dependency analysis to refine the scope of each patch segment. We first perform a diff behavior based slicing to collect the statements connected by the same type of diff behavior with the same entity (*e.g.*, variable, function, return value). Then, we introduce the pattern-guided slicing to gather the statements resolving the same concern. These patterns[1] are extracted from the public patch dataset [7, 53] across more than 300 popular GitHub repositories and are composed of anchor nodes to start the slicing, slicing direction with termination criteria to stop the slicing, and security attribute indicators to suggest the security patches segments. Thus, we can keep only the relevant statements to one security patch.

[1]These patterns are summarized manually by three students, each with more than three years of experience in software security.

After obtaining the patch segments, we conduct a segment-level dependency analysis that further merges dependent patch segments into individual patches to ensure their grammar correctness and functionality completeness. It contains three phases, namely, determining whether to keep the intraprocedural dependencies to avoid using the undefined variables, resolving the interprocedural call dependencies to seek the function declaration and implementation, and grouping similar segments to ensure one concern can be resolved entirely. Finally, when an individual patch contains both a security patch segment and a non-security patch segment, we follow pre-defined criteria to decide if it should be considered as an individual security patch. For instance, if a security patch segment is to fix the vulnerability introduced by a non-security patch segment (*e.g.*, adding a new feature), we treat it as an individual non-security patch.

We implement a prototype of DISPATCH and evaluate its decomposition correctness with four popular repositories, *i.e.*, OpenSSL, Linux Kernel, ImageMagick, and Nginx. Their repository sizes span from 84.5MB to 6.71GB. DISPATCH can successfully unravel 91.9% of individual security patches from entangled patches. We also compare with state-of-the-art (SOTA) approaches, including SPatch [36] (a slicing-based approach), GPT-4 [16] (an LLM-based approach), and UTANGO [32] (a GNN-based approach). The experimental results show that DISPATCH outperforms all the existing methods, with an accuracy improvement of at least 20%, and the results also reveal that the proposed patterns can be generalized to different types of applications.

In summary, we make the following contributions:

- We design a patch untangling system to unravel individual security patches from entangled ones.
- We propose a new graph-based patch representation to capture the modified syntax and dependencies between pre-patch and post-patch code.
- We develop a constrained slicing method to refine the scope of individual patches.
- We implement a prototype of DISPATCH and the experimental results show its effectiveness in disentangling security patches.

2 Preliminaries

2.1 Entangled Patches

A *patch* is a software component that, when installed, directly modifies files or device settings related to a different software component [9]. A patch can be a GitHub commit or a set of code differences between two versions of software generated by the `diff` command in Unix-like operating systems. Ideally, a patch should focus on resolving a single concern, *e.g.*, a bug fix, feature update, or maintenance. However, in the real world, a patch may involve multiple activities to address different concerns [40], and we call it as an *entangled patch*. Listing 1 shows an entangled patch that is composed of two patches, namely, a non-security patch (Line 4, Lines 7-8, and Line 13) to add a new user profile attribute and a security patch (Lines 9-10) to fix a Null Pointer Dereference bug.

```
1     typedef struct{
2         char* name;
3         int age;
4  +      char* email;
5     } UserProfile;
6  @@ void processUserProfile(UserProfile* userProfile,
7  -    char* newName, int newAge){
8  +    char* newName, int newAge, char* newEmail){
9  +    if (userProfile == NULL)
10 +        return;
11      userProfile->name = newName;
12      userProfile->age = newAge;
13 +    userProfile->email = newEmail;
14      print("Updated user profile\n");
```

Listing 1: Security patch entangled with non-security patch.

However, due to the update of the function signature in the non-security patch, users may deny the entire patch for version compatibility reasons, hindering the propagation of the security patch that can be applied separately from the non-security patch. We analyze 339 closed pull requests in OpenSSL, consisting of 200 randomly selected unmerged and all 139 merged requests. Of these, 221 involve C code, with 90 (40.7%) identified as entangled. Among the 90 entangled patches, 63 (70%) pull requests were not merged, while only 27 (30%) were merged. This highlights the challenges in adopting entangled patches. Detailed statistics from the case study are provided in Appendix A.

To assist users in understanding the composition of the entangled patches and determining what part of a patch to apply, it is critical to unravel individual patches from entangled patches. In this paper, we refer *individual patches* to the patches that resolve one single concern and can be separately applied to the target software.

2.2 Program Dependencies

Program dependencies play a crucial role in understanding the code syntax, execution logic, and code semantics. There are three primary types of program dependencies: data dependency, control dependency, and call dependency. Data dependency in a program occurs when a program statement refers to the data of a preceding statement, and control dependency represents a situation in which the execution of a program statement depends on the outcome of a preceding control statement. Call dependency indicates relationships between functions or procedures. Program dependencies focus on grasping the information in a single version of code. To better understand the patch code changes that involve two versions of code, it is critical to extend program dependency information to cover the differences between the two versions.

In this paper, we introduce a new type of program dependency, named *diff behavior*, to record the deleted/added/updated behaviors in code changes. A detailed description will be given in § 5. We integrate the diff behaviors with traditional control dependency graph (CDG), data dependency graph (DDG), and call graph (CG) to provide details on modified syntax (*e.g.*, new variable declarations and altered variable definitions) and updated dependencies (*e.g.*, new data linkages, altered conditional control dependencies, revised variable assignments, and updated function utilization).

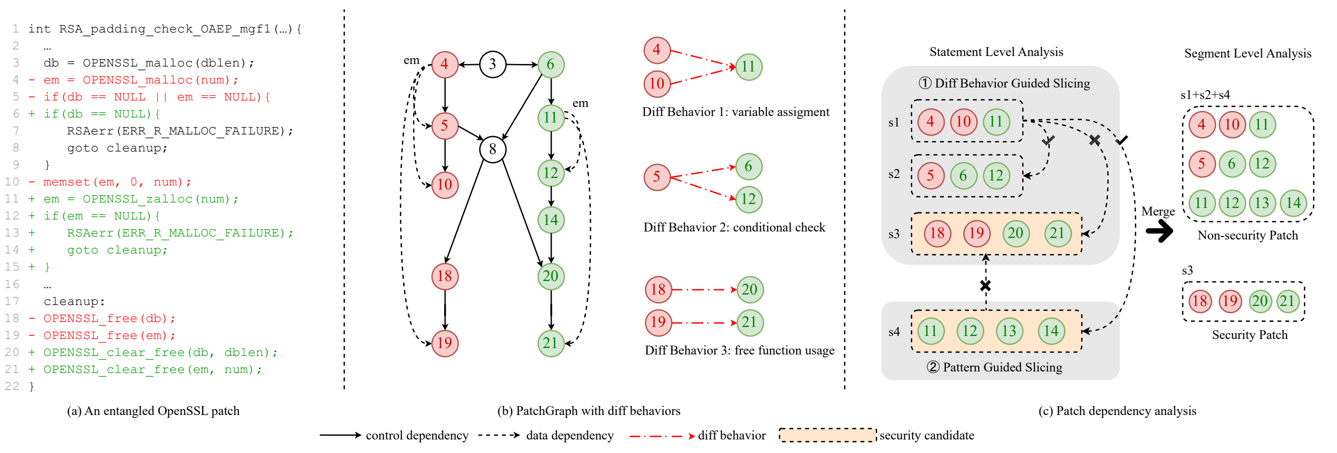
2.3 Program Slicing

Program slicing is a technique in program analysis to simplify the understanding and analysis of large and complex programs by focusing only on the related portions [54] (*i.e.*, slices). Slicing starts from a variable of a program point and traverses the program along dependency edges (*e.g.*, data dependency, control dependency, or both) until the specified criteria (*e.g.*, the number of hops, and the end) are met. We develop a constrained program slicing technique that specifies the beginning point and criteria for retaining relevant statements and eliminating irrelevant ones. The details can be found in § 6.2.

3 A Motivating Example

We use an example in Figure 1 to demonstrate how DISPATCH decomposes individual security patches entangled with non-security ones. Unlike existing methods [36, 44] that coarsely analyze patches at the statement level, we adopt a fine-grained approach by analyzing changes at the dependency level and token level by categorizing them as deleted, added, or updated. Furthermore, unlike decomposition methods designed for code review, which aim to reveal the dependencies and group statements as long as they have dependencies, we explicitly define boundaries for individual patches that can be applied back without breaking the original functionality. DISPATCH consists of two steps, namely, constructing a new graph-based patch representation called PatchGraph and conducting dependency analysis to generate individual patches.

Understanding an Entangled Patch. Figure 1(a) shows an entangled patch that is composed of a security patch (Lines 18-21) and a non-security patch (Lines 4-6, 10-15) [43]. From Line 3 to Line 5, the pre-patch allocates two memory spaces and then checks failures for both memory allocations. However, if the first allocation fails, there is no need to conduct the second checking. Therefore, the post-patch optimizes the memory allocation by allocating `em` (Line 11) after the successful allocation of `db` (Line 6). Instead of allocating `em` using `OPENSSL_malloc` (Line 4) and setting it to zero with `memset`



```
1  int RSA_padding_check_OAEP_mgf1(…){
2    …
3    db = OPENSSL_malloc(dblen);
4  - em = OPENSSL_malloc(num);
5  - if(db == NULL || em == NULL){
6  + if(db == NULL){
7       RSAerr(ERR_R_MALLOC_FAILURE);
8       goto cleanup;
9    }
10 - memset(em, 0, num);
11 + em = OPENSSL_zalloc(num);
12 + if(em == NULL){
13 +    RSAerr(ERR_R_MALLOC_FAILURE);
14 +    goto cleanup;
15 + }
16   …
17   cleanup:
18 - OPENSSL_free(db);
19 - OPENSSL_free(em);
20 + OPENSSL_clear_free(db, dblen);
21 + OPENSSL_clear_free(em, num);
22 }
```

(a) An entangled OpenSSL patch
em
em
Diff Behavior 1: variable assigment
Diff Behavior 2: conditional check
Diff Behavior 3: free function usage
(b) PatchGraph with diff behaviors
Statement Level Analysis
① Diff Behavior Guided Slicing
s1
s2
s3
s4
② Pattern Guided Slicing
Segment Level Analysis
s1+s2+s4
Non-security Patch
Merge
s3
Security Patch
(c) Patch dependency analysis
control dependency
data dependency
diff behavior
security candidate

Figure 1: A motivating example of separating a security patch from a non-security patch in an entangled patch.

(Line 10) in the pre-patch, the post-patch accomplishes these two steps by `OPENSSL_zalloc` (Line 11). The above changes belong to a non-security patch since it only optimizes the program structure without fixing any vulnerability. From Line 18 to Line 21, the function `OPENSSL_free` is replaced with `OPENSSL_clear_free` that clears the memory containing sensitive information before the free operation, indicating a security patch to prevent memory leakage.

Due to the data dependencies on variable `em` (i.e., lines 4 and 19, lines 11 and 21), existing data dependency-based patch decomposition solutions [22, 32, 47, 51, 56] consider all changed code in Figure 1(a) as a single patch. However, the value of em in Line 21 is independent of Line 11 since no matter whether Line 11 has been applied, the value of em stays unchanged. Therefore, we should consider them as two individual patches.

