VERIBIN: Adaptive Verification of Patches at the Binary Level

Hongwei Wu*, Jianliang Wu†, Ruoyu Wu*, Ayushi Sharma*, Aravind Machiry* and Antonio Bianchi*
*Purdue University
{wu1685, wu1377, sharm616, amachiry, antoniob}@purdue.edu
†Simon Fraser University
wujl@sfu.ca

***Abstract*—Vendors are often provided with updated versions of a piece of software, fixing known security issues. However, the inability to have any guarantee that the provided patched software does not break the functionality of its original version often hinders patch deployment. This issue is particularly severe when the patched software is only provided in its compiled binary form. In this case, manual analysis of the patch's source code is impossible, and existing automated patch analysis techniques, which rely on source code, are not applicable. Even when the source code is accessible, the necessity of binary-level patch verification is still crucial, as highlighted by the recent XZ Utils backdoor.**

To tackle this issue, we propose VERIBIN, a system able to compare a binary with its patched version and determine whether the patch is "Safe to Apply", meaning it does not introduce any modification that could potentially break the functionality of the original binary. To achieve this goal, VERIBIN checks functional equivalence between the original and patched binaries. In particular, VERIBIN first uses symbolic execution to systematically identify patch-introduced modifications. Then, it checks if the detected patch-introduced modifications respect specific properties that guarantee they will not break the original binary's functionality. To work without source code, VERIBIN's design solves several challenges related to the absence of semantic information (removed during the compilation process) about the analyzed code and the complexity of symbolically executing large functions precisely. Our evaluation of VERIBIN on a dataset of 86 samples shows that it achieves an accuracy of 93.0% with no false positives, requiring only minimal analyst input. Additionally, we showcase how VERIBIN can be used to detect the recently discovered XZ Utils backdoor.

I. INTRODUCTION

Timely creation and deployment of security patches is essential to protect programs against corresponding security vulnerabilities [59]. However, studies [28], [47] show that vendors take significant time to apply these security patches. From the vendor perspective, applying patches involves a risk of regression and breaking the existing program functionality [24], [89]. Such regressions are expensive, especially when affecting mission-critical embedded systems such as aircraft and power grids. Consequently, vendors either spend considerable time testing the patched version of a program to ensure that it does not contain any regressions [12], [15], [25], [31], or they may decide not to patch a system even if it is affected by known vulnerabilities. This friction in verifying patches results in a "patch gap" [55], [65], which is detrimental to the security of the unpatched system, especially if the underlying vulnerability is publicly known [48].

To handle the tension between the risk of regression and the patch gap, we need techniques to verify that a given patch neither affects the program's functionality nor causes regression and, consequently, is safe to apply. In this context, "safe" refers to the preservation of functionality, rather than addressing whether the code is affected by vulnerabilities. Existing patch verification techniques, such as SPIDER [51] and SID [79], require source code patches. However, when the code running on a device is provided by a third party, such source code patches are usually unavailable to vendors, who only receive patched binaries. Furthermore, in old legacy systems, where the exact source code version is lost, or the process for building the software from source code is not documented – binary-level patching is the only option [8]. Recently, researchers have proposed many techniques [26], [32], [65], [71], [82], [84], [87] to directly patch the program binaries, and a few companies, such as 0Patch [9], offer binary-level patches for unmaintained software. Obviously, patches generated in this way cannot be verified using existing, source-code-based, patch verification approaches. As demonstrated by the recent XZ Utils backdoor [10], it is feasible to circumvent source-level verification techniques by injecting malicious code directly into the binary through a modified compilation pipeline. This demonstrates the critical need for and importance of binary-level patch verification, even when the source code is accessible. This necessity has also been recently highlighted by the DARPA AMP program [8], which emphasizes the importance of binary patch verification and tasked several researchers and organizations to develop such techniques.

Previous work (*i.e.,* SPIDER [51]) has considered a patch as "safe" if it only restricts a function's accepted input space without affecting its behavior for valid inputs, and it has formally defined source-code-level properties that need to hold for a patch to be considered as functionality-preserving. *Our goal is to aid the patch verification process by identifying, highlighting, and verifying patch-introduced changes directly in binaries. We aim to verify analogous properties, which we call Safe to Apply (StA) properties, without the need for source code, by directly comparing a patched binary with its original version.* With the term "safe", our definition of StA emphasizes maintaining the program's original functionality, ensuring that the patch does

Network and Distributed System Security (NDSS) Symposium 2025
23 - 28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.23359
www.ndss-symposium.org

not introduce regressions or alter intended behavior. It does not, however, guarantee that the patch addresses the underlying vulnerability. Evaluating semantic equivalence to determine whether patches preserve the original functionality has been previously explored by several source-level approaches [17], [46], [51], [52], [78]. However, as we will show, the verification of these properties at the binary level poses various challenges that require our solution to differ significantly from source-code-based solutions. Additionally, we make our verification "adaptive", enabling analysts to assist in cases where automated verification is infeasible since it requires domain-specific knowledge.

More precisely, given a binary patch (*i.e.,* the original and patched binaries) and the patch-affected function addresses, we use under-constrained symbolic execution (Section V-B) to explore all reachable paths in the affected functions. Then we verify StA properties for corresponding path pairs in the original and patched functions. This path-based verification also enables us to be adaptive and use additional information an analyst can potentially provide (Section V-D). Specifically, if we are unable to verify certain properties automatically (*e.g.,* because it requires semantic reasoning), we perform introspection of the corresponding symbolic constraint to identify the root cause. This will be translated into a query on the patch that an analyst can answer. We use the analyst's response to simplify symbolic constraints and verify the corresponding property. Our design choices (Section V) aim to produce a high-assurance system with zero false positives, *such that any patch considered StA by* VERIBIN *is always safe to apply*.

We evaluate VERIBIN on three datasets with a total of 86 pairs of original and patched binaries whose sizes vary from 5KB to 120MB. Our evaluation shows that VERIBIN achieves 93.0% accuracy (*i.e.,* the number of correctly identified patches over the total number of patches) with an average of 1,293.8 seconds of runtime and does not generate any false positives.

In summary, we make the following contributions:

- We design VERIBIN, the first system capable of describing patch behavior at the binary level and determining whether the patched binary can be safely deployed (*i.e.,* it is functionality preserving). To this aim, we formalize the definition of Safe to Apply (StA) patches for binaries and define StA properties.
- We design an efficient, path-based technique to verify StA properties. Our techniques reduce the analysis time by 75% compared to the standard approach.
- We also make our analysis adaptive so that analysts can augment VERIBIN's analysis by providing domain-specific information, which is not available in the patched binary.
- We evaluate VERIBIN on 86 pairs of binaries coming from three different datasets. Our results show that VERIBIN can delineate patch behaviors with 93.0% accuracy and no false positives. Additionally, we showcase how VERIBIN can be used to detect the recently discovered XZ Utils backdoor.

To foster further research in this field, we make VERIBIN publicly available [11].

II. BACKGROUND

In principle, a patch to a binary is *Safe to Apply* (StA) if it does not break the existing functionality. However, every patch, unless empty, impacts the binary's functionality in some way. Nevertheless, certain patches, particularly security patches [47], [64], only restrict the input space by adding validation checks to reject invalid inputs, while preserving the program's behavior for valid inputs. As explained in previous work [51], such input-restricting patches are StA. Therefore, at the high level, we aim to verify that *a given patch to a binary is StA by ensuring that it restricts inputs and does not affect the program's behavior for valid inputs*. Here, "safe" specifically refers to maintaining the integrity of the program's original functionality, instead of focusing on whether the patch addresses the underlying vulnerability.

A. Terminology

As mentioned before, our techniques will work at the binary level and have access only to the original and the patched binary. We will use the patch in Listing 1 as our running example and x86-64 as the target architecture to explain our techniques. Figure 1 shows the assembly level Control Flow Graph (CFG) of the corresponding original and patched functions.

Original and Patched entities. We will use o and p to indicate original and patched entities. For instance, for a binary B, B_o and B_p represent the original and patched binary, respectively. Similarly, F_o and F_p indicate the original and patched functions of a function F, respectively.

Valid Exit Path (VEP) and Error-handling Exit Path (EEP). We assume the CFG of a function has a single entry (*i.e.,* no predecessors) and multiple exit Basic Blocks (BBs) (*i.e.,* no successors). A *complete path* in a CFG starts at an entry BB and ends at an exit BB. For instance, all the paths shown in Figure 1 are complete paths. We consider all of a function's arguments as its "inputs". Within a function, a Valid Exit Path (VEP) is a complete path that the function execution takes only on valid inputs. In other words, an input that follows a VEP is a valid input, such as P_{o2} and P_{p3} in Figure 1. Similarly, an Error-handling Exit Path (EEP) is a complete path where inputs that follow this path are considered invalid and are thus rejected by the function, such as P_{o1}, P_{p1}, and P_{p2}.

Path Constraint (PC). A path constraint (PC) is a condition that must be met by a function's input for the execution to follow a specific path. For example, the path constraint of P_{o1} is `edi > 0x3ff`. We denote the path constraint of a path k as C_k.

B. Identifying StA Patches

We start from previous work [51] to define a Safe to Apply (StA) patch. Specifically, a patch is StA if it satisfies two properties (StA properties): (i) It does not increase the valid input space of the binary (*non-increasing input space*), and (ii) The output of the patched binary remains the same as the original for all the valid inputs (*output equivalence*). The first property ensures that the patched program does not accept any new input values, preserving the original input constraints. The second property guarantees that the output of the patched program will be identical to the original program for all valid inputs, maintaining consistent functionality.

Given the original (B_o) and patched (B_p) binaries, our goal is to verify that the patch applied to B_p is a StA patch by verifying StA properties. We perform this verification at the function level. Notably, we start by identifying the set of functions modified by the patch and check that the changes made to all these functions satisfy StA properties. A patch is considered Safe to Apply for F_o and F_p if it satisfies the StA properties. Further, a patch is considered StA for B_o and B_p if

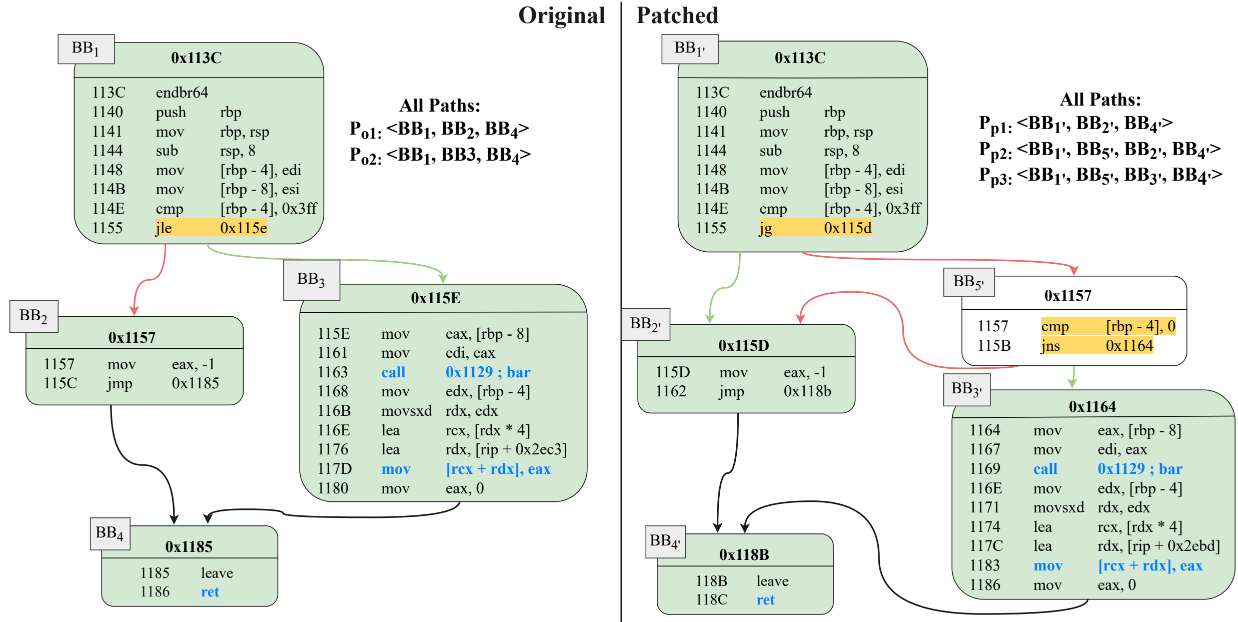

Fig. 1: The Control Flow Graph (CFG) of the original and patched function in Listing 1. The instructions marked with blue denote the output of the function. The green basic blocks are matched blocks while the white basic block indicates the unmatched block. If a basic block has multiple out-going edges, we use the green and red edges to represent the true and false branches, respectively.

```
1   #define MAX_SIZE 1024
2   int data[1024];
3   int foo(int a, int b){
4 -     if(a >= MAX_SIZE){
5 +     if(a >= MAX_SIZE || a < 0){
6           return -1; }
7       data[a] = bar(b);
8       return 0; }
```

Listing 1: Running example

it is StA for all the modified functions. Specifically, given an original function (F_o) and its corresponding patched function (F_p), we verify the StA properties as described below:

Non-Increasing Input Space (P1). We check that the patch does not increase the valid input space, *i.e.,* all valid inputs to F_p are also valid inputs to F_o. In other words, an input that executes a VEP in F_p should also execute a VEP in F_o.

