Automatically Mitigating Vulnerabilities in x86 Binary Programs via Partially Recompilable Decompilation

Pemma Reiter
pdreiter@asu.edu
Arizona State University
USA

Hui Jun Tay
htay2@asu.edu
Arizona State University
USA

Westley Weimer
weimerw@umich.edu
University of Michigan
USA

Adam Doupé
doupe@asu.edu
Arizona State University
USA

Ruoyu Wang
fishw@asu.edu
Arizona State University
USA

Stephanie Forrest
steph@asu.edu
Arizona State University
USA

ABSTRACT

When vulnerabilities are discovered after software is deployed, source code is often unavailable, and binary patching may be required to mitigate the vulnerability. However, manually patching binaries is time-consuming, requires significant expertise, and doesn't scale to the rate at which new vulnerabilities are discovered. To address these problems, we introduce Partially Recompilable Decompilation (PRD), which extracts and decompiles suspect binary functions to source where they can be patched or analyzed, applies transformations to enable recompilation of these functions (*partial recompilation*), then employs binary rewriting techniques to create a patched binary. Although decompilation and recompilation do not universally apply, PRD's fault localization identifies a function subset that is small enough to admit decompilation and large enough to address many vulnerabilities. Our approach succeeds because decompilation is limited to a few functions and lifting facilitates analysis and repair.

To demonstrate the scalability of PRD, we evaluate it in the context of a fully automated end-to-end scenario that relies on source-level Automated Program Repair (APR) methods to mitigate the vulnerabilities. We also evaluate PRD in the context of human-generated source-level repairs. In the end-to-end experiment, PRD produced test-equivalent binaries in 84% of cases; and the patched binaries incur no significant run-time overhead. When combined with APR tools and evaluated on the DARPA Cyber Grand Challenge (CGC) benchmarks, PRD achieved similar success rates as the winning CGC entries, while presenting repairs as source-level patches which can be reviewed by humans; In some cases, PRD finds higher-quality mitigations than those produced by top CGC teams. We also demonstrate that PRD successfully extends to real-world binaries and binaries that are produced from languages other than C.

1 INTRODUCTION

Mitigating security vulnerabilities is challenging when vendor support is not available or when recompiling from the entire source code is infeasible. In these cases, vulnerabilities must be addressed at the binary level, which is tedious, error-prone, requires expertise, and is not scalable to the large number of vulnerabilities that plague today's software. Although automated program repair (APR) methods are maturing quickly [48], the most popular methods operate only on source code [49]. At the same time, automated approaches that lift, patch, recompile, and rewrite binary code are not available, as today's decompilation tools also suffer from scalability issues [38] or focus on readability rather than recompilability [18, 65, 81], producing results that may be inaccurate or not recompilable when applied to an entire binary program [38].

To provide the benefits of source-level analyses, repairs, or other transformations at the binary level, this paper proposes *Partially Recompilable Decompilation* (PRD). The PRD approach is based on three insights:

(1) *partial localization*: A relevant subset of binary functions can be identified and successfully decompiled by off-the-shelf tools, even when those tools fail for full binaries. No restrictions are placed on the grammar or optimization level used to produce the binary.
(2) *recompilation and rewriting*: A small number of modified functions can be patched into an existing binary, even across multiple languages and respecting performance constraints.
(3) *informative decompilation*: A decompiled subset of a binary can provide sufficient context for off-the-shelf source-level analyses and transformations, even when those analyses and transformations operate only on source.

PRD uses partial decompilation to generate high-level source code for a small number of relevant functions (e.g., those that are implicated by the vulnerability), supports source-level analysis and editing of the decompiled code (either manually or with automated tools), and supports partial recompilation and binary rewriting to generate a patched binary with recompiled content.