Representing a Patch with PatchGraph. To capture what has been modified and how the modifications happen, we introduce diff behavior and integrate the diff behavior with traditional program dependency graphs (*i.e.*, DDG, and CDG) as PatchGraph. As shown in Figure 1(b), the left part is the CDG and DDG of the patch (simplified due to space limitation). On the right part of Figure 1(b), three additional diff behaviors are identified for a given patch: (1) changing the memory allocation functions for `em` in Lines 4, 10, and 11, (2) splitting the conditional statement of Line 5 into two `if` checks in Lines 6 and 12, and (3) updating the functions between Lines 18 and 20, and Lines 19 and 21, where a more secure function replaces the previous one. By combining the pre- and post-patch's CDG, DDG, and diff behaviors, PatchGraph provides a comprehensive view of patch code changes to better understand the purpose of the changed code.

Unraveling a Patch with Dependency Analysis. Given the PatchGraph, we propose a two-stage patch dependency analysis to group the modification-dependent changes as individual patches. There is a statement-level analysis that performs slicing with the predefined criteria to determine the scope of each patch segment and a segment-level analysis that analyzes the unvisited dependencies among patch segments and retains the necessary ones to ensure grammar correctness and function completeness.

Statement Dependency Analysis. It performs both diff behavior guided slicing and pattern guided slicing over the code statements. First, three types of diff behaviors are used to guide the slicing. The slicing results are three segments (*i.e.*, s1, s2, and s3) as shown in Figure 1(c) ①. Among them, we consider s3 (Lines 18-21) as a security patch candidate since it includes a security function replacement. Then, we slice the PatchGraph guided by patch patterns derived from empirical knowledge, including anchor nodes, slicing directions with termination criteria, and security attribute indicators. As demonstrated in Figure 1(c) ②, the anchor nodes are if statements checking exception value (Line 12) with a forward slicing to obtain error-handling statements (Lines 13 and 14), and a backward slicing to get the assignment of *em* (Line 11). In this way, we obtain the patch segment s4 (Lines 11-14) and identify it as a security patch candidate due to its complete error-handling process. After the above diff behavior guided slicing and pattern guided slicing, we obtain four patch segments (s1-s4), which serve as the basic components for generating individual patches.

Segment Dependency Analysis. The segment level analysis re-examines whether the unvisited dependencies across patch segments need to be preserved to ensure compilation and grammar correctness. As shown in Figure 1(c), the dependency between s1 and s2 as well as the dependency between s1 and s4 are essential. The dependencies between s1 and s3, and s3 and s4 are deemed dispensable as the values of `em` and `db` remain unchanged regardless of these dependencies. After merging the essential dependent segments, we obtain two individual patches, *i.e.*, s1+s2+s4 and s3. Then, we further check the security patch candidates contained in each individual patch to decide if an individual patch is a security patch. Since the added `em` check and error handling in Lines 11-14 have already been performed in Line 5 of the

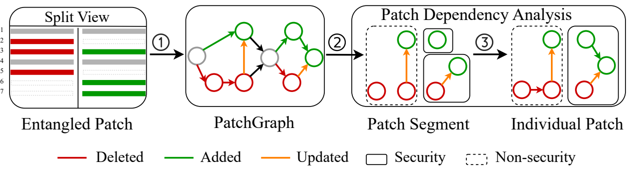


Figure 2: Overview of DISPATCH: ① PatchGraph construction; ② statement dependency analysis; ③ segment dependency analysis.

pre-patch, s1+s2+s4 (containing both Line 5 and Lines 11–14) does not actually introduce a new security check. Therefore, we consider it a non-security patch. Finally, we decompose the original entangled patch into two individual patches: one non-security patch spanning Lines 4 to 15 and one security patch covering Lines 18 to 21, as shown in Figure 1(c).

4 System Overview

DISPATCH decomposes entangled security patches into individual patches, and the overall workflow of DISPATCH is illustrated in Figure 2. Given the version diffs, OSS patch, or Git commit, DISPATCH begins by extracting the modified syntax and dependencies (*i.e.*, diff behaviors) and represents them with a graph representation (*i.e.*, PatchGraph). Next, we perform patch dependency analysis to group the modified statements that need to be applied together (*i.e.*, individual parch). The patch dependency analysis is conducted at two different levels: (1) the statement level, where dependent modifications are grouped into patch segments, and (2) the segment level, where dependent patch segments are combined into individual patches.

4.1 Patch Representation with PatchGraph

Different from representing the classical program graph representation (*e.g.*, AST and PDG), which focuses on recording the syntax structure and capturing the program dependencies in one single version of code, the representation of a patch should be able to capture the changed syntax and dependency between the two versions. Previous methods [44, 52] directly merge the pre-patch and post-patch graph representations, making the graph remain detached, inadequate, and redundant. For instance, Lines 1 and 2 in Figure 3, despite referring to the same code line, are not connected by an edge nor marked as updated. Additionally, while `*p` remains unchanged, its associated dependencies are still maintained. The direct merging will lead to the failure to highlight fine-grained modifications and will neglect to distinguish between the changed and unchanged statements, let alone assist in aggregating changes that serve the same purpose. To fill this gap, we introduce *diff behavior* to describe the actual fine-grained changes and integrate all diff behaviors as *PatchGraph* to represent a patch.

4.2 Patch Dependency Analysis

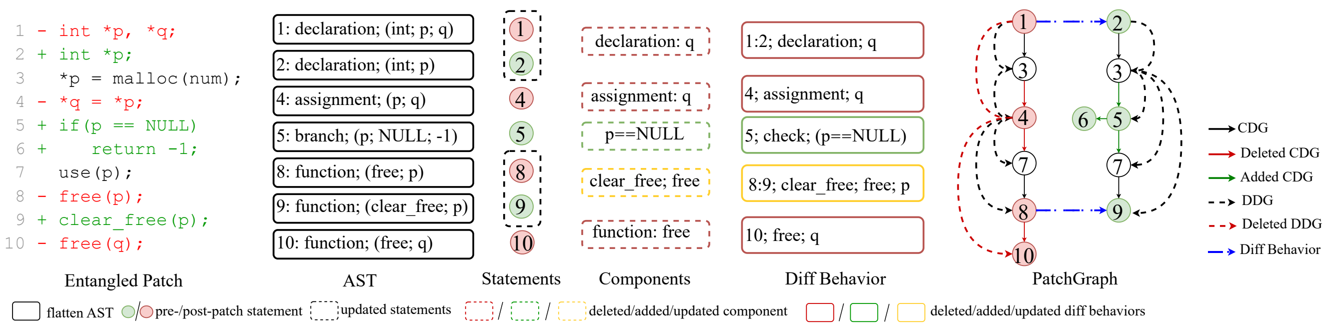
Given the PatchGraph representing an entangled patch, we obtain individual patches by decomposing the large graph into smaller subgraphs through patch dependency analysis. Different from classical program dependency analysis [51, 56], which clusters according to data or control dependencies, patch dependency analysis treats the diff behavior (*i.e.*, delete, add, and update syntax and dependency) as predominant dependencies to link the dependent nodes. To achieve this, we conduct statement-level and patch segment-level analysis.

Patch Statement Dependency Analysis. During the statement-level dependency analysis, we aim to aggregate the code changes for the same purpose. Here, how to define the scope of a modification, specifically where to start the slicing and when to terminate it, is challenging. For example, in Figure 1 (a), although Line 19 is data-dependent on Line 10, they belong to two individual patches, requiring us to define the scope for each. To solve it, we propose a new constrained slicing mechanism named *pattern-guided slicing* to group the modification-dependent statement as patch segments. We empirically summarize a set of security and non-security patch patterns, which are used to initiate the slicing with the anchor nodes, perform program slicing guided and terminated by the constraints to form patch segments, and identify the security patch segment candidates with a security-attribute indicator. In this way, the entangled patch is split into multiple patch segments, and each patch segment resolves one concern (*e.g.*, initialize/reassign a variable, and update function usage).

Patch Segment Dependency Analysis. To guarantee each patch segment can be individually applied (grammar correctness) without disrupting the intended functionalities (functionality correctness), we further analyze the dependencies among the sliced patch segments and merge segments with essential dependencies to generate final individual patches. To further confirm if the individual patch is a security one or not, we analyze the patch segments that constitute an individual patch. If the individual patch only has one type of patch segment, the security characteristic aligns with the segment's label. If the individual patch consists of security patch segments and non-security patch segments simultaneously, we conduct further analysis to check if it is a security patch.

5 Patch Representation

As shown in Figure 3, we construct PatchGraph by overlaying the essential diff behaviors on traditional program dependencies. First, we generate traditional program graph representation, *i.e.*, Abstract Syntax Tree (AST), CDG, DDG, and call graph (CG). Among them, pre-patch and post-patch ASTs are for locating the deleted/added/updated statements and components, *e.g.*, variable, function, value, and string. Pre-patch and post-patch CDGs, DDGs, and CGs are used to recognize modified dependencies. Afterwards, we identify diff behaviors at the statement level. By integrating diff behaviors on top of



```
1  - int *p, *q;
2  + int *p;
3    *p = malloc(num);
4  - *q = *p;
5  + if(p == NULL)
6  +     return -1;
7    use(p);
8  - free(p);
9  + clear_free(p);
10 - free(q);
```

Figure 3: The workflow of PatchGraph construction.

traditional dependencies, we have PatchGraph to precisely represent a patch file.

5.1 Diff Behavior Indentification

We introduce the *diff behavior* to represent the fine-grained code changes of a patch. The diff behavior describes the modification details between pre-patch and post-patch statements. There are three types of modifications and we use them to describe changed statements and the tokens respectively.

Deleted. Deleted statements refer to statements that exist in pre-patch but are absent in the post-patch. The tokens in a deleted statement are considered as deleted components.

Added. Added statements are present in the post-patch but absent in the pre-patch. The tokens in an added statement are considered as added components.

Updated. These statements are present in both pre-patch and post-patch nodes, albeit with some variations. In a patch, updated statements typically come in pairs (except some statements are broken into several lines or several statements have been merged as one), one each from the pre-patch and post-patch, and we introduce the diff behavior edges to link these pairs, with updated components serving as an attribute. The components in an updated statement need to be compared to determine if they have changed. If changed, they will be further tagged with deleted, added, or updated type.

To capture diff behaviors, we first identify the deleted/added/updated statements and then recognize the modified tokens to locate the fine-grained actual updates. Then, the diff behaviors are extracted and recorded using node or edge attributes for PatchGraph construction.

Identifying Deleted/Added/Updated Statements. We learn from the GitHub split view [14] to identify deleted, added, and updated statements in a patch. As shown in the split view (as shown in Figure 2), the deleted statements are only shown in red on the left side and there are no green statements on the right side; the added statements are only shown in green on the right side and there are no red statements on the left side; the updated statements are the statements that have both red deletions on the left side and green additions on the right side. We first identify each hunk using the hunk header (*e.g.*, `@@ -7 +7 @@ void func_name(int a)`) and determine the deleted and added statements within the hunk by recognizing the "-" and "+" prefixes at the beginning of each line. Next, we compare the added and deleted statements within each hunk. If an added statement and a deleted statement share the same statement type (e.g., function declaration) and have variables in common (e.g., function parameters for a function call statement or the left-hand side variable for an assignment statement), we pair them as updated statements. Moreover, we also consider the situation where one statement has been broken into multiple ones (*e.g.*, one if check has been broken into two) and multiple statements have been merged into one (*e.g.*, multiple same type declaration statements have merged into one line). According to these, we can easily recognize deleted, added, and updated statements for the given patch.