Output Equivalence. We check that, for all valid inputs, the output of F_p is the same as that of F_o. In line with existing work [51], we define the "output" of a function as all its externally visible effects, *i.e.,* its return value, writes to non-local memory regions, and function calls along with arguments.

We limit our output equivalence analysis only to valid inputs. More precisely, for a given valid input i, we use VEP_o^i and VEP_p^i to denote the VEP executed by i in F_o and F_p, respectively. We then check that for all valid inputs, the output produced in VEP_o^i is the same as that of the corresponding VEP_p^i. Specifically:

- **Non-local Memory Writes Equivalence (P2)**: All non-local memory writes operations in VEP_p^i of F_p should write the same value to the same memory region as in the corresponding VEP_o^i of F_o.
- **Return Value Equivalence (P3)**: The return value of F_p along VEP_p^i should be the same as the return value of F_o along VEP_o^i.
- **Function Call Equivalence (P4)**: The function calls made along VEP_p^i in F_p should be equivalent to the function calls made along VEP_o^i in F_o, *i.e.,* the same function is called with the same arguments.

III. Challenges and Solutions

Although previous work [51] demonstrates techniques for verifying StA properties, applying them at the binary level is challenging due to the absence of source-level abstractions such as variable names, data types, and labels. For instance, SPIDER creates a symbolic variable for each L-value expression (*e.g.,* `p->fld`), but at the binary level, these expressions translate into multiple dependent load instructions, resulting in complex symbolic expressions. Beyond the engineering challenges of dealing with binaries [56], we identified three main technical challenges, which we address in this section along with our proposed solutions.

A. Finding Error-handling Exit Paths (EEPs)

Accurately identifying EEPs is essential for verifying our properties. Most existing work [44], [50], [51], [63], [75] relies on source code and leverages certain common error-handling patterns easily detectable at the source-level, but invisible in binaries. For example, CheQ [50] and EECatch [63] use application-specific error-handling functions (*e.g.,* `BUG`, `WARN` in the Linux kernel) to identify error-handling blocks and consider all paths reaching them as EEPs. However, function names are often unavailable in binaries, requiring additional strategies. SPIDER [51] uses `goto` error labels [7] (*e.g.,* `goto fatal:`) to mark error blocks, but these labels are absent in binaries. Apex [44] and ErrDoc [75] use disjoint termination as error markers. For instance, in Listing 1, the error path (*i.e.,* `if(..) return -1`) and valid path (*i.e.,* `return 0;`) terminate at different source locations (*i.e.,* Line 6 vs. 8). However, compilers usually optimize return

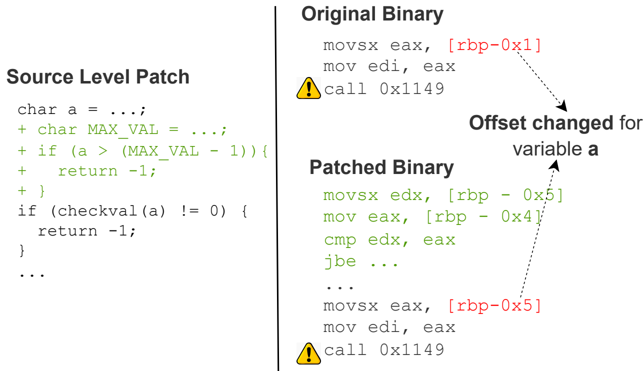

Fig. 2: Patch snippet demonstrating difficulty in verifying P4 at the binary level.

points in binaries, merging them into a single location, as shown in the CFGs of the corresponding original (at `0x1186`) and patched (at `0x118C`) binary in Figure 1. Pattern-based approaches are challenging in binaries since compilers can set error values in multiple, semantically identical ways.

Our Solution: Instead of identifying error markers, we detect EEPs directly from all the reachable execution paths in the function's CFG. We symbolically execute each path, collecting symbolic values of all operands in each expression. This precise modeling of instruction semantics allows us to verify whether a given path is an EEP. Using a combination of static analysis techniques and heuristics, we identify EEPs from the symbolic state of each path (Section V-C).

B. Handling Complex Symbolic Constraints

Verifying StA properties requires comparing path constraints and symbolic values. For instance, to verify P1, SPIDER [51] disjunctively combines (*i.e.,* logic OR) the PCs of all VEPs in F_p and F_o, then checks that the combined PC of F_p implies the combined PC of F_o. The lack of data types complicates the path constraints by reducing all constraints to bit-vectors instead of using corresponding theories. For instance, in Listing 1, we could use the integer theory [69] of Satisfiability Modulo Theory (SMT) solvers [58] to represent the constraint `a >= MAX_SIZE` as the variable `a` is of type `int`. However, in the corresponding binary (Figure 1), type information is lost, requiring the use of bit-vector theory, which is less efficient [60].

The absence of variable names and other identifiers complicates the combination and comparison of symbolic values between B_o and B_p. Variables in binaries are typically identified by their offsets, which may differ between B_o and B_p. This offset change can hinder the verification of StA properties, leading to imprecise results. Consider the example in Figure 2, the offset of the variable `a` changed from `[rbp-0x1]` in F_o to `[rbp-0x5]` in F_p, potentially causing the function call `call 0x1149` to be incorrectly flagged as inequivalent (⚠) due to differing arguments, leading to an incorrect conclusion that P4 does not hold.

Our Solution: We noticed that the issue of complex constraints arises when combining PCs of different VEPs. To address this, we avoid combining PCs when possible. Instead, we find input equivalent VEPs or Matching Path Pairs between F_o and F_p (Section V-D2). We define Matching Path Pair (MPP) as a pair of VEPs, (*i.e.,* VEP_p^i and VEP_o^i), such that any input i, if it executes VEP_p^i in F_p then it executes VEP_o^i in F_o. Identifying MPPs simplifies constraints and the verification of StA properties. For instance, P1 holds if all VEPs of F_o and F_p are part of MPPs. Similarly, we can verify P2-P4 individually for each MPP. If we cannot find MPPs, we fall back to combining all VEPs. We use the terms *Divisible* and *Indivisible* to distinguish between the cases of verification through MPPs or through combining all VEPs, respectively.

We address Compiler-Introduced Offset Changes (CIOCs) using heuristic-based techniques, including content equivalence checking, shift-by-same-offset checking, and structural position checking (details can be found in Section V-D1).

C. Handling Semantically Equivalent Changes

Patches often replace function calls with other equivalent function calls or add calls to logging functions. For instance, the patch in Listing 2 replaces the `htmlNodeDumpFormatOutput` function with the `xmlNodeDump` functions. Although these functions are semantically equivalent, such patches do not strictly satisfy all StA properties (*e.g.,* adding additional function calls violates P4). Under the assumption that these functions are equivalent, the patch is StA. Existing work, such as SPIDER, handles this by pattern matching and maintaining a list of semantically irrelevant function names and source code patterns. However, defining such patterns at the binary level is challenging due to differences in Instruction Set Architecture (ISA) and the absence of function names.

```
1 - htmlNodeDumpFormatOutput(buf, docp, node, 0, format);
2 + xmlNodeDump(buf, docp, node, 0, format);
3   mem = (xmlChar*) xmlBufferContent(buf);
4   if (!mem) { RETVAL_FALSE; ...
```

Listing 2: Example of a real-world patch that replaces a function call with an equivalent one

Our Solution: Based on our experience, while these semantics-preserving changes are difficult to verify automatically, a human analyst can quickly recognize them. Hence, we use an *adaptive verification technique* to handle this (Section V-D). When an StA property fails, we introspect the corresponding symbolic constraint to identify the root cause of the failure. We then convert the root cause into a query for the analyst, highlighting the difference between the two symbolic expressions related to the patch. We use the analyst's response to simplify symbolic constraints and verify the corresponding property.

IV. Overview

This section presents an overview of VeriBin, a system to verify whether a given patch is Safe to Apply (StA), starting from a patched binary and its original version.

Design Considerations. VeriBin focuses on patches modifying existing functions, excluding those that add or remove an entire function. This design choice aligns with the predominant nature of security patches, which typically modify existing functions [47], [51], [64]. Moreover, verifying patches that add or remove functions requires whole program analysis and thus does not scale for real-world programs. Our system is designed to function effectively with binaries, regardless of symbol availability. Although not essential for core functions, the presence of debug symbols enhances the accuracy of specific tasks such as function prototype extraction

and Error-handling Exit Path (EEP) detection. Additionally, we conservatively interpret SMT solver failures as implication or equivalence failures, leading VERIBIN to classify patches as not StA. Figure 3 illustrates the framework of VERIBIN.

Preprocessing. Given binaries B_o, B_p, and a pair of patch-affected function addresses, the preprocessing step (Section V-A) uses existing binary analysis tools to extract essential information for patch verification. This includes CFGs of the affected functions F_o and F_p, function prototypes (*i.e.,* number and type of arguments) for all the function calls within F_o and F_p, and matching BBs' addresses. The extracted data is automatically stored in a configuration file. For the example in Listing 1, this step retrieves the matching BBs' addresses and function prototypes for `foo` and `bar`, adding this information to the configuration file. Meanwhile, this step reconstructs the CFGs of the F_o and F_p as illustrated in Figure 1.

Symbolic Execution. For F_o and F_p, symbolic representations of each execution path are collected using under-constrained symbolic execution. This process employs fresh symbols for function arguments and global variables, symbolically executing from the function entry point and recording the state at each instruction. In the context of Figure 1, all paths (*i.e.,* P_{o1}, P_{o2}, P_{p1}, P_{p2} and P_{p3}) are converted into their symbolic representation, and their PCs are computed.

Identifying VEPs and MPPs. EEPs are identified using a combination of static analysis and heuristics (Section V-C), with the remaining paths classified as VEPs. In the example, P_{o1} in F_o, and P_{p1} and P_{p2} in F_p are identified as EEPs, while P_{o2} (in F_o) and P_{p3} (in F_p) are identified as VEPs. From VEPs of F_o and F_p, we identify Matching Path Pair (MPP) (Section V-D2). As mentioned before in Section III-B, an MPP is a pair of VEPs, (VEP_p^i, VEP_o^i), where any input i executing VEP_p^i in F_p also executes VEP_o^i in F_o. In the example from Figure 1, only one MPP is identified: (P_{p3}, P_{o2}).

Verifying StA Properties. Given the identified MPPs, the StA properties are verified as follows: P1 verification (Section V-D3) ensure all VEPs of F_p are part of an MPP, indicating that all valid inputs to F_p are also valid for F_o. In P2 verification (Section V-D3), for each MPP ($\text{VEP}p^i$, $\text{VEP}o^i$), we identify and symbolically check the equivalence of non-local memory writes. In the example, non-local writes at `0x1183` in P_{p3} and `0x117d` in P_{o2} are compared, with identical symbolic values satisfying P2. P3 verification (Section V-D3) involves symbolically comparing return values along paths in each MPP. For the example, this verifies that the return value (`0`) is identical in P_{p3} and P_{o2}. P4 verification (Section V-D3) extracts and compares function calls in each MPP, ensuring they target the same function with identical symbolic argument values. In the example, function calls at `0x1169` in P_{p3} and `0x1163` in P_{o2} are compared, both targeting `0x1129` with the same arguments (content at `rbp-8`, which is `esi`), thus satisfying P4. The patch is considered StA if all four properties hold.

Handling Missing MPPs (*i.e.,* Indivisible Case). When Matching Path Pairs (MPPs) cannot be identified for certain patches, all Valid Exit Paths (VEPs) and their Path Constraints (PCs) are combined to verify StA properties. For P1 verification, the PCs of all VEPs in F_p and F_o are disjunctively combined to obtain C_{VEP_p} and C_{VEP_o}. The implication $C_{VEP_p} \rightarrow C_{VEP_o}$ is then checked, ensuring that any input executing a VEP in F_p also executes a VEP in F_o. Similar approaches are applied to verify P2, P3, and P4 in this indivisible case.

Adaptive Verification. We resort to adaptive verification (Section V-D) through analyst assistance when automated verification fails. This process involves identifying the failure's root cause, converting it to a textual query about the patch, and consulting the analyst. The analyst's response is then used to simplify symbolic constraints and re-verify the failed property. The identification of MPPs facilitates this adaptive verification, enabling human-interpretable queries on program properties (*e.g.,* "Is changing the 6th argument of `foo` from 20 to 10 considered StA?") rather than presenting analysts with complex symbolic constraints to solve.