To identify a set of suspicious functions (partial localization), we apply coarse-grained fault localization (CGFL) directly to the binary. For recompilation and rewriting, we apply source code and binary transformations to accommodate decompiled functions, in conjunction with a standard binary rewriting technique, the detour [21]. Many source-level analyses can be applied to the partially-decompiled code, as long as they do not require access to the full program. Here, we focus on mitigating vulnerabilities and other bugs as an indicative application, using both automated (APR) and manual program repair methods. We use *automated Binary REpair using PArtially REcompilable Decompilation* (BinREPARED) to refer to PRD when it is enhanced to operate with off-the-shelf program repair tools.

Our evaluation considers multiple benchmarks, including C and C++ programs taken from the DARPA Cyber Grand Challenge

(CGC) and from the MITRE CVE List [1], multiple, independently developed and evaluated APR tools (Prophet and GenProg), and off-the-shelf tools (e.g., Hex-Rays [9] and Ghidra [8] decompilers).

To summarize, the main contributions of this paper are:

- Partially Recompilable Decompilation (PRD): A novel technique for applying source-level analyses to binary executables based on partial localization, informative decompilation, and general recompilation.
- An end-to-end evaluation of BinREPARED's ability to apply off-the-shelf source-based APR tools to mitigate vulnerabilities in CGC challenge binaries. We find that both APR tools, when used with PRD, mitigate the tested vulnerabilities with success rates that match and sometimes exceed the same APR tools operating on the full source. Our tools collectively mitigate 85 of 148 unique proof of vulnerabilities (20 challenge binaries that the winning cyber-reasoning system did not patch) and provide human-readable references that developers can analyze and amend.
- An empirical validation of our assumptions and an evaluation of the individual techniques. We find that 79.5% of individual functions are successfully decompiled and then recompiled (while only 2% of the full C binaries succeeded), and PRD produces test-equivalent binaries for 84% of CGFL-generated PRD binaries. We also find no significant run-time overhead and demonstrate multi-language generality.
- An evaluation of two CVEs that demonstrate language (C and C++) and compiler (Clang and G++) generality.

To further open and reproducible science, our prototypes, the curated benchmark dataset, and all of our experimental results are available at *git@github.com:pdreiter/FuncRepair.git*.

2 BACKGROUND

We briefly describe the techniques PRD uses for analyzing and manipulating binary content, as well as those BinREPARED employs to localize and repair faults.

2.1 Binary Decompilation and Rewriting

A binary program, or *binary* for short, refers to a structured executable file composed of encoded binary instructions, *machine code*. *Disassembling* is the process of lifting machine code to assembly instructions. *Decompilation* is the general process of lifting lower-level abstractions (e.g., assembly instructions) to high-level representations, producing source code or source-code-like representations. Since much information, such as control flow structures, function prototypes, variable names, and variable types, is lost during compiling, decompilers must infer the lost information [25, 31, 54, 81]. Such inference is unsound, often leading to unreadable decompilation output, incorrect results, or decompilation failures.

Binary rewriting alters a compiled binary file directly while retaining the ability to execute the file [32]. To maximize compatibility, PRD uses a binary rewriting strategy that appends recompiled binary content to an existing ELF binary, overwriting binary content to *detour* execution to the appended content. This does not interfere with how the operating system (OS) loads the binary. Since the ELF loader will not change the appended binary content, this strategy brings several obstacles: its symbols will not be relocated or resolved; its global constructors or destructors will not be run; and its static data sections will not be initialized. To address these obstacles, PRD generates position- and execution- independent binary content (see Section 4).

2.2 Fault Localization

Fault localization (FL) methods pinpoint the likely locations of a bug by analyzing a program, either dynamically or statically. One common approach is Spectrum-Based Fault Localization (SBFL), in which program spectra (characteristics) are obtained through dynamic and static analysis and used to implicate code regions. Spectra are not limited to code coverage but can include data- or information-flow [19, 45], function call sequences [83], or program counter samples [63]. SBFL combines the spectra to produce suspiciousness scores and a ranking, or risk evaluation in a formula [13]. In addition, SBFL does not require content, such as historical development information, which is used by other methods [26, 69], as this is not available without source code.