Identifying Deleted/Added/Updated Components. Given the deleted/added/updated statements, we recognize fine-grained components. We identify all the necessary elements in deleted statements (*e.g.*, parameters, variables, functions, conditions, branches, loops, and return values) as deleted components. Similarly, we retrieve added components. In updated statements, we first compare tokens between paired statements to eliminate unchanged tokens. Then, we identify the components exclusive to pre-patch statements as deleted, ones only in post-patch statements as added, and ones in both pre-patch and post-patch statements as updated.

Identifying Diff Behaviors. We describe the deleted code behaviors according to the component type and the corresponding deleted tokens by utilizing a 3-tuple $(\textit{modification type}, \textit{token type}, \textit{token details})$, *e.g.*, $(\textit{deleted}, \textit{assignment}, *q)$. Similarly, we describe the added behaviors. For the updated behaviors, besides the type and details of the component, we also need the identifier of the two statements to complete the behavior. We adopt the 4-tuple $(\textit{modification type}, \textit{token type}, \textit{paired statement id}, \textit{token details})$ to describe the above behaviors. For example, as shown in Figure 3, the diff behavior between Lines 8 and 9 are represented as $(\textit{updated}, \textit{function}, \textit{pre\_8: post\_9}, \textit{free: clear\_free})$. The captured diff behaviors represent the properties of nodes and edges in the PatchGraph.

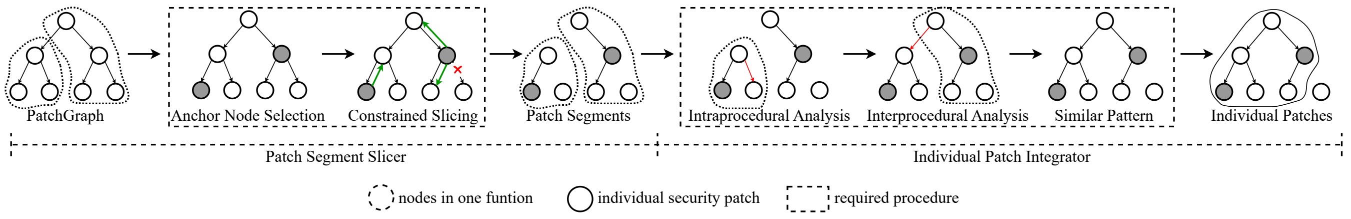


Figure 4: The workflow of individual patch generation.

5.2 PatchGraph Construction

To model patch code changes, we integrate the diff behaviors and classic program representations (*i.e.*, CDGs, DDGs, and CGs) into a joint structure named PatchGraph. To construct PatchGraph, we first generate CDG, DDG, and CG for pre-patch and post-patch code. Since each node in CDG, DDG, and CG is a statement and the edge shows dependencies between two statement nodes (*i.e.*, control dependency, data dependency, and caller-callee relationship), we can merge them into one graph and add the diff behavior on top of it. We use (V,E) to describe the PatchGraph where V is a set of nodes and E is a set of directed edges.

Nodes in PatchGraph. The nodes in PatchGraph are statements in the patch file and we represent the set of nodes V using a 5-tuple $(id, version, code, type, property)$. We parse the patch file to get the nodes represented with the line number in pre-patch and post-patch as node *id* and node *version* ($version \in \{pre\text{-}patch, post\text{-}patch\}$). Then, we pair the statements to recognize the node *type* of each statement: formally, $type \in \{deleted, added, updated, unchanged\}$. Given the statement type, we extract the node *property* from the tokens. If the node type is *updated*, we add the paired node *id* to the property; if the node type is deleted/added, we add the modified content to the property. For example, one of the nodes in Figure 3 can be described as (*pre-8*, *pre-patch*, *free(p);*, *updated*, *post-9:clear_free(p);*).

Edges in PatchGraph. The set of directed edges E is represented by a 6-tuple (id_1, id_2, *version*, *modification type*, *edge type*, *property*), where id_1 and id_2 represent the *id*s of start and end nodes. The *verison* reflects where the node exists, *i.e.*, the pre-patch, post-patch, or both. There are four types of edges in PatchGraph: (1) edges in pre-patch CDG and post-patch CDG, (2) edges for the deleted/added/updated variables in DDG, (3) edges in pre-patch CG and post-patch CG, and (4) edges connecting updated statement pairs introduced by diff behavior. We use the *modification type*, *edge type*, and the *property* together to record these fine-grained modifications, where $modification\ type \in \{deleted, added, updated, unchanged\}$ and $edge\ type \in \{CDG, DDG, CG, diff\ behavior\}$. The *property* records the CDG edges, the defined/used variable for DDG edges, the callee function for CG edges, or the modified *token type* and *token details* identified by diff behavior. For example, the edge between Line 8 and Line 9 in Figure 3 is described as (*pre-8*, *post-9*, *both*, *updated*, *diff behavior*, *function* | *free:clear_free*).

6 Individual Patch Generation

Given the detailed changes represented in PatchGraph, we conduct program slicing to split the entangled patch into multiple individual patches. By empirically proposing a set of patch patterns, which map the modified syntax to modification purpose, to guide the slicing, we obtain patch segments where each one resolves one issue. Moreover, to guarantee the completeness of each individual patch, we further analyze the relationship among patch segments to merge modification-dependent patch segments and guarantee they can by applied back. The workflow is shown in Figure 4.

6.1 Patch Pattern Discovery

To determine the scope of each individual patch, we first empirically summarize a set of patch patterns that can help locate the start and termination statement of each slice, as well as provide an interpretation of whether a slice performs a security fix. To this end, we manually analyze 5k security patches identified in NVD [7] and PatchDB [53] to summarize the security patterns upon diff behaviors. Then, we randomly select 5k non-security commits in Github for non-security patch patterns. After cross-verification by three doctoral students, each with more than three years of experience in software security, we summarize 12 types of security patch patterns and three types of non-security patch patterns. In this work, we focus on C/C++ since they are the languages with the highest number of security issues [31]. Note that the pattern set is extensible, and users can define new patterns according to specific application scenarios.

Patch Pattern Structure. To represent each individual patch and provide insight into its security characteristics, we introduce a new structure consisting of the *anchor node*, *slicing direction with termination criteria*, and *security attribute indicator*. Specifically, anchor nodes serve as the starting statements to initiate the slicing. The slicing direction with termination criteria determines which statements will be associated with the anchor node. If there are no specified termination criteria, we do not conduct slicing in that direction. Such a slicing result is a set of statements completing one purpose. We name it *patch segment*. To determine if a patch segment is security-related, we further define the security attribute indicator. The slices that fail to match the security attribute indicator will be labeled as non-security patch segments. The patterns are represented as a 4-tuple: (*anchor node*, *forward slicing termination* *criteria*, *backward slicing criteria*, *security indicator*). The anchor node and security indicator specify the statement type and associated keywords, while the termination criteria define the selected dependency and the termination statement type. For example, Lines 5 and 6 in Figure 3 illustrate a pattern instance for *Add Sanity Check*. The anchor node corresponds to conditional statements with the keyword if. Forward slicing is constrained by control dependencies and terminates at the end of the if check. Backward slicing is limited to identifying updates to variables within the if check. The security indicator validates the presence of specific keywords in the conditional statement, such as NULL, not NULL, MAX, MIN, other invalid or boundary values, or negative return status.

Security Patch Pattern. We summarize 12 security patterns in four categories, *i.e.*, sanity check, function operation, variable operation, and return statement, and represent these patterns with the above structure. The summary can be found in Table 1 and the detailed description in Appendix B.1.

Non-security Patch Pattern. To distinguish security and non-security patches, we summarize non-security patterns, including debugging, code refactoring, and new features. The summary can be found in Table 1 and the detailed description in Appendix B.2.

6.2 Patch Statement Dependency Analysis

Then, we conduct customized program slicing to produce patch segments, serving as candidates for individual patches. This process involves two methods: diff behavior-based clustering to group similar changes and pattern-guided slicing to collect statements for similar issues.

Diff Behavior Based Clustering. We conduct slicing to group nodes linked by same diff behavior into a patch segment that solves one issue. We assess if they are security-related patch segments by checking if they introduce restricted conditions or apply more memory management actions, *e.g.*, `free()` in s3 of Figure 1.

Slicing Guided by Patch Patterns. According to the summarized patterns, we perform slicing to cluster statements for individual modification purposes. We start our slicing from the statements that match the definition of anchor nodes in our summarized patterns. One slicing from a statement for a specific pattern is performed at a time. For example, Line 5 in Figure 3 has been identified as an anchor node that belongs to the add sanity check pattern since it is an if statement. For each anchor node, the backward and forward slicing is performed on specified dependencies until our defined end condition is met or the start/end of the code is achieved. For instance, given the anchor node Line 5, we perform forward slicing to the end of the check body and backward slicing to locate the new declaration of `p`. Thus, we get Lines 5 and 6 as a patch segment as `p` has not been newly defined. The generated slice (*i.e.*, patch segment) will be labeled as security or non-security according to our defined security indicators. To exemplify, Lines 5 and 6 are a security patch segment since they check the `NULL` in the conditional statement and return a negative status, indicating the error-handling process.

6.3 Patch Segment Dependency Analysis

Our final goal is to generate security patches that can fully solve one issue without disrupting the original functionalities. To achieve this, DISPATCH further analyzes the dependencies among patch segments and groups the segments sharing the required dependencies as individual patches. Then, DISPATCH confirms if each individual patch is a security patch.

6.3.1 Patch Segment Integrator

To determine if the current segment relies on other segments, we examine the remaining intraprocedural and interprocedural dependencies. Also, we conduct a similarity analysis to group similar patch segments into one patch so that the same issue can be addressed thoroughly at one time.

Intraprocedural Remaining Dependency Analysis. To merge the patch segments that have the indispensable data/-control dependencies, we first focus on the unvisited dependencies within the function. We assess the necessity of unused CDG and DDG edges from grammatical viewpoints. For CDG edges, we ensure the nested statements' completeness, retaining dependencies in nested segments. We include DDG edges if variables in later patches are newly defined and assigned, discarding dependencies not in the case.

```
diff --git a/.../v3_purp.c b/.../v3_purp.c
+const ASN1_INTEGER *X509_..._serial(X509 *x) {
+    X509_check_purpose(x, -1, -1);
+    return(x->akid != NULL? x->akid->serial : NULL);
+}
diff --git a/.../v3_purp.c b/.../v3_purp.c
+const GENERAL_NAMES *X509_..._issuer(X509 *x) {
+    X509_check_purpose(x, -1, -1);
+    return(x->akid != NULL? x->akid->issuer : NULL);
+}
```

Listing 2: An example of similarity analysis.

Interprocedural Remaining Dependency Analysis. The interprocedural analysis aims to complete the patch across functions and files, like function calls of modified or newly added functions, which are supposed to be applied together. To do so, we analyze the relationship between the caller and the callee. If the callee function is newly added or the callee's parameters have changed, we keep the call graph edges between them to complete the call chain. Moreover, the interprocedural dependency analysis handles the definition and use of `macro`, identifying the new macro identifiers and merging their definitions' and usages' patch segments.

Patch Segment Similarity Analysis. Besides the above dependency analysis, we also conduct a similarity analysis to fill the gap that patch segments may share the same fixing structure to patch the same type of vulnerabilities without any call, data, control, or modification dependencies. For instance,

Table 1: The patch patterns deployed in pattern-guided slicing.