V. DESIGN

Figure 3 shows the overall design of VERIBIN. VERIBIN comprises four main computational steps, which we explain in the rest of this section.

A. Preprocessing

VERIBIN takes, as input, the original binary (B_o) and the patched binary (B_p), together with the address of F_o (the original function in B_o) and the address of F_p (its patched version in B_p)[1]. During the preprocessing step, we analyze B_o and B_p to collect necessary information for verification.

(1) Callee (or Called) Functions' information: We gather information about all callee functions in F_o and F_p. Specifically, we collect function prototypes and the matching callee function addresses between B_o and B_p. Within F_o and F_p, we record the target address of each `call` instruction. Using a binary decompiler [5], we extract the callee function prototypes and fetch matching function addresses from the perfectly matched function addresses generated via binary diffing. To enhance the accuracy of function prototypes, we perform backward decompilation of the callee functions called within the function being analyzed. Specifically, before decompling a function, we first decompile all its callee functions to provide additional context for more accurate prototype recovery.

(2) Matching Basic Blocks addresses: We gather the addresses of matching basic blocks in F_o and F_p via binary diffing.

(3) Control Flow Graph (CFG): We concstruct the CFGs for F_o and F_p.

The information gathered during preprocessing (except for the CFGs) is stored in a configuration file for later use. Optionally, an analyst can edit and correct this information if the preprocessing step is inaccurate (*e.g.,* an analyst can provide accurate argument information for uncommon variadic functions).

B. Symbolic Execution

We symbolically execute F_o and F_p from their entry points, collecting symbolic expressions of states along each execution path.

[1]Analysts can obtain the addresses of the patch-affected functions by performing function-level binary diffing using existing tools. In a side experiment, we found that in 81.4% of our dataset of 86 binaries, no human effort was required, as the patch-affected function addresses exactly matched those identified by BinDiff (*i.e.,* not-perfectly-matched functions). Yet, we consider the process of fully automating the identification of the patched functions orthogonal to our effort.

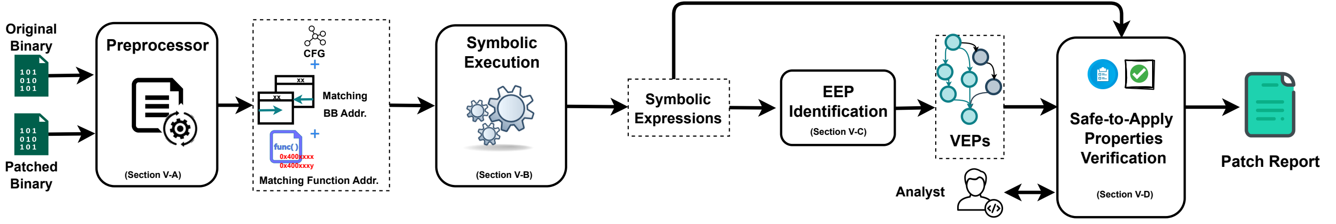

Fig. 3: Overall design of VERIBIN.

We handle loops by allowing each block to be visited at most twice, and infeasible paths are avoided through proactive pruning. Precisely, at each conditional jump (*i.e.,* `IfThenExpr` followed by a `JumpExpr`), a new path is explored only if the corresponding path constraint is satisfiable.

For inter-procedure function calls, we generate a symbolic expression that includes the function name and symbolic arguments, assigning this expression to the return register as the function's return value. Non-constant pointer arguments are treated as output arguments, meaning they can be modified to carry new data as a result of the function's execution. This is often the case with pointer arguments, for which the content stored at the addresses they point to may be altered by the called function. Therefore, we update the content stored at the memory locations pointed by these output arguments similarly to the return value of the function. As an example, in the code `foo(&a,b)`, `a` is considered as an output argument, and the memory is updated by storing the value of `foo(&a,b)` at the location `a`. Note that `foo` is not symbolically executed and is treated as an uninterpreted function [35].

At the end of this process, we have all feasible execution paths from the function's starting basic block to its exiting basic blocks. For each execution path, we track the following information: *(1) Path Constraint:* The input conditions that must be satisfied for the execution path to be followed. *(2) Symbolic Register Values:* A dictionary mapping register names to their symbolic values. *(3) Symbolic Memory:* A dictionary mapping symbolic addresses to the symbolic content stored at those addresses. *(4) Function Calls and Their Arguments:* A dictionary correlating each function call site with the symbolic expressions corresponding to the callee function's arguments.

C. EEPs Identification

As discussed in Section IV, our analysis requires identify Error-handling Exit Paths (EEPs) to verify StA properties. We identify EEPs at the path level, considering an execution path from Section V-B to be an error-handling exit path if it satisfies any of the following heuristic rules:

Heuristic A: Call to Error-Handling Functions. Error-handling functions, such as `exit`, are commonly used to handle unrecoverable errors and terminate the execution of the program, usually indicating an EEP. We maintain a list of known library error-handling functions (*e.g.,* `exit`, `__assert_fail`, `abort`), and check if the path contains calls to any function in this list. For statically linked binaries, we use FLIRT [4] to recover function names, ensuring these library error-handling functions are identifiable even in stripped binaries. Additionally, we utilize the 'noreturn' information (*i.e.,* functions that do not return to the caller, typically used for error handling or program termination) from the function prototypes recovered using the binary decompiler, adding those functions to our error-handling function list. User-defined error-handling function names can also be specified in the configuration file.

Heuristic B: The return value refers to an invalid return value. A path is considered an EEP if its return value is identified as invalid. We collect all unique return values from the feasible paths within the function. If the unique values are `0` and `1`, the value associated with a shorter average path length is considered invalid. For more than two unique values, negative return values are typically considered invalid. This heuristic is based on the observation that most error-handling functions use a negative one (`-1`) or boolean values (`1/0`) as the return value to indicate errors [50], [63]. To accommodate project-specific error codes, we also incorporate invalid return values specified by the analyst.

Heuristic C: The path's length is relatively short. A path is considered an EEP if its length is relatively short compared to other paths in the CFG. This heuristic is based on the observation that patches adding checks for invalid inputs often lead to quick termination of one branch, while the other branch leads to the main function body. We consider a path as an EEP if the path length is less than a specific ratio (determined to be 0.8) of the average length of all the paths in the CFG, as detailed in Appendix B.

D. StA Properties Verification

This subsection details the verification of the four properties discussed in Section II-B

1) Handling Compiler-Introduced Offset Changes: In binary analysis, memory allocation to different addresses post re-compilation is a common occurrence, often a result of compiler optimizations. While these addresses are different in F_o and F_p, they represent the same content (*e.g.,* the same variable in the source code). We name these changes as *compiler-introduced offset changes* (CIOCs). The following techniques are used to handle CIOCs:

Content-Based Comparison. When symbolic expressions differ solely due to memory loads from distinct fixed addresses, we verify the equivalence by comparing the contents at these addresses in both B_o and B_p. This is particularly relevant for global read-only constants, like constant strings, relocated by the compiler.

Offset Analysis. Based on our observation, when a compiler changes the offset of local variables, it retains the relative position of the variables. In other words, all variables shift by the same offset. Given two symbolic expressions from F_o and F_p, we collect all the unmatched local variable addresses

in these symbolic expressions. If these addresses differ by a consistent offset, we attribute this to a CIOC, treating the variables at these addresses as equivalent.

Structural Position Correlation. Based on our observation, symbolic expressions appearing in structurally similar positions of two ASTs typically represent the same variable. Hence, when symbolic expressions appearing in corresponding structural positions of two ASTs only differ by an offset, we consider these two symbolic expressions equivalent.

We add these offset mappings as assumptions in our patch report, enabling an analyst to verify whether these assumptions hold. A case study in Section VIII demonstrates the necessity of addressing CIOCs.

2) Finding Matching Path Pairs: As described in Section III-B, we prefer to use MPPs to verify StA properties, thereby avoiding complex symbolic constraints and improving VERIBIN's performance. To find MPPs, we first obtain all feasible paths in F_o and F_p (Section V-B). We then remove EEPs from these paths (Section V-C), maintaining a list of remaining valid exit paths (VEPs), denoted as VEP_o and VEP_p, respectively. An MPP is a tuple, $\langle p,o \rangle$, where $p \in VEP_p$ and $o \in VEP_o$, such that any input i executing p in F_p also executes o in F_o. In other words, the path constraints of p (C_p) imply those of o (C_o). We use an SMT solver to check whether $C_p \Rightarrow C_o$ by verifying the following formula:

$$unsat(\neg(C_p \Longrightarrow C_o)) \quad (1)$$

This ensures that an input satisfying C_p but not C_o does not exist (*i.e.,* $C_p \wedge \neg C_o$ is false).

Given a $p \in VEP_p$, to find its MPP, a naive approach would be to try all VEPs in VEP_o until a path o, for which (1) holds, is found. However, this search would require calling the SMT solver a number of times that has a quadratic complexity over the number of VEPs. To speed up this search, we rely on the intuition that if two paths are more "similar", they are more likely to be an MPP and satisfy (1). For this reason, they should be preferred first.

Preferential Matching. Based on the above intuition, we preliminary compute a heuristic "path similarity" metric between all possible pairs of paths (p',o'), where $p' \in VEP_p$, and $o' \in VEP_o$. The similarity score is computed using the formula:

$$S(p',o') = N_{MB}/(N_B^{p'} + N_B^{o'}) \quad (2)$$

where N_{MB} is the number of matching basic blocks obtained from binary diffing, $N_B^{p'}$ is the number of basic blocks in p', and $N_B^{o'}$ is the number of basic blocks in o'. If binary diffing fails to identify any matched basic blocks between p' and o', we use a string similarity score based on the textual representations of the path constraints of p' and o'.

After computing all the similarity scores, we sort the pairs by their similarity score and verify (1) starting with the highest-scoring pairs. For performance considerations, we limit the number of verifications attempts for each path $p' \in VEP_p$. If this limit is reached without finding an MPP for p', we assume that no MPP exists for this path.

If all valid paths in F_p and F_o are part of an MPP (*i.e.,* $\forall p \in VEP_p$, $\exists o \in VEP_o \mid \langle p,o \rangle \in MPPs$ and $\forall o \in VEP_o$, $\exists p \in VEP_p \mid \langle p,o \rangle \in MPPs$), the functions can be divided into MPPs, and we call them *divisible*. In this case, we verify whether a patch is StA by verifying whether the four properties (P1-P4) hold or not for all MPPs. If the functions cannot be divided into MPPs (*i.e.,* they are *indivisible*), we follow a similar approach to existing work [51] by combining different VEPs.

3) Verifying StA Properties: Next, we will discuss in detail how VERIBIN verifies four StA properties with or without MPPs, including non-increasing input space, non-local memory writes equivalence, return value equivalence, and function calls equivalence.

P1: Non-Increasing Input Space. We verify that the patch does not increase the valid input space, *i.e.,* all valid inputs to F_p are also valid inputs to F_o. Note that we cannot rely solely on new EEPs to determine whether a patch is StA since a patch can restrict the input space by either introducing new EEPs or by limiting the conditions under which an existing EEP can be reached. Therefore, VERIBIN checks P1 through path constraint implication rather than simply checking for the existence of new EEPs. We use C_{VEP_p} and C_{VEP_o} to denote the input space of F_p and F_o for all valid paths, respectively. Thus, P1 can be represented as $C_{VEP_p} \Rightarrow C_{VEP_o}$ where, $C_{VEP_p} = \bigcup_{p' \in VEP_p} \{C_{p'}\}$ and $C_{VEP_o} = \bigcup_{o' \in VEP_o} \{C_{o'}\}$. We elaborate on verifying whether $C_{VEP_p} \Rightarrow C_{VEP_o}$ holds or not in two cases: divisible and indivisible functions.

Divisible. As discussed in Section V-D2, F_o and F_p are divisible if $\forall p \in VEP_p, \exists o \in VEP_o$ such that $C_p \Rightarrow C_o$. Therefore, when F_o and F_p are divisible, $C_{VEP_p} = \cup\{C_p\} \Rightarrow \cup\{C_o\}$, where $\langle p, o \rangle$ is MPP identified in Section V-D2. Since $\cup\{C_o\} \Rightarrow C_{VEP_o}$, it follows that $C_{VEP_p} \Rightarrow C_{VEP_o}$, meaning P1 holds as long as F_o and F_p are divisible.