2.3 Automated Program Repair

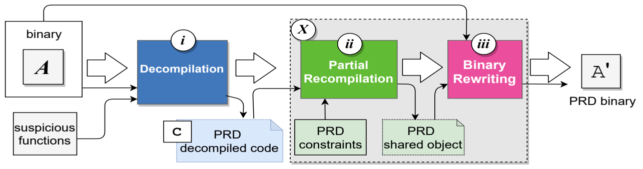
Automated program repair (APR) techniques generate patches for defects in software with minimal or no human intervention [24]. There are many popular APR techniques (see Gazzola *et al.* for a survey [23]). A key distinction is between those that rely on test suites to validate repair correctness (often called *search-based* or *test-based*) and those that rely on formal semantics (often called *semantics-based*). In this work, we use two independently developed tools that do not natively support binary repairs, Prophet [40] and GenProg [34, 35]. We also report results for GenProg's deterministic variant, "AE" [77]. These tools all use mutation operators to generate program variants, each of which must be compiled and executed against tests to determine if it is a plausible repair. Each tool has strengths and weaknesses, and all test-based tools are susceptible to overfitting to their tests. However, this is less of an issue in a setting like the CGC, where mitigating a vulnerability takes priority over correctly repairing its root cause.

3 MOTIVATING EXAMPLE

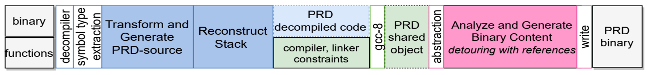
To motivate and illustrate our approach, consider a security engineer who is responsible for *KPRCA_00018* (*Square_Rabbit*) from the DARPA CGC dataset. This casino-inspired game has an integer

```
int cgc_split() {
        int v0; int result; char v2; card_t *v3; card_t *v4;
        squarerabbit_t *split_srabbit; squarerabbit_t *srabbit;
        int i; i = 0;
        for (srabbit = g_srabbit; srabbit->player_finished && i <
             cgc_split_len(); srabbit = &split_hand[v0])
            v0 = i++;
        if (!srabbit->double_or_split || !cgc_can_split(srabbit))
            return -1;
        v2 = g_srabbit->split_len;
/*BUG*/g_srabbit->split_len = v2 + 1;
/*BUG*/if ( v2 > 1 )
/*BUG*/     return -1;
```

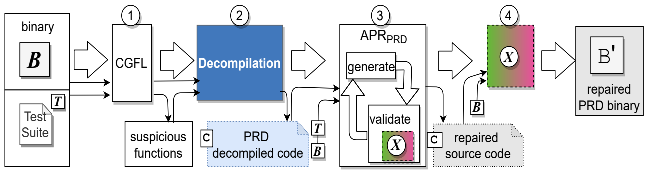
Listing 1: Decompiled code for *KPRCA_00018* function, *cgc_split*, generated by the Hex-Rays decompiler. The low readability of decompilation result does not limit APR tools.



(a) PRD consists of three stages: *(i)* decompilation (`blue`), *(ii)* partial recompilation (`green`), and *(iii)* binary rewriting (`fuschia`). PRD requires only a function and the original binary, A, to generate PRD decompiled code and a detoured binary. A stage's (`color`) is consistent across all sub-figures.



(b) An explication of PRD in terms of external tools (white), internal processes, and outputs.



(c) The four stages of BinREPARED: (1) Coarse-Grained Fault Localization (CGFL); (2) decompilation (PRD); (3) source-level repair; (4) partial recompilation and binary rewriting (PRD). The BinREPARED meta-APR tool uses a compatible PRD-enabled APR method for repair.

Figure 1: Stages of PRD and BinREPARED.

overflow vulnerability, which a player can use to crash the program. Our engineer needs to prevent crashes, but the software is no longer supported by the vendor, the source code is not available, and direct binary fault localization and patching, which are costly and error-prone [27], are not feasible.