Type	Category	Patterns	Structure			
			Anchor Nodes	Backward Slicing	Forward Slicing	Security Indicator
Security	Sanity Check	Add Security Check	added if check	variable def of if checks	end of if	error handling
		Update Security Check	updated conditional statement	variable def of if checks	end of if	changed condition
	Function	Add Mem Deallocation Func	added memory function	allocation function	-/NULL assignment	memory deallocation
		Update Mem Management Func	updated memory releasing func	-	-	memory releasing
		Add Concurrency Control	added lock function	locked variable definition	unlock function	unlock function
		Update Concurrency Control	deleted lock function	-	unlock function	deleted lock function
		Function Encapsulation	deleted if check	-	sealed in new func	error handling
	Variable	Initialize Variable	updated variable assignment	-	-	NULL
		Enlarge Buffer Size	updated pointer declaration	-	-	enlarged buffer size
	Statement	Update Return Value	updated return statement	return value assignment	-	error status
		Enhance Error Handling	added goto statement	deleted return statement	goto content	error status
		Update Return Type	void type in pre-	changed type in post-	added return stmt	error status
Non Security	Debug	Debug	added debug function	-/deleted debug function	-	debug function
	Refactor	Refactor	added conditional statements	deleted conditional statements	end of conditional block	same condition
	New Feature	New Feature	function definition	-	end of function	-

in Listing 2, two functions have the total same patch logic and structure to handle x509 certificates, and they should be patched together. By calculating the similarity analysis between the patch segments, the same issues are addressed thoroughly in one individual patch. To capture such syntactic similarity, we calculate the sequential similarity among patch segments. The sequential similarity is calculated using Levenshtein distance [55], which is straightforward and efficient. We consider two segments as similar pairs when the sequential similarity is greater than 0.9.

6.3.2 Individual Security Patch Identification

Given the individual patches, if they only contain one type of patch segment, *i.e.*, security patch segments or non-security patch segments, we label the individual using the patch segments' label. For the remaining individual patches that include the security patch segment and non-security patch segment at the same time, we conduct a two-fold analysis. First, if the security patch candidate is part of a non-security patch segment, given the fact that the goal of the non-security segment is to reformat, debug, refactor, or introduce new features, not specifically to address the security issue, we treat such an individual patch as non-security. Second, we check if the functionality of the security patch candidate already exists in the pre-patch by comparing if they contain the same source code for conditions in sanity checks. If the conditions in the security segment already exist in the pre-patch code (*e.g.*, s4 in Figure 1.), we treat the individual patch as a non-security patch. The remaining individual patches are labeled as individual security patches.

7 Evaluation

In this section, we conduct experiments on real-world projects to answer the following research questions (RQs).
RQ1: What is the accuracy and scalability of DisPatch to unravel entangled patches? (§ 7.1)
RQ2: Can DisPatch outperform baselines? (§ 7.2)
RQ3: How DisPatch performs when identifying individual security patches? (§ 7.3)
RQ4: How DisPatch performs in other software? (§ 7.4)
RQ5: How DisPatch performs when decomposing sophisticated commits? (§ 7.5)
Experiment Setup. Our experiments are conducted on a machine with an Intel Core i7 1.8GHz CPU and 16GB of memory, running Ubuntu 22.04. We use four popular software, *i.e.*, OpenSSL (446,747 LOC), Nginx (242,153 LOC), ImageMagick (874,459 LOC), and the Linux Kernel (18,963,973 LOC), to evaluate the performance of DisPatch and the coverage of the extracted patterns.

7.1 Accuracy and Scalability (RQ1)

To evaluate the effectiveness of the proposed system on entangled patch decomposition, we apply DisPatch on diffs between every two neighboring letter versions [10] in OpenSSL 1.1.1 because letter version contains more bug fixes and security patches, and each diff represents an entangled patch that simultaneously contains multiple entangled security and non-security patches. However, there are not always 1-to-1 mappings between commit and individual-patch. An individual patch may consist of multiple commits, and a commit may contain multiple individual patches. Therefore, we manually identify all the individual patches of all these 23 letter version diffs as the ground truth. We use the Exact Match and Accuracy as the evaluation metrics. As the name suggests, exact match refers to the individual patches generated by DisPatch that are the same as the ground truth line by line. The accuracy refers to the percentage of exactly matched individual patches.
Accuracy. We first use Figure 5 to plot the cumulative distribution function (CDF) of unraveling accuracy results on all the neighboring letter version diffs of OpenSSL 1.1.1. We conduct an ablation study to uncover the contributions introduced by each design of our system, *i.e.*, slicing with patch

Table 2: Ablation study results on patch unraveling in OpenSSL 1.1.1.

Version	Groundtruth	Traditional Slicing			Slicing with Diff Behavior			Slicing with Diff Behavior and Patch Statement Analysis			DisPatch		
	# IP	**# GP**	**# EMIP**	**Accuracy**	**# GP**	**# EMIP**	**Accuracy**	**# GP**	**# EMIP**	**Accuracy**	**# GP**	**# EMIP**	**Accuracy**
0-a	122	420	38	31.15%	318	50	40.98%	236	80	65.57%	151	102	**83.61**% (+52.46%)
a-b	113	602	37	32.74%	423	45	39.82%	366	62	54.87%	129	103	**91.15**% (+58.41%)
b-c	81	977	39	48.15%	532	42	51.85%	135	57	70.37%	94	73	**90.12**% (+41.97%)
c-d	118	401	35	29.66%	343	47	39.83%	291	80	67.8%	131	108	**91.53**% (+61.87%)
d-e	114	1760	42	36.84%	947	47	41.23%	175	70	61.4%	125	107	**93.86**% (+57.02%)
e-f	11	57	2	18.18%	46	2	18.18%	38	4	36.36%	15	9	**81.82**% (+63.64%)
f-g	16	97	8	50%	46	9	56.25%	38	10	62.5%	20	12	**75**% (+25%)
g-h	82	221	40	48.78%	195	47	57.32%	176	59	71.95%	89	76	**92.68**% (+43.9%)
h-i	24	36	10	41.67%	27	12	50%	36	16	66.67%	27	21	**87.5**% (+45.83%)
i-j	29	34	13	44.83%	31	15	51.72%	39	20	68.97%	32	26	**89.66**% (+44.83%)
j-k	17	40	8	47.06%	37	9	52.94%	36	10	58.82%	29	15	**88.24**% (+41.18%)
k-l	72	142	27	37.5%	140	32	44.44%	133	40	55.56%	89	67	**93.06**% (+55.56%)
l-m	58	86	40	68.97%	77	40	68.97%	73	50	86.21%	66	53	**91.38**% (+22.41%)
m-n	23	47	13	56.52%	37	16	69.57%	29	18	78.26%	28	19	**82.61**% (+26.09%)
n-o	22	45	13	59.09%	41	13	59.09%	29	13	59.09%	22	22	**100**% (+40.91%)
o-p	19	26	10	52.63%	23	10	52.63%	29	13	68.42%	23	15	**78.95**% (+26.32%)
p-q	5	11	1	20%	9	1	20%	8	2	40%	6	4	**80**% (+60%)
q-r	33	65	16	48.48%	54	18	54.55%	52	18	54.55%	35	31	**93.94**% (+45.46%)
r-s	4	14	0	0%	11	0	0%	10	1	25%	3	3	**75**% (+75%)
s-t	23	36	11	47.83%	33	12	52.17%	30	19	82.61%	27	22	**95.65**% (+47.82%)
t-u	24	38	6	25%	37	7	29.17%	35	16	66.67%	30	21	**87.5**% (+62.5%)
u-v	3	7	1	33.33%	7	1	33.33%	6	2	66.67%	3	3	**100**% (+66.67%)
v-w	12	25	3	25%	18	4	33.33%	16	10	83.33%	14	11	**91.67**% (+66.67%)
Total	1025	5187	413	40.29%	3432	479	46.73%	2016	670	65.37%	1188	923	**90.05**% (+49.76%)

# IP is for the number of individual patches; **# GP** is for the number of generated patches; **# EMIP** is for the number of exactly matched individual patches.
DisPatch is slicing with diff behavior, patch statement analysis, and patch segment analysis.

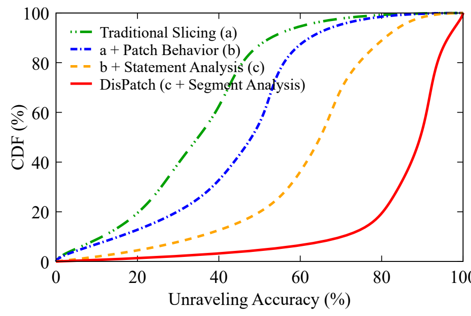


Figure 5: CDF of unraveling accuracy on OpenSSL 1.1.1.

behavior, patch statement analysis, and patch segment analysis. From the figure, we can see that the unraveling accuracy for the traditional slicing (based on control and data dependency) ranges from 0% to 68.97%, with the most distributed under 50%. After adding patch behavior for slicing, the result is improved, and the best accuracy achieves 69.57%. Then we additionally adopt patch statement analysis, and the unraveling accuracy is in the range of 25% to 86.21%. Finally, after incorporating the patch segment analysis, DisPatch performs the best, with the lowest accuracy of 75% and the highest accuracy of 100%.

We list the unraveling result on each letter version of OpenSSL 1.1.1 in Table 2. The first column lists the version numbers of two consecutive versions of OpenSSL, which respectively served as the original and patched source code. The second column reports the number of manually identified individual patches, serving as the ground truth. The columns from third to fourteenth show the results of entangled patch decomposition based on traditional slicing (columns 3-5), slicing with patch behavior (columns 6-8), slicing with patch behavior and statement analysis (columns 9-11), and slicing with patch behavior, statement analysis, and segment analysis (*i.e.*, DisPatch) (columns 12-14).

From Table 2, we can see that among 1025 individual patches in OpenSSL 1.1.1, DisPatch successfully unravels 923 of them with an average accuracy of 90.05%, indicating that DisPatch can decompose entangled version diff with high accuracy. As a comparison, the performance of DisPatch outperforms the traditional slicing without diff behavior by about 50%. Besides, from the ablation study results, when patch behavior and statement analysis are introduced, the number of generated decomposed patches decreases in most cases, and the unraveling accuracy keeps increasing. This indicates that the patch behavior provides essential information among patch code changes, and the patch statement analysis can further aggregate the modifications with the same pattern. For versions `f-g`, `h-i`, and `o-p`, the number of decomposed patches decreases when the patch behavior is introduced, while the number increases with the patch statement analysis, which reveals that the pattern-guided slicing cuts off the redundant dependencies efficiently, but we still need patch segment analysis to obtain an individual patch. The result implies the importance of each component of DisPatch and signifies the necessity to make them work together.