Indivisible. When the functions are indivisible, we obtain C_{VEP_p} by disjunctively combining (*i.e.,* using logic OR) the constraints of all paths in VEP_p. Specifically, $C_{VEP_p} = \bigvee_{p' \in VEP_p} C_{p'}$. Similarly, we get C_{VEP_o} by combining the constraints of all paths in VEP_o. We verify whether P1 holds or not by solving the following formula with an SMT solver:

$$unsat(\neg(C_{VEP_p} \Longrightarrow C_{VEP_o})) \quad (3)$$

Example. In Figure 1, the basic blocks with green background are matching basic blocks (BB_1 matches with BB_1', BB_2 with BB_2', etc.). For each $p' \in VEP_p$, we first calculate the path similarity score with each $o' \in VEP_o$. In this case, there is only one path in each, $P_{p_3} \in VEP_p$ and $P_{o_2} \in VEP_o$, yielding a similarity score $S(p_3, o_2) \approx 0.43$. We then check whether $C_{p3} \Rightarrow C_{o2}$. Clearly, C_{p3}, with an additional condition check, narrows the input space. Therefore, any input satisfying C_{p3} also satisfies C_{o2}, indicating $C_{p3} \Rightarrow C_{o2}$. As a result, the pair $\langle p3, o2 \rangle$ is identified as an MPP. Given that both VEP_p and VEP_o contain only one VEP, and $C_{VEP_p} = C_{p3} \Rightarrow C_{VEP_o} = C_{o2}$, the property P1 is met.

P2: Non-Local Memory Writes Equivalence. We check that all non-local memory writes operations in VEP_p^i of F_p write the same value to the same memory region as that in the corresponding VEP_o^i of F_o. Non-local memory writes (global writes) are defined as memory writes to addresses other than stack-relative addresses (*e.g.,* addresses relevant to `rsp` and `rbp` in x86-64). An example from Figure 1 illustrates that a write operation storing `edi` into `rbp - 4` in $BB_{1'}$ is not considered a global write, whereas storing `eax` into `rcx + rdx` in $BB_{3'}$ qualifies as a global write.

We represent global write operations with the notation G_P for F_p and G_O for F_o. Each global write is denoted as a tuple $\langle a,v \rangle$, with v being the symbolic value written and a being the corresponding address. We define equivalency between two symbolic expressions as $a \equiv b$ if $unsat(\neg(a == b))$, and inequivalency as $a \not\equiv b$ if $sat(\neg(a == b))$. Equivalence between write operations in F_p and F_o is defined as $a_o \equiv a_p$ and $v_o \equiv v_p$, meaning the addresses and values in these operations are equivalent, respectively. Accordingly, P2 holds (*i.e.,* $G_P \Leftrightarrow G_O$) if (1) $\forall \langle a_o,v_o \rangle \in G_O, \exists \langle a_p,v_p \rangle \in G_P$ such that $a_o \equiv a_p \wedge v_o \equiv v_p$, and (2) $\forall \langle a_p,v_p \rangle \in G_P, \exists \langle a_o,v_o \rangle \in G_O$ such that $a_o \equiv a_p \wedge v_o \equiv v_p$.

As with P1, we use different approaches to verify P2 when F_p and F_o are divisible or indivisible.

Divisible. In divisible functions, each global write g is denoted as $\langle a,v \rangle$, indicating symbolic value v is stored at address a upon satisfying path p'. For each MPP, $\langle p,o \rangle$, we use G_p and G_o to represent the global writes in the paths p and o, respectively. Given $C_p \Rightarrow C_o$, a direct comparison of v_o and v_p suffices without comparing their constraints. If $\forall \langle p, o \rangle \in MPPs$, $G_p \Leftrightarrow G_o$, we conclude that $G_{VEP_p} \Leftrightarrow G_{VEP_o}$, thus P2 holds.

Indivisible. When F_o and F_p are indivisible, we obtain G_{VEP_p} and G_{VEP_o} by merging writes of different values to the same address from all the paths VEP_p and VEP_o, respectively. For example, if address a has multiple values v_1, and v_2 from paths p_1, p_2, and p_3, represented as $\langle a,v_1 \rangle$, $\langle a,v_2 \rangle$, and $\langle a,v_2 \rangle$, we combine the path constraints for each unique value. For v_1, the constraint is C_1, and for v_2, it is $C_2 \vee C_3$. The merged value is determined using the formula:

$$v_{merged} = ITE(C_1, v_1, ITE(C_2 \vee C_3, v_2, v_{default})) \quad (4)$$

Where `ITE(c,a,b)` represents an If-Then-Else symbolic expression, with output `a` when constraint `c` is met, otherwise `b`. The $v_{default}$ serves as a placeholder for the memory's original value before any write operation.

Finally, to verify P2 (*i.e.,* $G_{VEP_p} \Leftrightarrow G_{VEP_o}$), we use an SMT solver, checking if $v_{p_{merged}} \equiv v_{o_{merged}}$. The SMT solver conditions are $(\neg(v_{o_{merged}} == v_{p_{merged}} \wedge C_{VEP_o} == True \wedge C_{VEP_p} == True))$, with C_{VEP} denoting the path constraint for all valid paths. This ensures the comparison of values only occurs within VEPs.

P3: Return Value Equivalence. We check that the return value of F_p along VEP$_p^i$ matches the return value of F_o along VEP$_o^i$. VeriBin verifies the return value equivalence only if the patch-affected function has a return value (which can be inferred from the function's prototype). We use R to denote the return value. P3 holds if $R_{VEP_p} \equiv R_{VEP_o}$. As before, we use different approaches for divisible and indivisible functions.

Divisible. In each path, the return value is stored in a specific return register (*e.g.,* `rax` in x86-64). Exploiting this fact, after symbolic execution, we retrieve the return value from the symbol value map of the current path (as mentioned in Section V-B). P3 holds if $\forall \langle p,o \rangle \in MPPs$, $R_p \equiv R_o$.

Indivisible. Similar to global writes, when the functions are indivisible, we obtain R_{VEP_p} and R_{VEP_o} by merging return values and their corresponding path constraints from all VEP_p and VEP_o using `ITE` expression. P3 holds if $unsat(\neg(R_{VEP_p} == R_{VEP_o} \wedge C_{VEP_p} == True \wedge C_{VEP_o} == True))$.

P4: Function Calls Equivalence. The function calls made along VEP$_p^i$ in F_p should be equivalent to those made along VEP$_o^i$ in F_o, *i.e.,* the same function should be called with the same arguments. As mentioned in Section II-B, a function call in F_o is considered equivalent to one in F_p if the callee function and its arguments are equivalent. For a callee function cf that is invoked multiple times, we use N_{cf}^o and N_{cf}^p to denote the number of calls to cf in VEP_o and VEP_p if indivisible (or MPP_o and MPP_p if divisible).

Divisible. When the functions are divisible, we check the function call equivalence for each MPP, and P4 holds if the function calls are equivalent on all MPPs. We identify callee functions and their arguments in p as $\langle cf_p, args_p \rangle$, where cf_p and $args_p$ represent the callee function and its arguments from `call` instructions in p. Similarly, we gather $\langle cf_o, args_o \rangle$ in o. Equivalence is established if $\langle cf_p, args_p \rangle \equiv \langle cf_o, args_o \rangle$, meaning for each cf_p, there's a matching cf_o where $cf_p \equiv cf_o$ and $args_p \equiv args_o$, and vice versa.

Initially, we check the equivalence of cf_p and cf_o. We consider $cf_p \equiv cf_o$, if these two functions are perfectly matched via binary diffing. For cases where $cf_p \equiv cf_o$, we then determine the number of times cf is called in p and o. If $N_{cf}^p = N_{cf}^o$, we then proceed to compare the arguments of cf based on the similarity of their corresponding basic blocks. In the absence of perfectly matched basic blocks, we resort to comparing the arguments in the sequence they appear. We consider $args_p \equiv args_o$ if the symbolic expression of each argument in $args_p$ is equal to the corresponding argument's symbolic expression in $args_o$.

Indivisible. When the functions are indivisible, we get all the function calls and their arguments from all VEPs in F_p and F_o (denoted by VEP_p and VEP_o respectively) instead of using MPPs. Specifically, we use $\langle CF_p, ARGS_p \rangle$ and $\langle CF_o, ARGS_o \rangle$ to denote all function calls and their corresponding arguments in VEP_p and VEP_o, respectively. We use the same approach to verify whether $\langle CF_p, ARGS_p \rangle \equiv \langle CF_o, ARGS_o \rangle$ as to verify whether $\langle cf_p, args_p \rangle \equiv \langle cf_o, args_o \rangle$ in the divisible case.

E. Adaptive Verification

In certain cases, patches that do not fully satisfy the StA properties may still be classified as StA by analysts through a process known as adaptive verification, particularly when they introduce semantically equivalent changes. Adaptive verification involves providing additional information to analysts, pinpointing the precise cause of unsafety, and allowing them to decide whether to disregard the identified issue if it is deemed acceptable. For example, replacing a hashing function with a safer version (replacing `SHA1` with `SHA256`) may cause a deviation from the StA properties due to a different function call. However, an analyst may consider this modification acceptable.

When the `adaptive` mode is enabled, VeriBin identifies the root cause of any failure in the P1-P4 checks and prompts the analyst to determine whether the detected difference can be regarded as StA. Table I provides a detailed overview of the information about the queries VeriBin generates during the adaptive verification process, including the type of questions (Q), scenarios when each question is asked (Scenario), and the question or task presented (Prompt). The questions are as follows: **(Q1)** During P2, when the original value and the patched value differ, we highlight the differences in the symbolic expression, provide a counterexample that leads to different values in two versions, and ask the analyst if these differences can be considered as StA. **(Q2)** When there are additional (or fewer)

TABLE I: VERIBIN questions for adaptive verification. *Q*: type of questions. *Scenario*: specific conditions under which the question is asked. *Prompt*: the question or task presented.

Q	Scenario	Prompt
Q1	$\exists\langle a'_o, v'_o\rangle \in G_O,\ \langle a'_p, v'_p\rangle \in G_P$: $a'_o \equiv a'_p \wedge v'_o \not\equiv v'_p$	Can the difference between v'_o and v'_p be considered equivalent (StA)
Q2	$\forall\langle a_o, v_o\rangle \in G_O$, $\exists\langle a'_p, v'_p\rangle \in G_P$: $a_o \not\equiv a'_p$, or vice versa ($\langle a'_o, v'_o\rangle$)	Can VERIBIN consider $\langle a'_p, v'_p\rangle$ (or $\langle a'_o, v'_o\rangle$) to have no side effects (StA)?
Q3	$\forall cf_o \in VEP_o$, $\exists cf'_p \in VEP_p$: $cf_o \not\equiv cf'_p \wedge \forall cf_p \in VEP_p$, $\exists cf'_o \in VEP_o$: $cf_p \not\equiv cf'_o$	Can cf'_o and cf'_p be considered equivalent? ($cf'_o \equiv cf'_p$)
Q4	$\exists cf'_o \equiv cf'_p$: $N^o_{cf'_o} \neq N^p_{cf'_p}$	Can VERIBIN consider cf'_o (same as cf'_p) to have no side effects (StA)?
Q5	$\exists cf'_o \equiv cf'_p \wedge N^o_{cf'_o} = N^p_{cf'_p}$: $args^i_o \not\equiv args^i_p$	Can VERIBIN consider $args^i_o$ and $args^i_p$ to be equivalent?

global writes in F_p compared to F_o, VERIBIN asks whether this global write can be considered to have no side effects (StA). **(Q3)** When VERIBIN finds unmatched function calls from all the VEPs, VERIBIN asks whether the two external function calls can be considered equivalent. **(Q4)** When one function call is detected to be called a different number of times in F_p compared to F_o, VERIBIN asks whether this function can be considered to have no side effects (StA). **(Q5)** When the ith argument in one matching function call pair is different, VERIBIN asks whether the arguments can be considered the same.

The adaptive verification process is designed to aid the analyst in analyzing patch-introduced changes. We expect the analyst to have some understanding of the patch's intended behavior and the patched binary and to provide accurate information to VERIBIN. For example, if the analyst knows the the patch involves adding a `memset` function call, and VERIBIN confirms that the patch-introduced changes are about adding a `memset` function call, the analyst can unequivocally answer the question, *i.e.,* Q4, with 'Yes'. When the analyst is unsure about a question, they can conservatively answer with 'No', and VERIBIN will consider the patch as not Safe to Apply (non-StA). Based on the analyst's responses, we update the symbolic expressions and re-check the corresponding property. For example, if the analyst answers 'Yes' to Q1, VERIBIN will add the information that $v_o == v_p$ (*i.e.,* v_o is equal to v_p) to the SMT solver and re-check the property. We also record the analyst's responses to the questions, which can be used to improve the tool's performance in future analyses to avoid asking the same questions.

VI. IMPLEMENTATION

We implement VERIBIN using about 4,100 lines of Python code. For preprocessing, we used BinDiff [91] to get information about matching functions and basic blocks, IDA PRO [5] and `angr` [1] to collect the callee functions' prototypes in F_o and F_p.