We use *automated Binary REpair using PArtially REcompilable Decompilation* (BinREPARED) to address this problem. As Figure 1c illustrates, we assume that the buggy binary executable has a test suite consisting of functional (*100* positive) and bug-inducing (*one* negative) tests. Our fault localization has implicated a set of suspicious functions that includes the vulnerable *cgc_split* function. Finally, we assume Hex-Rays decompiler [9] as our off-the-shelf decompiler. Our goal is to generate a patched binary that preserves the expected functionality and repairs the vulnerability.

Our engineer may or may not be able to repair the vulnerability manually, so we further assume that she has access to a test-based APR tool, which can suggest a repair. First, we localize the likely source of the problem to a few suspicious functions in the binary (fault localization) and decompile only those functions. Listing 1 shows Hex-Rays decompiled code for the buggy *cgc_split* function from binary. In this example, the bug appears on lifted lines 13–15 where `g_srabbit->split_len` is incorrectly incremented, resulting in the integer overflow on return. Next, the bug is repaired, either manually by the engineer or automatically using an APR tool. This requires that candidate patches can be evaluated by executing them in conjunction with the original binary. This execution is enabled by PRD extensions to the decompiled source code (Figure 1a). PRD's *partial recompilation* and *binary rewriting* supports execution by composing the original binary content with the PRD decompiled code using *detouring* (Figure 2). Although certain program structures, such as function prototypes, undefined symbols, and linking artifacts complicate decompilation and recompilation [17, 38, 62, 66], we find that restricting attention to a small set of functions sidesteps such problems in practice. We also validate the PRD binary by comparing test outputs to the original.

In our example, APR operates on the source code in Listing 1 but evaluates candidate repairs in the full binary context using PRD, then identifies the developer-equivalent patch (e.g., by moving line 13 after lines 14–15). We apply PRD's recompilation and binary rewriting to these source-level patches, derived from the PRD decompiled code, to produce a patched binary. The patched binary detours to the repaired code whenever *cgc_split* is called. Altogether, we obtain the ease and benefit of source-level APR although applied to a binary.

4 PARTIALLY RECOMPILABLE DECOMPILATION

The goal of PRD is to allow vulnerabilities to be mitigated at the source level even when only the binary is available. As a first check, PRD generated binaries should be test-equivalent to the original binary when no changes are applied to the source. PRD takes as input a small set of binary functions (identified by fault localization) and consists of three interdependent stages: decompilation, partial recompilation, and binary rewriting, illustrated then detailed in Figures 1a-1b. The output is a single binary executable composed of the original binary rewritten with stack manipulation and executional detouring to the recompiled function content. Our prototype implementation operates on 32-bit Linux ELF executables (including those compiled by GCC, G++ and Clang) compatible with System V Application Binary Interface [28]. With additional engineering our approach can be extended to 64-bit as well as stripped binaries. In the rest of this section, we discuss key PRD aspects: Fault Localization, Decompilation, Partial Recompilation and Binary Rewriting.

4.1 Coarse-Grained Fault Localization (CGFL)

The three stages of PRD operate on a small number of *implicated functions*, depending on the subsequent source-level analysis. In our case, the vulnerable functions should be implicated. However, our binary setting provides different assumptions from most fault localization (FL) methods (Section 2.2)), leading to five requirements: (r1.1) does not require source code or the ability to recompile, (r1.2) prioritizes functions for decompilation, (r1.3) minimizes run-time overhead, (r1.4) avoids functions which cannot support detouring, and ultimately (r1.5) identifies vulnerable functions. We refer to our FL approach as *coarse-grained fault localization* (CGFL), which applies FL to binaries and identifies the set of suspicious functions targeted for decompilation. After decompilation, additional finer-grained (traditional) FL may be required to pinpoint suspicious statements or expressions.