We also analyze the cases where DisPatch fails to decompose and summarize three main reasons. First, the same modification semantics may have divergent ways of presentation. For example, in the Listing 3, the patch aims to enlarge the cookie size by changing the function name and the value of the corresponding macro. However, these two hunks do not have any dependencies or similarities, resulting in that

DISPATCH fails to integrate the two types of modifications. Second, the same value of the same purpose may be assigned to different variables. For example, in the Listing 4, the modifications in the two hunks aim to initialize the length of an SSL object. However, the initialization operations happen in different functions with different variable names; hence, such semantics fail to be captured by the diff behavior and similarity analysis. Third, DISPATCH fails to capture the indirect data dependencies. For example, in the Listing 5, `s->session` is instantiated from the struct `ssl_session_st`. Since the element of the struct `ssl_session_st` has been deleted, the other instance should also delete the usage of the deleted element. However, DISPATCH fails to capture the two-hop data dependency. In such cases, our semantic analysis may detect significant semantic differences between the pre-patch and the post-patch, failing to integrate such differences.

```
diff --git a/ssl/packet_local.h b/ssl/packet_local.h
-            || !PACKET_get_net_4(&cookie, &tm)
+            || !PACKET_get_net_8(&cookie, &tm)
diff --git a/.../..._srvr.c b/.../..._srvr.c
-#define MAX_COOKIE_SIZE (...+4+2+EVP_MAX_MD_SIZE
+#define MAX_COOKIE_SIZE (...+8+2+EVP_MAX_MD_SIZE
```

Listing 3: Same update purpose with different presentations.

```
diff --git a/ssl/s3_lib.c b/ssl/s3_lib.c
+        s->s3->tmp.psklen = 0;
diff --git a/.../statem_clnt.c b/.../statem_clnt.c
+   s->s3->tmp.pmslen = 0;
```

Listing 4: Same value assignment to independent variables.

```
diff --git a/ssl/ssl_local.h b/ssl/ssl_local.h
-    int peer_type;
diff --git a/.../statem_clnt.c b/...m/statem_clnt.c
-    s->session->peer_type = certidx;
```

Listing 5: Missing critical data dependency.

Scalability. As shown in Table 2, Figure 6 and Appendix D, the 23 entangled patches (*i.e.*, version diffs) consist of 3-122 individual patches (6-128 commits) as well as involve 4-716 modified files and 103-15,645 changed LOC. Among these entangled patches, DISPATCH maintains good performance consistently, indicating the scalability of DISPATCH to handle large patches. In summary, with PatchGraph and tailored slicing mechanism, DISPATCH can unravel the entangled patches with high accuracy and scale well to large entangled patches.

7.2 Comparison with Baseline Models (RQ2)

To further evaluate the performance of DISPATCH on general patch decomposition, we conduct comparison experiments with SOTA methods. We choose three representative works from different categories: SPatch [36] (a slicing-based approach), GPT-4 [16] (an LLM-based approach), and UTANGO [32] (a GNN-based approach). Note that these existing works only focus on general patch decomposition and DISPATCH is the first to unravel individual security patches.

Comparison with SPatch. SPatch divides a patch into multiple Change Units (CUs) as individual patches, integrating code changes from a single edit action—add, delete, update, move—that share a data flow into one CU. Since the source code is not released, we reimplement it following the design.

From Table 3, among 1025 individual patches in OpenSSL 1.1.1, SPatch successfully unravels 405 of them with an average accuracy of 39.51%, indicating that SPatch has limited ability to unravel the entangled version diff. Besides, we can also notice that the number of generated patches is larger than the ground truth and the average size of the generated patch is much smaller than actual ones. Compared with DISPATCH, The reasons behind the results are from three aspects. First, SPatch neglects the control flow, missing the critical execution logic, especially for the no-parameter functions. Second, SPatch does not take inter-procedural relationships, *e.g.*, the call dependency, and macro usages, which also cause the failure of bridging the dependent edit actions. Third, although the SPatch considers the update edit action, it still forgets to utilize the fine-grained updated information as attributes to merge the changes. To sum up, DISPATCH outperforms SPatch by increasing 50.54% of the unravelment accuracy.

Comparison with GPT-4. Large language models (LLMs) have been widely used in code understanding tasks, *i.e.*, security patch analysis [19, 28, 49]. The performance of LLMs in patch analysis is often influenced by how well the prompts are constructed. We choose GPT-4 and design the prompt in three content [19]: (1) basic content simply provides the task that we want GPT-4 to solve, (2) reward content uses motivational phrases like "If you perform very well, I will generously tip you.", and (3) punish content warns "If you perform badly, you will be fined." on two experiment settings: (1) zero-shot and (2) few-shot. For the second setting, we instruct GPT-4 to unravel entangled patches with two examples. Then, we ask GPT-4 to identify the individual ones.

From Table 3, we show the result from zero-shot with basic content, which is the best among the above six settings. There are seven blank results, and the reasons are from two aspects. First, GPT-4 can not process the version diff that exceeds their maximum token length, *e.g.*, `a-b` and `b-c`. Second, since our prompt is constructed by asking GPT-4 to decompose the version diff and then providing the version diff, GPT-4 may forget the instruction. Thus, GPT-4 may reply with some unrelated answers, *e.g.*, the output of `0-a` is an image. Compared with SPatch, GPT-4 can extract interprocedural semantics, but this will cause GPT-4 to group the changes resolving different purposes while belonging to the related function together, leading to the number of generated patches being much smaller while the average size is much larger. Therefore, GPT-4 only has a limited ability to understand tangled commits and identify the individual patches.

Comparison with UTANGO. UTANGO decomposes tangled commits with a context-aware, graph-based, code change representation learning model. Initially, they employ GloVe embedding [45] for each token and compute the average embedding to represent each statement. Subsequently, they uti-

Table 3: Comparison with slicing, LLM, and GNN based patch unraveling solutions.

Version	Groundtruth	SPatch [36]			GPT-4 [16]			UTANGO [32]			DISPATCH		
	# IP	# GP	# EMIP	Accuracy	# GP	# EMIP	Accuracy	# GP	# EMIP	Accuracy	# GP	# EMIP	Accuracy
0-a	122	365	42	34.43%	-	-	-	306	45	36.89%	151	102	**83.61%**
a-b	113	527	31	27.43%	-	-	-	371	40	35.98%	129	103	**91.15%**
b-c	81	614	39	48.15%	-	-	-	208	33	40.74%	94	73	**90.12%**
c-d	118	525	42	35.59%	-	-	-	417	39	33.05%	131	108	**91.53%**
d-e	114	1131	34	29.82%	-	-	-	1652	48	42.11%	125	107	**93.86%**
e-f	11	54	2	18.18%	17	4	36.36%	43	3	27.27%	15	9	**81.82%**
f-g	16	394	7	43.75%	-	-	-	358	11	68.75%	20	12	**75%**
g-h	82	261	40	48.78%	-	-	-	230	37	45.12%	89	76	**92.68%**
h-i	24	47	11	45.83%	18	7	29.17%	33	12	50%	27	21	**87.5%**
i-j	29	41	14	48.28%	7	0	0%	36	13	44.83%	32	26	**89.66%**
j-k	17	44	9	52.94%	14	8	47.06%	34	10	55.82%	29	15	**88.24%**
k-l	72	167	29	40.28%	6	4	5.56%	136	29	40.28%	89	67	**93.06%**
l-m	58	90	42	72.41%	3	3	5.17%	69	36	62.07%	66	53	**91.38%**
m-n	23	49	13	56.52%	13	11	47.83%	37	16	69.57%	28	19	**82.61%**
n-o	22	53	11	50%	20	13	59.09%	45	11	50%	22	22	**100%**
o-p	19	36	7	36.84%	15	9	47.37%	26	11	57.89%	23	15	**78.95%**
p-q	5	11	1	20%	4	4	80%	7	3	60%	6	4	**80%**
q-r	33	63	14	42.42%	3	1	3.03%	55	14	42.42%	35	31	**93.94%**
r-s	4	15	0	0%	3	3	75%	11	2	50%	3	3	**75%**
s-t	23	38	9	39.13%	18	6	26.09%	66	10	43.48%	27	22	**95.65%**
t-u	24	40	5	20.83%	14	8	33.33%	68	6	25%	30	21	**87.5%**
u-v	3	6	0	0%	4	2	66.67%	5	1	33.33%	3	3	**100%**
v-w	12	26	3	25%	8	5	41.67%	19	6	50%	14	11	**91.67%**
Total	1025	4597	405	39.51%	167	88	23.22%	4232	436	42.54%	1188	923	**90.05%**

lize a Label-GCN (Graph Convolutional Network) [21] to incorporate contextual information into the embedding. Finally, they apply an agglomerative clustering algorithm [38] to cluster the code changes. UNTANGO releases the code and model based on Java/C# [6]. We extend it to properly embed C/C++ code by training the GloVe model with OpenSSL and then feeding the embedding to their released model. Note that the original evaluation metrics used in UNTANGO are to calculate how many statements have been assigned to the correct cluster that represents an individual patch. Since we value the completeness of the generated patch, we evaluate how many individual patches can be identified accurately.

According to the discussion in UTANGO, it is less accurate when the number of changed statements is more than 40 [32]. However, the minimum line of the changed statement in our dataset exceeds 40. Besides, since UTANGO has been trained on Java/C#, UTANGO has a limited understanding of C/C++ code. Moreover, UTANGO overlooks the interprocedural relationships. To ensure a fair comparison, we evaluate UTANGO at the function level, which restricts the number of changed statements per decomposition. As shown in Table 3, UTANGO achieves an average unraveling accuracy of 42.5%, demonstrating its limited ability to decompose individual patches, even with fewer than 40 changed statements.

From the above comparisons, we conclude that the existing decomposition approaches are not capable of decomposing individual patches from entangled version diff effectively. Moreover, SPatch and UTANGO cannot identify security-related individual patches. They lack a mechanism to distinguish security patches and non-security patches and mix the security patches with any update related to them.

7.3 Unraveling Security Patch (RQ3)

To evaluate the ability of DISPATCH to decompose the individual security patches, we consider all of the individual security patches in 23 versions of OpenSSL 1.1.1, including the public security patches indexed by CVEs and the silent security patches annotated by previous papers [53,57,58]. They were used as the ground truth to calculate recall and precision of DISPATCH on generating individual security patches.

As shown in Table 4, among 504 ground-truth security patches, 465 are successfully identified by the DISPATCH, indicating a recall of 91.9%. Among 531 individual security patches unraveled by the DISPATCH, 465 of them are real security patches, resulting in a precision rate of 87.75%. Moreover, we check the results on official security patches that are publicly indexed by CVEs in OpenSSL 1.1.1 [4]. 87.5% of patches annotated by CVEs are correctly decomposed by DISPATCH. More detailed results can be found in Appendix E Table 11.

We analyze the cases where DISPATCH fails to unravel and identify the individual security patches. First, the individual security patches are not completely decomposed from the entangled diff. The reason can be found in § 7.1. Second, the individual patches are completely decomposed but labeled as non-security individual patches. This is mainly because of the limited scope of the patterns derived from the existing security patch dataset. For example, DISPATCH fails to detect the individual patch in the version diff between OpenSSL 1.1.1t and 1.1.1u that fixes the timing leak [2]. This vulnerability is fixed by updating the variable assignment logic, which is beyond our consideration.

We also investigate the false positives and identify two main causes: incomplete decomposition of individual patches (detailed in § 7.1) and mislabeling non-security patches

Table 4: Security patch unraveling results in OpenSSL 1.1.1.