Additionally, `angr` serves as our static binary analysis tool for extracting the Control Flow Graphs (CFGs) for both F_o and F_p. Furthermore, `angr` is used for the under-constrained symbolic execution on both F_o and F_p. We implement a plugin within `angr` to collect symbolic information across each execution path. This plugin monitors the symbolic values regarding P1-P4 for each execution path using breakpoints and some handler functions. We also attach hooks to function call instructions, handling them as uninterpreted functions, without executing the actual call.

During the symbolic execution phase, we use Claripy [3], the default SMT-solver utilized by angr, to generate symbolic expressions and manage constraint verification. Despite Claripy functioning as an abstracted constraint-solving interface for Z3 [29], it lacks support for uninterpreted functions. To address this, we supplement Claripy's capabilities by implementing a wrapper object for Z3's uninterpreted function. After symbolic execution, we collect the resulting Claripy expressions and convert them into Z3 expressions. These are subsequently fed into the Z3 SMT solver for checking StA properties. We use these tools because of their availability, reliability, and wide adoption in existing program analysis research works.

VII. EVALUATION

In this section, we first describe the evaluation setup along with our dataset. Next, we present the evaluation of VERIBIN in terms of effectiveness and efficiency, for both un-stripped and stripped binaries. We then discuss the adaptive verification mechanism and its evaluation. Finally, we present the comparison with a source-level patch verification technique, SPIDER [51].

A. Experimental Setup

We evaluated VERIBIN using a server (AMD EPYC 7773X) with 256 CPUs and 1TB of memory.

Dataset. We evaluate VERIBIN with three datasets: MicroPatch Bench [45], AMP Challenges dataset, and PatchDB binaries dataset. For each dataset, we compile both unstripped (**D1**, **D2**, and **D3**) and stripped (**D1_s**, **D2_s**, and **D3_s**) versions of the binaries.

The MicroPatch Bench dataset contains 62 real-world patches fixing CVEs affecting various open-source projects. Each patch within this dataset includes the source code of the original binary and a Dockerfile to generate the original and patched versions of the project. We run the provided Dockerfiles to generate 62 pairs of original and patched binaries. This dataset was created and provided to us by an external group of researchers.

The AMP challenges dataset is provided by DARPA for their Assured MicroPatching (AMP) project [8]. It contains 7 binaries in both their original and patched version, fixing different types of vulnerabilities. These binaries are designed for use with embedded platforms that are commonly found in the vehicle industry and are compiled for different architectures, such as `ARM` and `AVR` architectures.

PatchDB [77] is a large patch dataset containing source code commits, including security and non-security patches. However, this dataset does not include binaries or build instructions, requiring us to undertake these additional steps. Our criteria for selecting patches are stringent: they must be security-related with a CVE number, part of a project written in C, and not from large projects like Linux kernel. Subsequently, our focus is solely on FFmpeg, the project with the most patches. For this, we create a Dockerfile to compile both the original and the patched binaries. We further refine our selection to 56 patches, each impacting no more than three functions, with the largest affected function being less than 2,000 bytes in size.

TABLE II: Experiments Description

	Adaptive	MPP	CIOC
E1	✖	✔	✔
E2	✔	✔	✔
E3	✖	✖	✔
E4	✖	✔	✖

Ground Truth Collection. To gauge the accuracy of our tool in an automatic mode and the effectiveness of adaptive verification, we established ground truth (*i.e.,* whether a patch is a StA or not), using two different criteria. The first criterion, *Strict Adherence Ground Truth (GT_Strict)*, verifies if a patch strictly adheres to the StA properties in an automated mode without human interaction. The second criterion, *Analyst Judgment Ground Truth (GT_Analyst)*, considers whether a security analyst would deem a patch StA, even with semantically equivalent changes that may violate StA properties. Specifically, we classify a violation as StA if it: (1) Adds initialization code to global variables (*e.g.,* assigning a constant value `0` or calling `memset`); (2) Adds a call to a function with no side effects (*e.g.,* logging or `strlen`); (3) Replaces a function call with another having the same semantic effect, *e.g.,* replacing an insecure hashing function (`SHA1`) with a safer version (`SHA256`); (4) Modifies the arguments of a function that has no side effects. The ground truth is collected by two authors manually reviewing the source code of the patches, determining whether each patch meets our criteria for being considered StA.

Experiment Configuration. As depicted in Table II, we conducted four experiments denoted as E1, E2, E3, and E4. E1 serves as our baseline experiment, representing the baseline configuration. E2 focuses on a specific aspect: adaptive verification. E3 and E4 focus, respectively, on the utilization of Matching Path Pairs (MPPs) optimization, and the detection of Compiler-Introduced Offset Changes (CIOCs). Their details are provided in our ablation study (see Appendix A) to quantify the contribution of MPPs and CIOCs. On average, using MPPs reduced the verification time by 75%, while using CIOCs improved the accuracy by 12%.

B. VERIBIN *Results: un-stripped binaries*

In this subsection, we will discuss the VERIBIN results obtained in E1, on un-stripped binaries (D1, D2, D3), and examine several key aspects: applicability, correctness, false negatives, effectiveness of the EEP detection heuristics, manual configuration, and runtime.

TABLE III: Dataset Composition. *Arch.*: architecture is not supported. *No Func.*: Cannot detect patch-affected target functions. *Timeout*: Analysis timeout. *Memory*: Analysis memory exceeded.

Dataset	Total	Support	Excluded and Corresponding Reasons			
			Arch.	No Func	Timeout	Memory
D1	62	42	0	4	2	14
D2	7	6	1	0	0	0
D3	56	38	0	0	11	7
Unstripped	125	86	39			
D1_s	62	41	0	5	2	14
D2_s	7	6	1	0	0	0
D3_s	56	38	0	0	11	7
Stripped	125	85	40			

Applicability. As shown in Table III, for D1, we support 42 patches in this dataset, with some exclusions due to various challenges: Firstly, difficulty in accurately identifying the patch-affected functions in the binaries (labeled as 'No Target Func'). This challenge often arises when source-level modifications are not detected by binary diffing tools, which may occur either because the patch is not successfully applied, or because the modification is too subtle to be detected (*e.g.,* replacing a string). Secondly, there are instances where our analysis exceeded an 8-hour time limit (noted as 'Timeout'). Thirdly, there are cases where our analysis surpassed a 100 GB memory limit (marked as 'Memory'). For D2, we support 6 out of 7 patches, excluding one binary compiled for the `AVR` architecture, which is not supported by `angr` (marked as 'Arch.'). For D3, we support 38 out of 56, with exclusions due to timeouts (11 instances) and memory exceeded limitation (7 instances). Overall, we support 86 out of 125 patches (68.8%) in the un-stripped dataset.

TABLE IV: Results Categorization (StA: Safe to Apply, TP/FP: True/False Positives, TN/FN: True/False Negatives)

		VERIBIN StA	VERIBIN non-StA
Ground Truth	**StA**	TP	FN
Ground Truth	**non-StA**	FP	TN

Correctness. We evaluate the correctness of VERIBIN by comparing its results with the ground truth. Table IV shows how we define True/False positives and True/False negatives, with respect to the ground truth. Note that we also classify such cases as False Negatives if the StA properties that fail as determined by our tool, VERIBIN, do not align with the established ground truth result for non-StA patches. Although not desired, it is acceptable to have false negatives, where in VERIBIN considers a StA patch to be non-StA — in these cases, an analyst might have to verify the patch manually.

We use the accuracy metric (ACC) and false positive rate (FPR) to measure the correctness of the patch verification results produced by VERIBIN, where $ACC = \frac{TP+TN}{P+N}$ and $FPR = \frac{FP}{FP+TN}$.

TABLE V: VERIBIN Results Summary. *ACC*: accuracy. *FPR*: False Positive Rate.

Dataset	Number	ACC	FPR	VERIBIN Result TP	TN	FP	FN
D1	42	90.5%	0.0%	**17**	**21**	0	**4**
D2	6	100.0%	0.0%	0	**6**	0	0
D3	38	94.7%	0.0%	**12**	**24**	0	**2**
Unstripped (E1)	86	93.0%	0.0%	**28**	**52**	0	**6**
D1_s	41	85.4%	0.0%	**15**	**20**	0	**6**
D2_s	6	83.3%	0.0%	0	**5**	0	**1**
D3_s	38	94.7%	0.0%	**12**	**24**	0	**2**
Stripped (E1)	85	89.4%	0.0%	**27**	**49**	0	**9**
D1	42	90.5%	0.0%	**26**	**12**	0	**4**
D2	6	100.0%	0.0%	**2**	**4**	0	0
D3	38	94.7%	0.0%	**18**	**18**	0	**2**
Unstripped (E2)	86	93.0%	0.0%	**46**	**34**	0	**6**

Table V summarizes the results for VERIBIN. For a more detailed analysis of our evaluation, including comprehensive results for each patch, please refer to our GitHub repository [11]. In these datasets, VERIBIN identifies StA patches and correctly describes patch behaviors regarding StA properties, with 93.0% ACC and 0% FPR. We also notice that VERIBIN performs better for D2 than D1 and D3, because these patches are relatively simpler (*i.e.,* mostly adding additional checks) and targetting smaller functions. In a more conservative approach, categorizing all patches that time out or exceed memory limits as FN, the ACC drops to 66.7%.

False Negatives. As shown in Table V, in total, among the 3 datasets, there are 6 false negatives (*i.e.,* VERIBIN considered a StA patch to be non-StA, or the failing properties for a non-StA patch determined by VERIBIN do not match with the ground truth result). For patch #9 and #35, the CFG recovered from `angr` is broken. For patches #7 and #70, where changes are introduced into loops, VERIBIN could not explore all execution paths adequately due to our unrolling-loop-once strategy, resulting in path constraint implication failures. For patch #54, VERIBIN encounters difficulties in correctly pairing varying indirect calls between the original and the patched functions. For patch #30, VERIBIN fails to explore all execution paths due to un-initialized inter-function global variables.

Manual Configuration. Among D1, D2, and D3, we need human effort to correct the configuration file for five patches. This additional information pertains to Error-handling Exit Paths (EEPs), where our existing EEP detection mechanisms fail to detect all patch-related EEPs. These EEPs fall into distinct categories: (1) Functions with no return values (Patch #12, #24); (2) Functions that return 0 as an invalid return (Patch #42, #77); (3) Functions that return a special return format: as in Patch #34, where a function returns a structured value. A comprehensive evaluation of the EEP detection heuristics' effectiveness is detailed in Appendix B.

Runtime. To evaluate the runtime of VERIBIN, we record the time elapsed excluding the preprocessing step (which is a one-time, cached effort, requiring an average of 922.0 seconds for D1, D2, and D3 samples). Across D1, D2, and D3 samples, the average runtime of VERIBIN is 1,293.8 seconds, with 566.3 seconds for symbolic execution and 635.8 seconds for checking StA properties.

Based on our evaluation, we found that, on average, the most time-consuming procedure is symbolic execution, which correlates with the number of reachable execution paths and the complexity of each execution path. However, in some specific cases, querying the SMT solver can take even longer. For example, for patch #35, VERIBIN required 6 hours for the verification to finish, where around 5k seconds is for symbolic execution and around 15k seconds are for StA properties checking. With further analysis, we find the reason is that the patch-affected function is a hashing function with complicated calculations, making it harder for the SMT solver to solve the equivalence constraints of the symbolic expressions.

Overall, our results demonstrate that VERIBIN can effectively verify closed-source patches, since our design choices guarantee a high level of assurance that patches classified as StA by VERIBIN are indeed StA.

C. VERIBIN *Results: stripped binaries*

This subsection presents VERIBIN results from E1, focusing on stripped binaries (D1_s, D2_s, D3_s). Configuration files are regenerated for these stripped binaries, with manual configuration for five patches regarding EEPs, as in the un-stripped binaries.

Applicability. As shown in Table III, compared with unstripped binaries, we fail to handle one additional patch in the stripped dataset. The reason is that for Patch #30, the tools used in our pre-processing step fail to correctly recognize the target function in the stripped binary. Overall, VERIBIN supports 85 out of 125 patches (68%) in the stripped dataset.

Correctness. In the stripped datasets (D1_s, D2_s and D3_s), VERIBIN identifies StA patches and correctly describes patch behaviors regarding StA properties, with 89.4% ACC and 0% FPR, as shown in Table V.

False Negatives. Among the three datasets, there are nine false negatives. For five patches (*i.e.,* #9, #35, #7, #70, and #54), we inherit the same false negatives as the unstripped binaries. The additional four false negatives are due to inaccurate information for stripped binaries, which fall into the following two categories: (1) Incorrect function prototype information: Stripped binaries often have problematic function prototypes. Information such as the type of function arguments and whether a function returns a value can be incorrect, leading to three patches (#16, #28, and #45) having incorrect results. (2) Incorrect matching function information: For patch #42, the pre-processing step fails to correctly match functions in the original and patched binaries. We verified that an analyst can potentially correct the four additional false negatives by manually providing accurate information about functions' prototypes and matching functions' locations in the configuration file.