Version	Groundtruth	DISPATCH			
	# ISP	# GSP	# EMISP	Recall	Precision
0-a	53	62	46	86.79%	74.19%
a-b	56	63	51	91.07%	80.95%
b-c	33	34	31	93.94%	91.18%
c-d	64	59	59	92.19%	100%
d-e	44	48	42	95.45%	87.5%
e-f	5	4	4	80%	100%
f-g	8	7	5	62.5%	71.43%
g-h	43	44	42	97.67%	95.45%
h-i	9	8	7	77.78%	87.5%
i-j	14	15	14	100%	93.33%
j-k	9	9	9	100%	100%
k-l	37	40	35	94.59%	87.5%
l-m	33	36	30	90.91%	83.33%
m-n	10	13	10	100%	76.92%
n-o	10	10	9	90%	90%
o-p	11	12	8	72.73%	66.67%
p-q	4	4	4	100%	100%
q-r	19	19	18	94.74%	94.74%
r-s	1	2	1	100%	50%
s-t	13	13	13	100%	100%
t-u	15	16	14	93.33%	87.5%
u-v	2	2	2	100%	100%
v-w	11	11	11	100%	100%
Total	504	531	465	91.9%	87.75%

# ISP: the number of individual security patches.
# GSP: the number of generated security patches.
# EMISP: the number of exactly matched individual security patches.

Table 5: Patch unraveling results.

Application	Groundtruth	DISPATCH		
	# IP	# GP	# EMIP	Accuracy
Nginx	26	29	24	92%
Linux Kernel	114	118	113	99%
ImageMagick	50	58	48	96%
Total	190	205	185	95%

as security-related due to context absence. For instance, `cleanup_old_md_data` in Listing 6 was mistakenly tagged as a security patch, but it's merely a refactor, including an if-check without changing the security level.

```
diff --git a/crypto/digest.c b/crypto/digest.c
-   if (ctx->digest && ctx->digest->ctx_size) {
-       OPENSSL_clear_free(ctx->md_data, ctx_size);
-       ctx->md_data = NULL;
-   }
+   cleanup_old_md_data(ctx, 1);
```

Listing 6: Lack of context information.

Based on these observations, DISPATCH can identify individual security patches with the help of the pre-defined patterns, achieving a high recall and precision.

New Security Patch Identification. As shown in Table 11 (Appendix E), there are 24 security patches publicly indexed by NVD [7] in OpenSSL 1.1.1. When labeling the ground truth, we identify 480 more security patches. Among these newly identified security patches, DISPATCH can decompose and pinpoint 444 (92.5%) of them, indicating DISPATCH is able to identify undisclosed security patches.

7.4 Effectiveness on Different Software (RQ4)

To evaluate the pattern's coverage and DISPATCH's performance across diverse software ecosystems, we evaluate DISPATCH on three other popular open-source projects covering different application domains, i.e., Linux Kernel (operating system core), Nginx (webserver), and ImageMagick (image editing software), each with a significant number of known vulnerabilities. We manually decompose the version diffs into individual patches as the ground truth. Our experiments focus on the following relatively new version comparison: Linux Kernel v5.16-rc7 and v5.16-rc8, Nginx release-1.1.0 and 1.1.3, and ImageMagick 7.0.1-7 and 7.0.1-10.

Table 6: Security patch unraveling results.

Application	Groundtruth	DISPATCH			
	# ISP	# GSP	# EMISP	Recall	Precision
Nginx	12	14	11	92%	79%
Linux Kernel	73	74	73	100%	99%
ImageMagick	24	22	22	92%	100%
Total	109	641	571	94.6%	92.6%

Table 5 shows DISPATCH achieves 95% accuracy, decomposing 185 out of 190 individual patches. DISPATCH's effectiveness and generalization capability across different applications come from 3 key aspects: (1) PatchGraph captures syntax differences by analyzing statement types and comparing fine-grained tokens, which are application-agnostic; (2) the identified 12 security and 3 non-security patterns effectively cover a wide range of real-world patch instances; (3) the pattern-guided slicing adopts fuzzy matching of pattern structures, which is also not application-dependent. We also compare the selected baseline models with DISPATCH. As shown in Table 12 from Appendix F, the results align with the comparison on the OpenSSL dataset.

We further evaluate the effectiveness of DISPATCH on unraveling individual security patches. Table 6 shows that the average unraveling recall is 94.6% and precision is 92.6%. From the experimental results, DISPATCH can effectively decompose and identify individual security patches, while the decomposition ability of DISPATCH is consistent in both security and non-security scenarios.

7.5 Effectiveness on Complex Patches (RQ5)

To assess DISPATCH's effectiveness on complex entangled security patches, two scenarios are exemplified.

Security Patches Entangled with Non-security Patches. Listing 7 demonstrates how DISPATCH decides the boundary between two patches. There are two individual patches in Listing 7. The individual non-security (Lines 1-7) patch refactors the while loop with the for loop, and the individual security patch (Line 8) frees the `fds` to avoid memory leakage. The value of the freed variable is assigned in the non-security patch. The refactoring pattern and the added memory-related function pattern split the entangled patch into two patch segments. The patch segment analysis confirms that `fds` is not new, eliminating the need for merging. This example illustrates how patch statement analysis defines patch boundaries and the necessity of segment analysis.

Security Patches Entangled with Security Patches. Listing 8 showcases a patch that entangles two security patches

```
1 +     size_t i;
2 -     while (numfds > 0) {
3 -         openssl_fdset((int)*fds, &asyncfds);
4 -         numfds--;
5 -         fds++;
6 +     for (i = 0; i < numfds; i++) {
7 +         openssl_fdset((int)fds[i], &asyncfds);
8 +     OPENSSL_free(fds);
```

Listing 7: Security patches entangled with non-security ones.

(Lines 2-3, Line 4). While it is important to timely apply all security patches, there is still a need to decompose entangled security patches to inform users or system administrators about specific details, allowing them to make informed decisions on patching. In this context, DISPATCH demonstrates its capability to decompose such patches, as evidenced by its effective use of update conditional statements and memory-related function patterns to separate individual patches.

```
1 diff --git a/.../pem_pk8.c b/.../pem_pk8.c
2 -     if (klen <= 0) {
3 +     if (klen < 0) {
4 +     OPENSSL_cleanse(psbuf, klen);
```

Listing 8: Security patches entangled with security patches.

8 Discussion

Limitations. DISPATCH uses Tree-sitter [11] to generate AST and adopts MLTA [35] to identify the indirect call targets. Thus, DISPATCH inherits their limitations inevitably. Besides, the current patch patterns may have two limitations. First, we extract patch patterns from public security patch database, where the distribution of vulnerability types may be skewered. However, from our observation, the patch patterns for popular software may not vary a lot. Second, the patterns are generated by security experts, who may introduce bias to certain types and be subjective to distinguish a security patch and a non-security one for some cases (*e.g.*, security patch and non-security may share the same pattern). Here, we follow the guidelines in NVD CWE [42] and classify a patch as a security one if it fixes a vulnerability associated with any CWE type. Moreover, the definition of an individual patch can be subjective, as opinions may differ among experts. Some may consider a single modification as a patch, while others may take it as a part of a fix. To mitigate this subjectivity, we involved three experts in the labeling process. These experts collaboratively studied the individual patches and discussed labeling criteria. To further minimize the subjectivity, we compile and test the project with individual patches.

Usability. DISPATCH can be adapted to other languages (like C, Java, Python, and Go) by extending the Tree-sitter [11]. We will extend our tool to support more languages by introducing more language-specific patterns. Furthermore, DISPATCH is platform-independent since we only utilize source code without any requirements on patch format.

Patch Quality. For the manually labeled patches, the syntax quality has been guaranteed twofold. First, the experts scrutinized the entangled patches and the individual patches with their empirical knowledge. Second, we applied the ground truth back to the pre-patch source code; then, we built and compiled the patched code. As for semantic quality, we randomly select individual patches and check them manually. From the result, semantic correctness can be guaranteed. To ensure the original functionality, we run available test cases (*e.g.*, built-in test cases in OpenSSL [8]) after applying each individual patch. This confirms that the individual patches preserve functionality when the decomposition results align with the ground truth. Additionally, all obtained individual patches are not cumulative and can be applied back independently.

9 Related Work

Entangled Patch Analysis and Decomposition. Researchers have made some efforts to decompose tangled commits. The heuristic-based approaches obtain single-purpose commits by utilizing the pre-defined templates or mining patterns [20, 25, 27, 29, 30, 48, 50]. The slicing-based approaches [25, 39, 56] and graph clustering-based approaches [18, 22, 24, 32, 41, 44, 47, 51] conduct dependency analysis and group the nodes in the dependency graph using program slicing techniques and graph clustering algorithms, respectively. Besides, [25, 47, 56] introduce an interactive mechanism to compensate for the automatic process. Moreover, [23] adopts the information from IDE, and [26, 46] decompose the tangled commits manually.

Constrained Program Slicing. Researchers introduce multiple constraints to guide the program slicing. [34] introduces the key point (center) and slices code gadgets from DFG or CFG. [59] refines code gadgets by proposing code attention, which is a subset of statements matching with specific syntax characteristics. [33] refines the starting point and node types for program slicing. [37] defines manifestation points according to the type of vulnerabilities. Compared with them, pattern-guided slicing offers adaptability to DISPATCH by guiding the slicing procedure with fuzzy matching.

10 Conclusion

We propose DISPATCH to unravel individual security patches from entangled ones with PatchGraph and patch dependency analysis. PatchGraph represents the fine-grained diff behaviors with the enriched graph structure capturing the deleted/added/updated semantics. The patch dependency analysis involves statement and segment dependency analysis to obtain complete individual security patches. We evaluate the effectiveness of DISPATCH on version diffs of four real-world popular projects. The experimental results show DISPATCH effectively unravels more than 90% of general individual patches and over 87% of individual security ones. We compare our approach with SOTA methods and DISPATCH outperforms them, with an accuracy improvement of over 20%.

Acknowledgments

We thank our Shepherd and the reviewers for their insightful feedback. This work is partially supported by ONR grant

N00014-23-1-2122 and an IDIA P3 Faculty Fellowship from George Mason University.

Ethics Considerations

Our work performs program analysis on open-source code from GitHub. All experiments are conducted on publicly available source code and detection models. Since no new vulnerabilities will be generated or identified by this work, no ethical issues will arise.

Open Science

The source code and dataset of this work are available at: https://figshare.com/articles/software/DISPATCH/28256150.

References

[1] "Official Patch for CVE-2021-3449," https://github.com/openssl/openssl/commit/fb9fa6b51defd48157eeb207f52181f735d96148.

[2] "Official Patch for CVE-2022-4304," https://github.com/openssl/openssl/commit/3f499b24f3bcd66db022074f7e8b4f6ee266a3ae.

[3] "Official Patch for CVE-2023-0464," https://github.com/openssl/openssl/commit/879f7080d7e141f415c79eaa3a8ac4a3dad0348b.

[4] "OpenSSL Vulnerabilities Fixed in OpenSSL 1.1.1," https://www.openssl.org/news/vulnerabilities-1.1.1.html.

[5] "grammar zoo - browsable c99 grammar 2015," 2015. [Online]. Available: https://slebok.github.io/zoo/c/c99/iso-9899-tc3/extracted/index.html

[6] "Git Repository of UTANGO." https://github.com/Commit-Untangling/commit-untangling, 2022.

[7] "National Vulnerability Database," https://nvd.nist.gov, 2023.

[8] "OpenSSL Project," https://github.com/openssl/openssl, 2023.