Runtime. Similar to unstripped binaries, the average runtime of VERIBIN for stripped binaries is 1,367.4 seconds, with 635.9 seconds for symbolic execution and 657.2 seconds for checking StA properties.

Generally, VERIBIN performs slightly less effectively in stripped binaries compared to unstripped binaries. The primary challenge in stripped binaries is recovering the accurate function information, including function prototypes, matching functions, and the function CFGs. Without this information, VERIBIN may face difficulties in correctly identifying the patch-introduced changes, which can impact the accuracy of StA properties checking.

D. *Adaptive Verification Evaluation (E2)*

TABLE VI: VERIBIN questions for adaptive verification. *Q*: type of questions. *N_p*: the number of patches in which the current question is asked among the datasets. *Occ.*: the total number of occurrences.

Q	Q1	Q2	Q3	Q4	Q5	Total
N_p	21	18	5	20	14	49
Occ.	25	26	5	33	26	115

As discussed in Section V-E, when a patch does not satisfy the StA properties automatically, an analyst can further verify this patch with adaptive verification enabled. In this case, VERIBIN may ask analysts for specific information regarding the detected differences. Table VI provides a statistics overview of the information about the queries VERIBIN generates during the adaptive verification process, including the type of questions (Q), the number of patches in which the question is asked (N_p) and the total number of occurrences for a question (Occ.). Note that a single patch may require the same type of question multiple times. We find that Q1 is the most common type of question, where VERIBIN highlights differences between the original and patched values and asks if these differences can be considered StA. The second most common question type is Q4, where VERIBIN detects differences in the number of function calls between the original and patched functions. For example, in patch #25, the analyst could know from the patch description that the patch involves adding a `memset` function call, and VERIBIN detects that F_p has an

additional function call to memset compared to the F_o, the analyst can assuredly answer Q4 with 'Yes'. On average, 1.3 questions are asked per patch. Based on the authors' experience in determining the answers for each question, we estimated that each question takes around 10 minutes for an analyst to answer. On average, the total human effort for adaptive verification is estimated to be approximately 15 minutes per patch.

Notably, when considering semantically equivalent changes, we observed that 50 out of 86 (58%) patches are deemed StA in GT_Analyst (*i.e.,* whether a security analyst would consider a patch StA in scenarios involving semantically equivalent changes), whereas in GT_Strict, only 32 of 86 (37%) patches strictly adhere to the StA properties. Remarkably, as shown in Table V, in E2, the accuracy remains consistent at 93.0%, mirroring the performance in E1, as VeriBin effectively identifies patch-introduced changes. For the additional 18 StA patches, VeriBin properly interacts with the analyst, correctly identifying semantically equivalent changes as equal. In Section VIII, we provide a case study showing a concrete example of questions asked to the analyst.

E. Comparison with source-level technique SPIDER

In this subsection, we compare VeriBin with SPIDER [51], a state-of-the-art source-level patch verification tool. SPIDER verifies patches by comparing the source code of the original and patched versions of a program. To ensure a fair comparison, we run VeriBin on the un-stripped dataset (D1, D2, D3) and provide SPIDER with the corresponding source-level differences, evaluating their applicability and correctness.

SPIDER shares similar StA definitions with VeriBin, but it does not check each StA property as VeriBin does. Since SPIDER only provides StA or non-StA results without detailed information about the patch-introduced changes, we limited our comparison to these binary outcomes, disregarding the detailed StA properties. Additionally, SPIDER hooks certain library functions (*i.e.,* strcpy, strncpy, strlcpy, memcpy) and execute them in a pre-defined manner to ignore them as additional function calls. Therefore, we compare SPIDER results using a slightly different ground truth, GT_SPIDER, which is based on GT_Strict but modified 4 results to align with SPIDER's definition and implementation of StA.

Among the 125 patches from D1, D2, and D3, SPIDER supports 79 patches (63.2%), and correctly classifies 49 patches according to GT_SPIDER, with 62.0% ACC and 27.9% FPR. In contrast, VeriBin supports 86 patches (68.8%), and correctly classifies 83 patches according to GT_Strict, with 96.5% ACC and 0% FPR. For 33 patches, both SPIDER and VeriBin have correct results. We provide detailed results in Appendix C.

The differences in applicability and correctness between the tools can be attributed to their respective analysis levels and inherent limitations. SPIDER, being a source-level tool, can directly analyze the source code, making it more suitable for patches where VeriBin encounters timeouts or memory limits. However, SPIDER also fails on some patches due to limitations in source-level analysis, such as file type restrictions and difficulties in identifying patch-affected functions. Regarding correctness, SPIDER's failures stem from issues like incorrect detection of error-handling basic blocks, variable type differences, and ignoring extra function calls in error-handling blocks. These limitations were confirmed by the authors of SPIDER.

Overall, while source-level tools like SPIDER offer better applicability theoretically, they have practical limitations. In obfuscated scenarios, such as the XZ Utils backdoor where SPIDER cannot detect the patch-affected functions, binary-level tools like VeriBin provide a more reliable solution.

VIII. Case Studies

We now present four interesting case studies to demonstrate the effectiveness of VeriBin in detecting StA patches.

```
1  int64 crc64_resolve(void){
2    ... # variables initialization
3  -  __asm {cpuid}
4  -  if (_RAX){
5  -    __asm {cpuid}
6  -    if ((~_RCX & 0x80202) == 0)
7  +  v0 = __get_cpuid(1, v2, v3, &v4, v5, v6);
8  +  if (v0){
9  +    if ((~v4 & 0x80202) == 0)
10       return &crc64_arch_optimized;
11   }
12   return &crc64_generic;
13 }
```

Listing 3: Simplified pseudocode of the malicious modifications in XZ Utils, created by comparing decompiled code from version 5.5.2beta and version 5.6.0 of liblzma.so. In version 5.6.0, the original cpuid instruction is replaced with a call to a compromised __get_cpuid function. Note that these modifications are not directly visible in the source code.

CVE-2024-3094 (XZ Utils backdoor). To demonstrate the effectiveness of VeriBin in detecting real-world malicious patches, we applied it to analyze a binary affected by the recently discovered backdoor [10] in the XZ Utils package [6]. This backdoor was introduced in XZ Utils by modifying the build process of the liblzma library. Specifically, during compilation, an obfuscated script added to the compilation pipeline modifies the original source code, so that in the symbol resolution functions crc64_resolve, the assembly instruction cpuid is replaced by a function call to the malicious __get_cpuid function (as shown in Listing 3).

VeriBin can easily detect this malicious update by comparing the crc64_resolve function across two versions of liblzma.so: version 5.5.2beta (the last non-compromised version) and version 5.6.0 (the first version containing the malicious code) [2]. In particular, VeriBin detects an additional function call (__get_cpuid) in the malicious version, invalidating condition P4. Additionally, this modification alters how the function's return value is computed, invalidating condition P3. For these reasons, the patch is flagged as non-StA. These observations align with the underlying mechanics of the backdoor, demonstrating VeriBin's effectiveness in identifying and flagging non-StA patches.

We note that this backdoor highlights the need to employ verification of patches at the binary level for security-relevant projects, even when their source code is available. In fact, this backdoor cannot be easily detected by analyzing the source-code-level differences between version 5.5.2beta and version 5.6.0 of liblzma, due to the usage of obfuscation and the fact that the malicious modifications change the package's compilation pipeline, rather than its source code. For this reason, we envision the usage of VeriBin as part of a verification pipeline in which in every new version of a security-critical

[2] crc64_resolve is the function for which BinDiff returns the lowest similarity score when comparing the two versions of liblzma.

software package, the newly generated binary is compared with the binary of the previous version. If a modified function is detected as non-StA, an operator can be notified with the detected differences to verify whether or not they are expected.

```
1  int encrypt(...){
2    EVP_CIPHER_CTX *ctx; ...
3  - if(1!=EVP_EncryptInit_ex(ctx,EVP_des_ede3_cbc(),NULL,key,iv))
4  + if(1!=EVP_EncryptInit_ex(ctx,EVP_aes_256_cbc(),NULL,key,iv))
5    { handleErrors(); ... }
6    ...
7  }
```

Listing 4: Challenge 05 Patch (D2, #46), where an insecure cryptographic function (3DES) is replaced with a more secure version (AES), maintaining the same semantics.

Challenge 05 (D2, #46). Listing 4 illustrates the source code of a patch to a C-based program designed to run on a BeagleBone Black board [2]. This program encrypts log messages before publishing them on a CAN bus. The patch exemplifies the typical scenario in which a program is patched to replace an insecure cryptographic function (`3DES`) with a more secure version (`AES`), maintaining the same semantics. As such, the patch cannot be considered as StA according to our definition. However, it is reasonable to assume that an analyst can consider the 3DES cryptographic function as equivalent to the AES one. In this scenario, it is possible to use VeriBin's adaptive verification to ask the analyst to provide information regarding the semantic equivalence of 3DES and AES, and use the provided information to augment the analysis.

When verifying this patch, the `EVP_des_ede3_cbc` function does not automatically match with the `EVP_aes_256_cbc` function. For this reason, VeriBin seeks help from the analyst and asks whether these two functions are considered semantically equivalent. If the analyst answers "yes", VeriBin considers the two functions equivalent when verifying P1-P4. Consequently, it determines that P1 to P4 hold, indicating that, under the provided assumptions, this patch is StA.

```
1    ...
2  - if (... && !(nmemb && size))
3  + if (... && nmemb && size)
4      return AVERROR(ENOMEM);
5    return 0;
```

Listing 5: Patch for CVE-2013-4265 (D3, #78) that enlarges the input space, showcasing a non-StA patch accurately identified as non-StA by VeriBin.

CVE-2013-4265 (D3, #78). Listing 5 shows a simplified version of the source code associated with this patch. Before the patch, the function would return an error if either `nmemb` or `size` was non-zero. Post-patch, it returns an error only when both `nmemb` and `size` are non-zero. Essentially, the patch turns certain Error-handling Exit Paths (EEPs) to Valid Exit Paths (VEPs), thereby expanding the input space where P1 (non-increasing input space), is expected to fail. VeriBin correctly detects that the value corresponding to `AVERROR(ENOMEM)` is an invalid return value, subsequently filtering out EEPs and confirming the failure of P1. This is an illustrative example of a non-StA patch that gets correctly identified as non-StA by VeriBin.

Dillo-png.203-feh (D1, #8). Listing 6 shows the source code of the patch and part of the patched binary, which exemplifies how VeriBin handles Compiler-Introduced Offset Change (CIOC). This patch adds a check to the product of `png_ptr->height` and

```
1  //Source code
2  void png_handle_IHDR(png_structp png_ptr, ...){
3    png_byte buf[13]; png_uint_32 width; ...
4    width = png_get_uint_31(png_ptr, buf);
5    png_ptr->width = width; ...
6  + if (!((png_ptr->height * png_ptr->width >> 32) <= 0
7  + && (png_ptr->height * png_ptr->width))<= 536870911)){
8  +  exit(-1);}
9    ... }
10 ----------------------------------------------------------
11 //Assembly code
12     ...
13 -    sub rsp, 0x78
14 +    sub rsp, 0x88
15   ...
16   lea rsi, [rsp + 0x30]    ; original: rsi = rsp - 0x48
17                            ; patched: rsi = rsp - 0x58
18   mov rdi, rbx             ; rdi = png_ptr
19   call 0x404830            ; png_get_unit_31(png_ptr, buf)
20   mov [rbx + 0x1c8], rfax   ; png_ptr->width = width
21   ...
```

Listing 6: Simplified dillo-png.203-feh Patch (D1, #8), which exemplifies how VeriBin handles Compiler-Introduced Offset Changes (CIOCs).

`png_ptr->width`, which represents the size of the processed PNG file. At the source code level, this patch only affects P1, and it is StA since it decreases the function's input space. However, at the binary level, the compiler increases the stack size from `0x78` to `0x88`, changing the offsets of stack variables. For instance, `buf` is located at `rsp - 0x48` in F_o but at `rsp - 0x58` in F_p, which affects the arguments and return values in function calls. VeriBin detects this CIOC using the structural position correlation method, and correctly determines this patch as StA.

IX. Limitations and Future Work

In this section, we discuss VeriBin's limitations on soundness, scalability, and detection capabilities, followed by potential future directions for improvement.

A. Soundness

Unsoundness from inaccurate function information recovery. As VeriBin relies on several existing tools (*e.g.,* BinDiff, IDA Pro, `angr`, etc.) to analyze binaries, it inherits their limitations and imprecisions. Specifically, we collect matching function information from BinDiff, and we collect function prototypes using a combination of IDA Pro and `angr`; finally, we recover function CFGs using `angr`. These tools may not always provide accurate information, leading to potential inaccuracies in the analysis, especially in stripped binaries. Inaccuracies, including mismatching between functions, missing or wrong function prototypes, and incorrect function CFGs can lead to potential false positives and negatives. For example, if the recovered CFGs do not contain some basic blocks of a function, the tool will not be able to detect the patch-induced changes in those blocks, leading to a potential false positive. As a mitigation, we provide analysts with the option to manually adjust the automatically generated information regarding the analyzed functions. A potential direction for future work would be to investigate the application of advanced machine learning algorithms and deep learning frameworks [43], to enhance the extraction of function information from stripped binaries.

Unsoundness due to heuristics. A potential source of unsoundness in VeriBin is the reliance on heuristics to identify Error-handling Exit Paths (in Section V-C) and identify Compiler-Introduced Offset Changes (in Section V-D1).

However, these heuristics may not yield perfect results for all patches. In an extreme case when all the paths are considered as EEPs, the StA properties check will not be able to detect any differences between the original and patched binaries, leading to potential false positives. As a mitigation, we allow analysts to manually modify information about EEPs in the analysis configuration file. The development of a more advanced and sophisticated detection mechanism for EEP and CIOC can be a prospective direction for future research.

Unsoundness due to coverage. Another source of unsoundness in VERIBIN is the coverage issue. When symbolic execution fails to reach the basic block where patch-induced changes have been applied, either because it does not fully unroll a loop or because the reaching constraint is not satisfiable, our tool may subsequently fail to detect the discernible differences, potentially resulting in false positive results. To mitigate this, VERIBIN issues warnings for unvisited basic blocks modified by patches, which helps alert analysts to potential areas where the symbolic execution did not achieve full coverage.

B. Scalibility

As a binary-level tool utilizing symbolic execution and SMT solvers, VERIBIN faces inherent limitations related to these technologies. In our dataset of 125 patches, we excluded 14 patches due to timeouts and 21 patches due to memory exceedances. Upon investigation, we find that all the excluded timeout cases are due to symbolic execution, a prerequisite process preceding the StA properties check, and the memory exceedances are caused by both symbolic execution and the SMT solver. We notice that both the timeout and memory exceedances are due to the complexity of the function rather than the patch itself. Symbolic execution struggles with the exponential number of paths, leading to path explosion and subsequent timeouts and memory exhaustion. Similarly, the SMT solver is restricted by computational limitations, especially when handling non-linear mathematical constraints or when solving complex constraints emerging from a great number of paths in symbolic execution. To address these limitations, future developments for VERIBIN aim to implement more sophisticated path-pruning strategies and explore the efficacy of an abstraction-refinement approach [17]. These strategies are intended to filter out irrelevant paths and reduce the number of paths to be analyzed, thereby mitigating the issues of timeouts and memory exceedances.

C. Detection Capability

Our definition of StA patches focuses on ensuring that the patch does not break the existing functionality. However, this definition is not exhaustive, as it does not verify whether a patch addresses the vulnerability it was intended to fix. The limitation means that while a patch may be considered StA (*i.e.,* functionality-preserving) based on our current criteria, it may not necessarily resolve the underlying security issue. Additionally, our current approach does not consider the runtime characteristics such as time and space overhead, which are crucial in safety-critical applications like RTOS firmware for embedded systems, where strict execution deadlines must be met. While we have demonstrated that VERIBIN can analyze a wide variety of patches, it is affected by scalability issues that hinder its ability to analyze some patches, especially those affecting large functions. Additionally, coverage issues can prevent it from fully exploring the paths affected by patches.

To enhance the thoroughness of patch analysis, one future direction is to combine VERIBIN with directed fuzzing techniques. Directed fuzzing [40], [42] can target specific parts of the code, such as those affected by patches, and can help in identifying whether the intended vulnerability has been effectively mitigated.

X. RELATED WORK

Patch Analysis. Patch analysis [16], [61] has received much attention in the past decade with the rise of supply chain attacks [62]. Researchers have proposed techniques to analyze patches to identify vulnerability-contributing changes [66], understand the underlying software's evolution [18], [22], [37], [67], [72], link patches with bug reports [80], [88], search for patch applicability [13], [41], etc. VCCFinder [66] uses the code metrics and patch features (*e.g.,* keywords in commits) to identify patches that introduce vulnerabilities. ReDeBug [41] normalizes a given patch at its Abstract Syntax Tree (AST) level and uses it to identify other code fragments where the patch is applicable. Similarly, PatchScout [74] and TRACER [81] use various syntax-level features to rank patches according to their applicability to a given vulnerability report. SID [79] uses a set of symbolic patterns to identify patches that fix security vulnerabilities, while another similar work [39] identifies security patches in adjacent binary versions, by combining code property graphs analysis and language model-based analysis of instruction semantics. These patterns are designed based on the domain knowledge of security patches.

SPIDER [51], proposed by Machiry et al., formalizes the idea of *Safe to Apply* (StA) patches — patches that do not break the existing functionality. However, differently from VERIBIN, SPIDER requires the source code of the patched program and aims to be completely automated, accepting no human input, which leads to a failure to detect many (45%) security patches that are StA. In contrast, VERIBIN aims to prove that a patch is StA and adapts to analyst's feedback for cases that are hard to prove automatically.

Finally, most of the existing techniques [14], [19], [23], [51], [53], [70], [79], [80] work on source-level patches and use other patch metadata, such as the number of lines and commit message [72]. These techniques exploit source-level information. For instance, SPIDER uses AST matching to identify the statements and variables that are affected by the patch and focuses its verification on these variables. Similarly, the techniques used in SID assume the existence of symbolic expressions involving source variables. There are hybrid techniques, such as FIBER [83] and EFIBER [85], that use source-level information on binaries. These techniques use the source-level patch to get a binary-level signature, which can be used to test for the existence of the corresponding patch in other binaries. Conversely, our work does not assume the availability of source code and directly uses program binaries to identify the semantic differences between them.

Sementics Equivalence Checking. Semantic equivalence checking has been an active area of research. Regarding checking semantic equivalence at the source code level, these techniques can be broadly categorized into static and dynamic,

each addressing specific challenges. Early static analysis approaches such as SYMDIFF by Lahiri et al. [46] offer language-agnostic solutions using the Boogie intermediate language, but it requires complex user-provided specifications that match variables between the functions under comparison. Subsequent works focus on enhancing scalability and usability. For example, Malik et al.'s DIFFKEMP [52] checks the preservation of semantics of refactored code of large C projects, while Wang et al.'s Last Diff Analyzer [78] expands support to multiple languages, specifically Go and Java. A recent advancement in this area is ARDiff by Badihi et al. [17], which utilizes iterative abstraction and refinement to scale equivalence checking for syntactically similar programs. Complementing these static approaches, dynamic analysis approaches like Churchill et al.'s semantic program alignment technique [27] utilize execution traces to offer in-depth behavioral insights, however, it relies on comprehensive test cases provided by users.

Researchers have also developed techniques that directly check equivalence at the assembly or binary level. Lim and Nagarakatte's CASM-VERIFY [49], focusing on verifying cryptographic algorithm equivalence, translates the assembly into an internal domain-specific language for efficient equivalence checking. However, CASM-VERIFY requires manually crafted specifications of equivalent variables and its scope is limited to x86 instructions. Ming et al. introduced BinSim [57], a trace-based semantic binary diffing tool employing system call sliced segment equivalence checking. While effective for obfuscated binaries, BinSim's reliance on dynamic analysis requires the execution of both binaries under comparison, which may be impractical. Zou et al. introduced D-Helix [90], a generic decompiler testing framework that uses symbolic differentiation to detect semantic differences between the original binary and the recompiled decompiled output. While D-Helix effectively identifies decompiler inaccuracies, it focuses on testing decompilers rather than general-purpose binary equivalence checking and lacks the adaptive analysis approach that our tool provides.

Our work, VERIBIN, addresses these limitations by employing symbolic execution directly on binaries, and it focuses on verifying StA properties. We use SMT solvers to confirm equivalence for valid execution paths, ensuring functionality-preserving differences without relying on test cases or source code. Furthermore, VERIBIN offers adaptive verification to handle semantically equivalent changes that may violate the StA properties, addressing a gap in existing binary-level equivalence-checking techniques.

Binary Similarity. The problem of comparing a pair of binaries or, in general, binary similarity [38], has been explored before. BinHunt [36] uses graph isomorphism at the CFG level to identify differences between two binaries. APEG [21] computes the weakest pre-condition between a binary and its patched version to automatically identify the patched security issue and generate an exploit for it. Several machine learning-based approaches have also been developed to identify similar binaries [30], [33], [54], [73]. There are also dynamic analysis techniques [34], [76] that compare binaries based on their runtime characteristics, such as system call traces [20], [57], memory access patterns [86]. Our work is orthogonal to these approaches, as we focus on providing a more precise analysis of the impact of the patch-introduced observable behaviors, rather than a general binary similarity analysis. VERIBIN aims to formally prove certain StA properties of a patch using analysts' assistance when required. To this end, we use under-constrained symbolic execution (uc-symex) [68] to execute the original and patched function pairs, during which we verify our StA properties.

XI. CONCLUSION

This paper presents VERIBIN, a system capable of comparing a binary with its patched version to determine whether the patch is "Safe to Apply", *i.e.,* it does not introduce any modification that could potentially break the functionality of the original binary. Performing this binary-level analysis is challenging as it requires dealing both with scalability issues and the absence of semantic information, which is removed during the binary's compilation. Nevertheless, for unstripped binaries, our evaluation of 86 samples shows how VERIBIN yields an accuracy of 93.0% with 0% FPR in accurately characterizing patch behaviors, and a conservative accuracy of 66.7% with an FPR of 0% if considering timeout and memory exceeded cases as FN. For stripped binaries, VERIBIN achieves an accuracy of 89.4% with 0% FPR. Notably, this high level of accuracy demands only minimal configuration effort on the part of a human analyst. These results show how VERIBIN can be used effectively to automatically vet patched binaries provided by third parties, thereby speeding up patch deployment.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments which highly improved our paper. We would like to thank Fabio Pagani for his help in the early stages of this work and Michael Gordon from Aarno Labs for sharing with us the MicroPatch Bench dataset. This material is based on research sponsored by DARPA under contract number N6600120C4031. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] angr: a platform-agnostic binary analysis framework. https://github.com/angr/angr.

[2] Beaglebone black board. https://beagleboard.org/bone.

[3] Claripy: An abstraction layer for constraint solvers. https://github.com/angr/claripy.

[4] Flirt signatures. https://github.com/angr/flirt_signatures.

[5] Ida pro. https://hex-rays.com/ida-pro/.

[6] Xz utils. https://github.com/tukaani-project/xz.

[7] Use of goto in systems code. https://blog.regehr.org/archives/894, 2013.

[8] Assured Micropatching (amp). https://www.darpa.mil/program/assured-micropatching, 2020.

[9] Welcome to the era of vulnerability micropatching – 0patch. https://0patch.com, 2022.

[10] Cve-2024-3094 (xz utils backdoor). https://nvd.nist.gov/vuln/detail/CVE-2024-3094, 2024.

[11] Github repository of veribin. https://github.com/purseclab/VeriBin, 2024.

[12] John Admanski and Steve Howard. Autotest-testing the untestable. In *Proceedings of the Linux Symposium*, 2009.

[13] Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L Lawall, and Siau-Cheng Khoo. Semantic patch inference. In *Proceedings of the ACM International Conference on Automated Software Engineering (ASE)*, 2012.

[14] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, 2008.

[15] Ashish Arora, Ramayya Krishnan, Rahul Telang, and Yubao Yang. An empirical analysis of software vendors' patch release behavior: impact of vulnerability disclosure. *Information Systems Research*, 2010.

[16] Terrence August and Tunay I Tunca. Let the pirates patch? an economic analysis of software security patch restrictions. *Information Systems Research*, 2008.

[17] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[18] Gabriele Bavota. Mining unstructured data in software repositories: Current and future trends. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[19] David Binkley, Rob Capellini, L Ross Raszewski, and Christopher Smith. An implementation of and experiment with semantic differencing. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSME)*, 2001.

[20] Kristina Blokhin, Josh Saxe, and David Mentis. Malware similarity identification using call graph based system call subsequence features. In *Proceedings of the IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2013.

[21] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2008.

[22] Raymond P.L. Buse and Westley R. Weimer. Automatically documenting program changes. In *Proceedings of the ACM International Conference on Automated Software Engineering (ASE)*, 2010.

[23] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[24] Marco Castelluccio, Le An, and Foutse Khomh. An empirical study of patch uplift in rapid release development pipelines. *Empir Software Eng*, 2019.