[9] "Patch," https://csrc.nist.gov/glossary/term/patch, 2023.

[10] "Release Strategy of OpenSSL," https://www.openssl.org/policies/releasestrat, 2023.

[11] "Tree-Sitter," https://tree-sitter.github.io/tree-sitter/, 2023.

[12] 2024. [Online]. Available: https://github.com/nothings/stb/commit/98fdfc6df88b1e34a736d5e126e6c8139c8de1a6

[13] 2024. [Online]. Available: https://github.com/FFmpeg/FFmpeg/commit/54655623a82632e7624714d7b2a3e039dc5faa7e

[14] "Split View of a Commit." https://github.com/openssl/openssl/commit/041962b429ebe748c8b6b7922980dfb6decfef26?diff=split&w=0, 2024.

[15] "Understanding Patches and Software Updates." https://www.cisa.gov/news-events/news/understanding-patches-and-software-updates, 2024.

[16] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[17] alexbuckgit, "C docs - get started, tutorials, reference." 2024. [Online]. Available: https://learn.microsoft.com/en-us/cpp/c-language/?view=msvc-170

[18] R. Arima, Y. Higo, and S. Kusumoto, "A study on inappropriately partitioned commits: How much and what kinds of ip commits in java projects?" in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 336–340.

[19] W. Bai, Q. Wu, K. Wu, and K. Lu, "Exploring the influence of prompts in llms for security-related tasks."

[20] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 134–144.

[21] C. Bellei, H. Alattas, and N. Kaaniche, "Label-gcn: an effective method for adding label propagation to graph convolutional networks," *arXiv preprint arXiv:2104.02153*, 2021.

[22] S. Chen, S. Xu, Y. Yao, and F. Xu, "Untangling composite commits by attributed graph clustering," in *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, 2022, pp. 117–126.

[23] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 341–350.

[24] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[25] B. Guo and M. Song, "Interactively decomposing composite changes to support code review and regression testing," in *2017 IEEE 41st Annual Computer Software and Applications Conference*. IEEE, 2017, pp. 118–127.

[26] S. Herbold, A. Trautsch, and B. Ledel, "Large-scale manual validation of bugfixing changes," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 611–614.

[27] K. Herzig, S. Just, and A. Zeller, "The impact of tangled code changes on defect prediction models," *Empirical Software Engineering*, vol. 21, pp. 303–336, 2016.

[28] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1646–1656.

[29] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! are you committing tangled changes?" in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 262–265.

[30] ——, "Splitting commits via past code changes," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2016, pp. 129–136.

[31] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2201–2215.

[32] Y. Li, S. Wang, and T. N. Nguyen, "Utango: untangling commits with context-aware, graph-based, code change clustering learning model," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 221–232.

[33] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

[34] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[35] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.

[36] C. Luo, W. Meng, and S. Wang, "Strengthening supply chain security with fine-grained safe patch identification," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 873–873.

[37] Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, and W. Lee, "Vulchecker: Graph-based vulnerability localization in source code," in *31st USENIX Security Symposium, Security 2022*, 2023.

[38] D. Müllner, "Modern hierarchical, agglomerative clustering algorithms," *arXiv preprint arXiv:1109.2378*, 2011.

[39] W. Muylaert and C. De Roover, "Untangling composite commits using program slicing," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 193–202.

[40] H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization," in *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*. IEEE, 2013, pp. 138–147.

[41] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 819–830.

[42] NIST, "Common Weakness Enumeration Specification (CWE)," https://nvd.nist.gov/vuln/categories.

[43] OpenSSL, "crypto/rsa/rsa_oaep." https://github.com/openssl/openssl/compare/OpenSSL_1_1_0...OpenSSL_1_1_1, 2023.

[44] P.-P. Pârţachi, S. K. Dash, M. Allamanis, and E. T. Barr, "Flexeme: Untangling commits using lexical flows," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 63–74.

[45] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[46] K. A. Safwan and F. Servant, "Decomposing the rationale of code commits: the software developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 397–408.

[47] B. Shen, W. Zhang, C. Kästner, H. Zhao, Z. Wei, G. Liang, and Z. Jin, "Smartcommit: a graph-based interactive assistant for activity-oriented commits," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 379–390.

[48] L. Sousa, D. Cedrim, A. Garcia, W. Oizumi, A. C. Bibiano, D. Oliveira, M. Kim, and A. Oliveira, "Characterizing and identifying composite refactorings: Concepts, heuristics and patterns," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 186–197.

[49] X. Tang, Z. Chen, K. Kim, H. Tian, S. Ezzini, and J. Klein, "Just-in-time security patch detection–llm at the rescue for data augmentation," *arXiv preprint arXiv:2312.01241*, 2023.

[50] Y. Tao and S. Kim, "Partitioning composite code changes to facilitate code review," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 180–190.

[51] M. Wang, Z. Lin, Y. Zou, and B. Xie, "Cora: Decomposing and describing tangled code changes for reviewer," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1050–1061.

[52] S. Wang, X. Wang, K. Sun, S. Jajodia, H. Wang, and Q. Li, "Graphspd: Graph-based security patch detection with enriched code semantics," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2409–2426.

[53] X. Wang, S. Wang, P. Feng, K. Sun, and S. Jajodia, "Patchdb: A large-scale security patch dataset," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 149–160.

[54] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[55] Wikipedia, "Levenshtein distance." https://en.wikipedia.org/wiki/Levenshtein_distance, 2024.

[56] S. Yamashita, S. Hayashi, and M. Saeki, "Changebeadsthreader: An interactive environment for tailoring automatically untangled changes," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 657–661.

[57] J. Zhou, M. Pacheco, J. Chen, X. Hu, X. Xia, D. Lo, and A. E. Hassan, "Colefunda: Explainable silent vulnerability fix identification," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2565–2577.

[58] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "Spi: Automated identification of security patches via commits," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–27, 2021.

[59] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019.

A Entangled Patch Adoption

To justify that entangled patches are hard to merge while smaller individual patches are easier to adopt, we conduct case studies on OpenSSL's pull requests. We analyze 339 closed pull requests in OpenSSL, comprising 200 unmerged and 139 merged requests (as only 139 closed pull requests were merged). As shown in Table 7, 221 involve C code, with 92 from merged pull requests and 129 from unmerged ones. Among the C-related pull requests, the entangled rate of unmerged pull requests is nearly 20% higher than the merged ones, indicating the difficulties of entangled patch adoption.

Table 7: Entangled Rate of Pull Requests (PRs) in OpenSSL.

	# PR	# C-related	# Entangled	Entangled Rate
Merged PR	139	92	27	29.34%
Unmerged PR	200	129	63	48.84%

Table 8: Metadata of Merged Pull Requests in OpenSSL.

Metrics	Type	Duration (h)	# Commit	# File	# Reviewers
AVG	Entangled	151296	3	22	1
	Individual	2469	3	53	1
MAX	Entangled	4084987	78	583	28
	Individual	66173	6	1047	3
MIN	Entangled	13	1	1	0
	Individual	4	1	1	0

Additionally, we evaluate the difficulty of patch understanding and adoption from four perspectives: duration, the number of related commits, the number of modified files, and the number of assigned reviewers. As shown in Tables 8 and 9, entangled pull requests surpass individual ones in almost all

Table 9: Metadata of Unmerged Pull Requests in OpenSSL.

Metrics	Type	Duration (h)	# Commit	# File	# Reviewers
AVG	Entangled	35458	22	24	2
	Individual	20120	2	4	3
MAX	Entangled	217068	186	159	8
	Individual	165894	12	39	8
MIN	Entangled	<1	1	1	0
	Individual	<1	1	1	0

aspects, whether considering average, maximum, or minimum values. This further demonstrates that understanding an entangled patch requires more effort than individual patches.

B Patch Patterns

We summarize the patterns based on the grammar rules [5,17], *e.g.*, statements, identifiers, and keywords, and struct the pattern with a 4-tuple: (*anchor node*, *forward slicing termination criteria*, *backward slicing criteria*, *security indicator*). There are 12 security and three non-security patch patterns.

B.1 Security Patch Pattern

Add Security Check. This pattern focuses on newly added sanity checks to prevent security issues, *e.g.*, null pointer dereference, buffer overflow, uninitialized use, etc. A conditional statement checking critical values (*e.g.*, boundary values or return status) using specific keywords (*e.g.*, `NULL`) is identified as anchor node. Backward slicing determines where the variable within the condition statement is first introduced, and forward slicing traces until the end of the `if` statement code block. The security indicator is the existence of error-handling operations, *e.g.*, `NULL`, error report, or negative return status.

```
+    if (n < 0){
+       free(p);
+       return -1;
+    }
```

Listing 9: Instance of Add Security Check.

Update Security Check Conditions. This pattern describes the patches that aim to elevate the security level of the existing security check. The anchor nodes and security indicator are two conditional statements connected by a diff behavior edge and an attribute on the edge that shows more security checks are added. Similar to the previous pattern, the slicing procedure also finds the statements to define the checked variables and the scope of each conditional statement.

```
-    if (n < 0){
+    if (n =< 0){
```

Listing 10: Instance of Update Security Check Conditions.

Add Memory Deallocation Function. Memory deallocation functions refer to functions that free memory. This pattern describes the patches using such functions to prevent memory leakage, double free, or use after free. We consider the newly added memory deallocation functions (*e.g.*, function names containing `free`, `clean`, and `clear`) as anchor nodes and the security attribute indicator. Starting from the given node, we perform backward slicing via the data dependency to locate the statement of memory allocation, and forward slicing to locate the `NULL` assignment to the freed variable.

```
-    memcpy(dest, src, sizeof(src));
+    memcpy_s(dest, sizeof(dest), src, sizeof(src));
```

Listing 11: Update Memory Management Function Instance.

Update Memory Management Function. Security patches may update the memory management function for better resource allocation and release. The anchor nodes are the memory management function in the post-patch connected with update function diff behavior, also serving as the security attribute indicator.

Add Concurrency Control. To mitigate race conditions, patches usually enforce limits on access to the same memory area to ensure data consistency through lock-related functions (*e.g.*, `lock`) The anchor node and security indicator are the lock functions added in the post-patch. The backward and forward slicing aim to find the locked variable's definition and the corresponding `unlock` function, respectively.

Update Concurrency Control. Developers may introduce errors when managing multiple locks for concurrent access to the same resource. To resolve this, security patches adjust the execution locations of lock and unlock functions. In this context, the deleted lock functions serve as the anchor nodes and security attribute indicators. Forward slicing identifies the corresponding unlock functions with the locked variable.

Function Encapsulation. To reuse the security checks, developers encapsulate the existing checks with a new function. We capture this pattern with the deleted conditional checks as the anchor node and perform forward slicing to include the entire check. With the deleted conditional block, we perform backward slicing along the call relation to locate the newly added function call and check whether the deleted check has been redistributed to the function. The security attribute indicators align with the newly added security checks.

Initialize Variables. Uninitialized use occurs when a variable is declared but not initialized before being used. Security patches typically assign `NULL` to such a variable in the post-patch, which we can use as the anchor node. We take the initialization value of `NULL` as a security attribute indicator. No slicing is needed for this pattern.

Enlarge Buffer Size. One direct way to avoid buffer overflow is to enlarge the buffer size. The anchor node here is a pointer declaration statement in the pre-patch that connects to the pointer declaration statement in the post-patch. If the size is enlarged, *i.e.*, security indicator, we label it as a security segment candidate. No slicing is needed.

Update Return Value. The return value indicates the termination status of a function, and the value is to guide whether to continue the execution. When the return values are updated with negative values or macros with error keywords, we consider them as the anchor node and security indicators. We do not need to perform slicing for this pattern.

Enhance Error Handling. `goto` statements are adopted for

```
-       return NULL;
+       goto err;
+err:
+   free(p);
+   p = NULL;
+   return NULL;
```

Listing 12: Enhance Error Handling Instance.

```
-void func(param1, param2, ...){
+int func(param1, param2, ...){
+   int ret;
...
+   if(ret)
+      return ret;
...
+    return 0;
```

Listing 13: Update Return Type Instance.

code reuse, and are widely used for unified error handling with code reuse. We consider the modification that replaces the simple `return` with a more considerate error-handling process encapsulated in the `goto` as a security patch segment candidate. The anchor node is an added goto statement with an error-handling label. We perform backward slicing to locate the deleted return statement along diff behavior edge and forward slicing to contain the entire goto content. The indicator here is the returned error status.

Update Return Type. Return type corresponds with the function type. We consider changes to the return type from one type to another as candidates for security patch segments, as they pass different statuses to the caller function. Added return statements are treated as anchor nodes and security indicators. Backward slicing is performed to connect the value updates of the variable in the return statement.

B.2 Non-Security Patch Patterns

Debug. One way to get debug information is to print out error messages. Since the printing behaviors do not disrupt the program execution or change the value of its parameters, we consider these as non-security segments. The anchor node is the added debug function, *i.e.*, functions with `print`, `debug` or `log`. There is no security indicator associated with the debug statement and DISPATCH does not perform slicing.

Refactor. Refactoring restructures the code by transforming it into a cleaner and simpler design while maintaining the same variables and conditions, *e.g.*, changing multiple `if` to `switch`. The anchor nodes are added conditional statements that check the same condition but use different types of compound statements, *i.e.*, `if`, `switch`, `while`, and `for`. Forward slicing is performed to include the entire body of the conditional statements from both the pre-patch and post-patch versions, ensuring that no content changes have occurred.

New Feature. New features are always added by introducing new cases, functions, or files. There are two sub-patterns for the new feature. First, we directly treat the new files as a non-security patch segment. We identify the new function definition or new switch branches as an anchor node and perform forward slicing to include the entire function as a non-security patch segment. Second, variable declaration statements are also identified as anchor nodes, and forward slicing is performed along data dependencies to include all dependent statements as patch segments.

B.3 Pattern Extension

To add new patch patterns, developers only need to adhere to the specified patch pattern structure. Each new pattern is represented as a 4-tuple: (*anchor node*, *forward slicing termination criteria*, *backward slicing criteria*, *security indicator*). Anchor nodes are defined as a two-dimensional list, where the first dimension specifies the statement type, and the second dimension restricts statements based on keywords. Slicing termination criteria are represented as a three-dimensional list where the first dimension specifies the dependency type, which will be used to perform slicing, and the second and third dimensions specify the statement type and related restrictions. Finally, the security indicator is a two-dimensional list where the first dimension stores the indicator type, and the second dimension pinpoints the keywords. We will extend our patterns according to the structure in the future.

C Implementation

DISPATCH is implemented with Python scripts. Given a version diff, OSS patch, or commit, we first clone the corresponding repository using Git and roll back to retrieve the pre-patch and post-patch versions of the source code. We then use tree-sitter [11] to generate the ASTs for the modified files and construct the CDG, DDG, and CG. Next, we analyze the diff file hunk by hunk to extract diff behaviors. For each statement in a hunk, we compare the AST subtrees. If the statements are of the same type and share predefined commonalities, we record the changes. Control dependencies are updated if both nodes on an edge are modified, and data dependencies are updated if a variable is changed. We then integrate the diff behaviors, CDG, DDG, and CG into a unified graph representation, referred to as PatchGraph. Then, we traverse the nodes sequentially, select anchor nodes by matching the type and fuzzy matching the keywords, and perform constrained slicing while marking visited edges during the statement-level analysis. Unvisited edges are further analyzed to decide whether to keep or discard them based on the pre-defined rules. If the unvisited edge has been kept, the related patch segments are combined. The merged patch segments are further merged if they are similar by calculating the Levenshtein distance. This process concludes with the individual patch generation.

D OpenSSL Data Description

Table 10 lists the metadata of version diffs of OpenSSL 1.1.1, including the number of modified C/C++ files, related commits in C/C++, and lines of the version diff.

Table 10: Metadata of version diffs in OpenSSL 1.1.1.

Version	# File	# Commit	# Lines of Diff
0-a	111	124	3,741
a-b	143	120	10,496
b-c	106	83	15,645
c-d	186	125	6,050
d-e	716	128	8,994
e-f	19	9	623
f-g	31	20	2,550
g-h	98	79	4,153
h-i	28	28	916
i-j	36	28	843
j-k	22	20	337
k-l	78	80	2,736
l-m	59	58	1,441
m-n	35	29	954
n-o	24	24	531
o-p	21	19	712
p-q	6	8	189
q-r	34	29	1,275
r-s	4	6	206
s-t	33	22	1,782
t-u	20	18	1,198
u-v	6	8	103
v-w	9	8	161

E Unravelment Results on Security Patches Annotated by CVEs

In the NVD, there are 24 CVEs related to OpenSSL 1.1.1 that contain corresponding security patches in C/C++, as annotated in the CVE records. Among them, DISPATCH successfully decomposed 21 of them, yielding the 87.5% accuracy.

Table 11 shows the detailed results. We analyze the cases where DISPATCH fails to unravel and showcase the reasons. DISPATCH split the patch for CVE-2023-0464 and CVE-2019-1563 into isolated patch segments and fails to merge them. Listing 14 and 15 shows the simplified patch for CVE-2023-0464 [3], which has been fixed by adding the `node_maximum` and `extra_data` to constrain the validation for `X509` certificate. However, DISPATCH split it into two separate patches that add the two variables respectively, failing to merge them because the parameter `tree` exists in the pre-patch and the post-patch and DISPATCH treat it as unchanged and neglect its dependencies. Besides, DISPATCH fails to decompose the patch for CVE-2021-3449 [1], as shown in Listing 16, where the first hunk is the official patch. DISPATCH merges the two hunks since they perform the same change with a 100% similarity ratio.

```
diff --git a/.../pcy_local.h b/.../pcy_local.h
@@ ... @@ struct X509_POLICY_LEVEL_st {
+    size_t node_maximum;
diff --git a/.../pcy_node.c b/.../pcy_node.c
+    if (tree->node_maximum > 0 ...)
+        return NULL;
```

Listing 14: 1st patch for CVE-2023-0464.

Table 11: Unravelment Results on Security Patches Annotated by CVEs [4] in OpenSSL 1.1.1.

CVE ID	Patched Version	Result
CVE-2023-3817	OpenSSL1.1.1v	Yes
CVE-2023-3446	OpenSSL1.1.1v	Yes
CVE-2023-2650	OpenSSL1.1.1u	Yes
CVE-2023-0465	OpenSSL1.1.1u	Yes
CVE-2023-0464	OpenSSL1.1.1u	No
CVE-2023-0286	OpenSSL1.1.1t	Yes
CVE-2023-0215	OpenSSL1.1.1t	Yes
CVE-2022-4450	OpenSSL1.1.1t	Yes
CVE-2022-4304	OpenSSL1.1.1t	Yes
CVE-2022-0778	OpenSSL1.1.1n	Yes
CVE-2021-3712	OpenSSL1.1.1l	Yes
CVE-2021-3711	OpenSSL1.1.1l	Yes
CVE-2021-3450	OpenSSL1.1.1k	Yes
CVE-2021-3449	OpenSSL1.1.1k	No
CVE-2021-23841	OpenSSL1.1.1j	Yes
CVE-2021-23840	OpenSSL1.1.1j	Yes
CVE-2020-1971	OpenSSL1.1.1i	Yes
CVE-2020-1967	OpenSSL1.1.1g	Yes
CVE-2019-1563	OpenSSL1.1.1d	No
CVE-2019-1549	OpenSSL1.1.1d	Yes
CVE-2019-1547	OpenSSL1.1.1d	Yes
CVE-2019-1543	OpenSSL1.1.1c	Yes
CVE-2018-0734	OpenSSL1.1.1a	Yes
CVE-2018-0735	OpenSSL1.1.1a	Yes

```
diff --git a/.../pcy_node.c b/.../pcy_node.c
X509_POLICY_NODE *level_add_node(X509_POLICY_LEVEL *level,
     X509_POLICY_DATA *data,
     X509_POLICY_NODE *parent,
-    X509_POLICY_TREE *tree)
+    X509_POLICY_TREE *tree,
+    int extra_data)
diff --git a/.../pcy_tree.c b/.../pcy_tree.c
-    if (level_add_node(..., tree) == NULL) {
+    if (level_add_node(...., tree, 1) == NULL) {
```

Listing 15: 2nd patch for CVE-2023-0464.

F Baseline Supplement Experiments

Table 12 presents the comparison of patch unraveling accuracy among SPatch, GPT-4, UTANGO, and DISPATCH on Nginx, the Linux Kernel, and ImageMagick. The results are consistent with those observed for OpenSSL. Due to significant modifications in macro values and definitions, which SPatch and UTANGO fail to handle properly, their results are adversely affected.

```
diff --git a/.../extensions.c b/.../extensions.c
+ s->s3->tmp.peer_sigalgslen = 0;
diff --git a/.../extensions_clnt.c b/.../extensions_clnt.c
+ s->s3->tmp.peer_sigalgslen = 0;
```

Listing 16: Patch failing to be unraveled by DISPATCH for CVE-2021-3449.

G DISPATCH Scalability

As the number of individual patches in one entangled patch increases from 3 to 122, shown in Figure 6, DISPATCH main-

Table 12: Comparison with baseline approaches on Nginx, Linux Kernel, and ImageMagick.

Application	**Groundtruth**	**SPatch**			**GPT-4**			**UTANGO**			**DisPatch**		
	# IP	**# GP**	**# EMIP**	**Accuracy**	**# GP**	**# EMIP**	**Accuracy**	**# GP**	**# EMIP**	**Accuracy**	**# GP**	**# EMIP**	**Accuracy**
Nginx	26	47	13	50%	6	5	19%	192	15	58%	29	24	92%
Linux Kernel	114	205	62	54%	7	3	3%	208	58	51%	118	113	99%
ImageMagick	50	420	13	26%	5	0	0%	429	14	28%	58	48	96%
Total	190	672	88	43.3%	18	8	7.3%	829	87	45.7%	205	185	95%

tains good performance consistently.

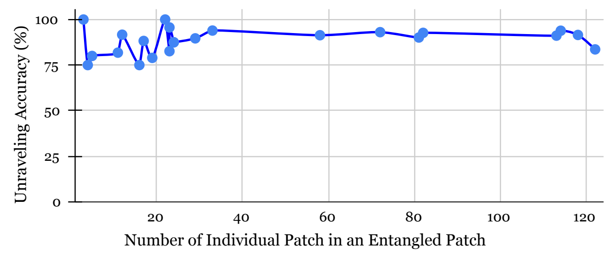


Figure 6: Accuracy on different sizes of entangled patch.