[25] Yu Chen, Fengguang Wu, Kuanlong Yu, Lei Zhang, Yuheng Chen, Yang Yang, and Junjie Mao. Instant bug testing service for Linux kernel. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, 2013.

[26] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2017.

[27] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019.

[28] Kees Cook. Security bug lifetime. https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime/, 2016.

[29] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[30] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2019.

[31] Nesara Dissanayake, Mansooreh Zahedi, Asangi Jayatilaka, and Muhammad Ali Babar. Why, how and where of delays in software security patch management: An empirical investigation in the healthcare sector. In *Proceedings of the ACM on Human-Computer Interaction*, 2022.

[32] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[33] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.

[34] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2014.

[35] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. An abstract domain of uninterpreted functions. In *Proceedings of the international conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2016.

[36] Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the International Conference on Information and Communications Security (ICICS)*, 2008.

[37] Emanuel Giger, Martin Pinzger, and Harald C Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2011.

[38] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *Proceedings of the ACM Computing Surveys (CSUR)*, 2021.

[39] Xu He, Shu Wang, Pengbin Feng, Xinda Wang, Shiyu Sun, Qi Li, and Kun Sun. Bingo: Identifying security patches in binary code with graph representation learning. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2024.

[40] H. Huang, A. Zhou, M. Payer, and C. Zhang. Everything is good for something: Counterexample-guided directed fuzzing via likely invariant inference. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2024.

[41] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: finding unpatched code clones in entire os distributions. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2012.

[42] Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. Evocatio: Conjuring bug capabilities from a single poc. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[43] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[44] Yuan Kang, Baishakhi Ray, and Suman Jana. Apex: Automated inference of error specifications for c apis. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, 2016.

[45] Aarno Labs. Micropatch bench. https://github.com/Aarno-Labs/micropatch-bench.

[46] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: a language-agnostic semantic diff tool for imperative programs. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2012.

[47] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[48] Hongyi Li, Daojing He, Xiaogang Zhu, and Sammy Chan. P1ovd: Patch-based 1-day out-of-bounds vulnerabilities detection tool for downstream binaries. *Electronics*, 2022.

[49] Jay P. Lim and Santosh Nagarakatte. Automatic equivalence checking for assembly implementations of cryptography libraries. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019.

[50] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Automatically identifying security checks for detecting kernel semantic bugs. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2019.

[51] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. Spider: Enabling fast patch propagation in related software repositories. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2020.

[52] Viktor Malík and Tomáš Vojnar. Automatically Checking Semantic Equivalence between Versions of Large-Scale C Projects. In *Proceedings of the IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021.

[53] Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, 2013.

[54] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *Proceedings of the international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2019.

[55] Miles A McQueen, Trevor A McQueen, Wayne F Boyer, and May R Chaffin. Empirical estimates and observations of 0day vulnerabilities. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*, 2009.

[56] Xiaozhu Meng and Barton P Miller. Binary code is not easy. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[57] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2017.

[58] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In *Proceedings of the international conference on Computer Aided Verification (CAV)*, 2007.

[59] Felicia M Nicastro. *Security patch management*. CRC Press, 2011.

[60] Robert Nieuwenhuis and Albert Oliveras. On sat modulo theories and optimization problems. In *Proceedings of the international conference on Theory and Applications of Satisfiability Testing (SAT)*, 2006.

[61] Mehrdad Nurolahzade, Seyed Mehdi Nasehi, Shahedul Huq Khandkar, and Shreya Rawal. The role of patch review in software evolution: an analysis of the mozilla firefox. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, 2009.

[62] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2020.

[63] Aditya Pakki and Kangjie Lu. Exaggerated error handling hurts! an in-depth study and context-aware detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[64] Mathias Payer and Thomas R Gross. Hot-patching a web server: A case study of asap code repair. In *Proceedings of the Annual Conference on Privacy, Security and Trust (PST)*, 2013.

[65] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 1dvul: Discovering 1-day vulnerabilities through binary patches. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.

[66] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[67] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSME)*, 2004.

[68] David A Ramos and Dawson Engler. {Under-Constrained} symbolic execution: Correctness checking for real code. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2015.

[69] Silvio Ranise and Cesare Tinelli. Satisfiability modulo theories. *Trends and Controversies-IEEE Intelligent Systems Magazine*, 2006.

[70] Sarah Rastkar and Gail C. Murphy. Why did this code change? In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013.

[71] Pemma Reiter, Hui Jun Tay, Westley Weimer, Adam Doupé, Ruoyu Wang, and Stephanie Forrest. Automatically Mitigating Vulnerabilities in x86 Binary Programs via Partially Recompilable Decompilation. *arXiv preprint arXiv:2202.12336*, 2023.

[72] Eddie Antonio Santos and Abram Hindle. Judging a commit by its cover: correlating commit message entropy with build status on travis-ci. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2016.

[73] Noam Shalev and Nimrod Partush. Binary similarity detection using machine learning. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)*, 2018.

[74] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[75] Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, 2017.

[76] Shuai Wang and Dinghao Wu. In-memory fuzzing for binary code similarity analysis. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[77] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. Patchdb: A large-scale security patch dataset. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.

[78] Yuxin Wang, Adam Welc, Lazaro Clapp, and Lingchao Chen. Last diff analyzer: Multi-language automated approver for behavior-preserving code revisions. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023.

[79] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.

[80] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: Recovering links between bugs and changes. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, 2011.

[81] Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. Tracking patches for open source software vulnerabilities. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

[82] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic hot patch generation for android kernels. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.

[83] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2018.

[84] Xiantao Zhang, Xiao Zheng, Zhi Wang, Qi Li, Junkang Fu, Yang Zhang, and Yibin Shen. Fast and scalable VMM live upgrade in large cloud infrastructure. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[85] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2021.

[86] Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. Patchscope: Memory object centric patch diffing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[87] Lei Zhou, Fengwei Zhang, Jinghui Liao, Zhengyu Ning, Jidong Xiao, Kevin Leach, Westley Weimer, and Guojun Wang. KShot: Live kernel patching with SMM and SGX. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

[88] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*, 2017.

[89] Xiaogang Zhu and Marcel Böhme. Regression Greybox Fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[90] Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave (Jing) Tian. D-Helix: A generic decompiler testing framework using symbolic differentiation. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2024.

[91] Zynamics. Bindiff. https://www.zynamics.com/bindiff.html.

APPENDIX A
ABLATION STUDY

TABLE VII: Ablation Study Results: Impact of MPP and CIOC.

Experiment	**ACC (%)**	**FPR (%)**	**Avg.T_a (s)**	**Avg.T_c.D (s)**
E1: MPP + CIOC	93.0	0.0	1,293.82	47.13
E3: No MPP + CIOC	93.0	0.0	1,300.46	194.28
E4: MPP + No CIOC	81.4	0.0	1,188.31	46.16

Metrics: ACC = Accuracy; FPR = False Positive Rate; Avg. T_a = Average time for VERIBIN to finish, excluding preprocessing; Avg. T_c.D = Average time for verifying StA properties in divisible cases only.

As mentioned earlier in Section VII, we conduct an ablation study to measure the contribution of MPPs and CIOCs on unstripped datasets. On average, using MPPs improved the verification time by 75%, while using CIOCs improved the accuracy by 12%. Table VII summarizes the results of our ablation study for MPP and CIOC. A detailed result table containing the individual patch outcomes can be found in our GitHub repository [11].

In this study, we compare the results that VERIBIN achieves in its baseline configuration (E1), with the results obtained in the following two configurations (E3 and E4):

Optimization for divisible functions (E3). Recall that in Section V-D we discuss divisible and indivisible functions and the possibility of directly comparing all MPPs when the functions are divisible. To quantify the advantages given by comparing MPPs in divisible functions, in E3, we disable this optimization and let VERIBIN merge all the paths without attempting to find MPPs.

According to the data presented in Table VII, using configuration E3, the accuracy remains the same, which means the strategies used by VERIBIN for divisible and indivisible functions are equivalent. However, overall, using MPPs detection makes VERIBIN faster for divisible cases, as the average time for divisible cases improves 75% from 194.28 seconds (in E3) to 47.13 seconds (in E1).

The benefit of using this strategy is particularly significant in a few specific samples, such as patch #67 (for which using MPPs detection saves 30 minutes of checking time). We also notice there are cases when the run time in E1 is greater than that in E3. Primarily, this happens because VERIBIN attempts to find MPPs for 10 minutes. Therefore, in the cases where the functions are indivisible, finding MPPs will fail, and 10 minutes will be wasted.

Handling Compiler-Introduced Offset Changes (CIOCs) (E4). Referring to Section V-D1, before comparing two constraints or values, we identify and filter out CIOCs so that VERIBIN will not consider these compiler-introduced changes. To quantify the effectiveness of this step, in E3, we do not filter out CIOCs before comparing constraints and values.

In Table VII, the ACC dropped from 93.0% (in E1) to 81.4% (in E4), since VERIBIN incorrectly considered CIOCs as in-equivalences. We noticed that the total run time decreased, which happens for two main reasons: (1) We remove the analysis step for identifying CIOCs, and (2) the SMT solver generally terminates earlier when $v_o \equiv v_p$ is False, which happens more commonly in E4.

APPENDIX B
EFFECTIVENESS OF EEP DETECTION HEURISTICS.

TABLE VIII: Effectiveness of EEP detection heuristics. Complete EEP Detection: the number of cases in which the heuristic(s) can identify all the patch-related EEPs.

Heuristic(s)	Complete EEP Detection
A	13 (33.3%)
A + B	26 (71.8%)
A + B + C	32 (82.1%)

Across 86 binaries (D1+D2+D3), we observed 37 cases (*i.e.,* original and patched binaries) that contain patch-related EEPs (*i.e.,* the patch either introduces a new EEP or jumps to an existing EEP). Detecting these EEPs is necessary for VERIBIN to get accurate outcomes. Our heuristic-based method (explained in Section V-C) automatically detected such EEPs in 32 out of 37 cases. Overall, our heuristics flagged 2,676 EEPs with an 87.9% true positive rate.

Table VIII provides further details about each heuristic's contribution. Note that for Heuristic A, we determine the hyperparameter ratio as 0.8 based on an analysis of a subset of 20 binaries from D1. During the execution of VERIBIN in D1, D2, and D3, we keep track of the EEPs identified by each heuristic. Subsequently, two of the authors manually verify the accuracy of the detected paths by examining both the source code and the CFG. We calculate the number of cases in which the heuristic(s) can identify all the patch-related EEPs (marked as 'Complete EEP Detection'). As more heuristics are combined—from Heuristic A alone to Heuristics A, B, and C together—the number of complete EEP detections increases, indicating enhanced coverage.

APPENDIX C
COMPARISON OF APPLICABILITY AND ACCURACY BETWEEN SPIDER AND VERIBIN

Figure 4 presents a Venn diagram comparing the applicability and accuracy for patch verification between SPIDER and VERIBIN, across 125 patches from D1, D2, and D3. SPIDER supports 79 patches (63.2%), and correctly classifies 49 patches according to GT_SPIDER, with 62.0% ACC and 27.9% FPR. In contrast, VERIBIN supports 86 patches (68.8%), and correctly classifies 83 patches according to GT_Strict, with 96.5% ACC and 0% FPR. For 51 patches, both SPIDER and VERIBIN are able to verify the patches, with 33 patches correctly classified by both tools.

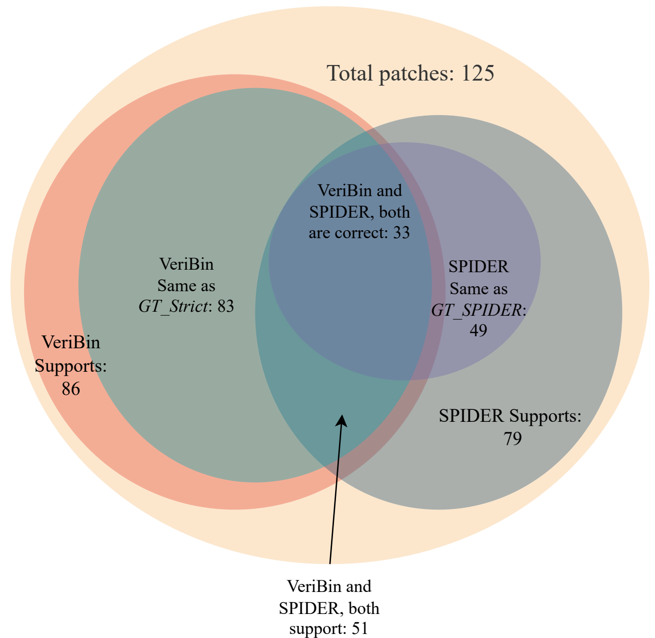

Fig. 4: Comparison of applicability and accuracy between SPIDER and VERIBIN, across 125 patches from D1, D2, and D3.