To localize vulnerabilities to a set of binary functions, we generated spectra from function coverage data of a binary's test runs using *Valgrind*'s [52] *callgrind* (satisfying r1.1–r1.3: it does not require source, identifies functions, and has overhead within 10× the original). To meet requirement r1.4, we eliminate functions smaller than 45 bytes (supporting detours, see Section 4.2.1). Algorithmically, we use a Rank Aggregation Fault Localization (RAFL) [50] method which combines several state-of-the-art SBFL metrics: Tarantula [30], Ochiai [29], op^2 [51], Barinel [14], and Dstar (star=2) [79]. RAFL consolidates ordered lists based on weighted ranks, identifying the top-$\mathcal{K}$ (35%) suspicious functions [37] (satisfying r1.5: in our experiments this implicates vulnerable functions in the top 35% more than 85% of the time).

4.2 Decompilation

Decompilation output can be produced by automated decompilers, human experts, or a combination (e.g., experts editing decompiler output). While *full decompilation* refers to the complete decompilation of a binary, we say a decompilation is *partial* when only certain functions in the binary are decompiled. This design choice mitigates some of the weaknesses of current decompilers. The vulnerable function(s) and dependencies are extracted from the binary and decompiled to source, including a new entry function and type, prototype, and object definitions and declarations compatible with partial recompilation. We consider fully automatic and partial decompilation.

4.2.1 Constraints and the Detour Entry Function. Because our decompiled code will ultimately be recompiled and integrated into an existing binary, we face additional constraints beyond those found in standard decompilation. We use binary rewriting to connect the recompiled functions with the rest of the binary via detouring rather than standard ELF loading and linking. This leads to four key requirements for PRD decompiled code: (r2.1) does not use dynamic linking, (r2.2) does not require global or static initialization, (r2.3) does not register constructors or destructors, and (r2.4) does not duplicate global symbols in detoured content.

To satisfy these constraints, we focus on the decompiled content's dependencies (symbols that the decompiled code requires) and generate a *detour entry function* to manage them. We use *detouring with references*, which passes in references to relevant external symbols so that decompiled and recompiled code can make use of them. Effectively, this mechanism enables execution similar to callbacks, but includes variables in addition to functions.

To obtain external symbols, we use information from the original binary for all required symbols, types, and declarations. During binary rewriting, detouring with references manipulates the stack according to the calling convention to pass in references to these external symbols. By assigning these references to variables (function or object references) we declare in decompiled source code and dereference as needed, the recompiled binary content is guaranteed to execute correctly. All external references are managed in the detour entry function, leaving the prototype of the decompiled function consistent with the original binary.

Applying detouring with references means that the original binary's function call and the detour entry function's prototype has diverged. This has two implications: (i1) our detour entry function is not compatible with the original binary's function call; and (i2) the stack state is not consistent upon return from our detour entry function. While PRD's binary rewriting phase handles (i1) (Section 4.4), (i2) is addressed during PRD's decompilation, when stack-correcting inline assembly is added before the entry function's return.

This approach satisfies all our requirements: r2.1 (static linking only), r2.2–r2.3 (the detour entry function assigns references, rather than relegating that work to initializers or constructors), and r2.4 (the detour code references symbols from the original binary).

4.2.2 Decompilation implementation. Our prototype uses the Hex-Rays IDA Pro tool [9]: Hex-Rays generates an initial decompilation and a custom IDAPython batch script (Python3) obtains corresponding local typedefs, functions, and struct definitions. We then apply transformations to support PRD: (t1) substitute common primitives such as `_DWORD` and `_BYTE` and any Hex-Rays definitions, (t2) resolve a definition order for structs and local types, (t3) identify external symbols and generate the required detour entry function definitions. To support detouring with references, t3 also resolves the minimum set of dependent symbols for decompiled functions' calltrees, efficiently managing stack use. These rule-based transformations are a best-effort heuristic to produce an informative decompilation. To generate the inline assembly that reconstructs the stack required for detouring with references, we (1) analyze the initial decompiled PRD source, (2) generate the appropriate in-line assembly to reconstruct the stack on detour exit, and (3) insert this assembly in the detour entry function's source code.

For simplicity, we refer to this final output (the informative, lifted decompilation with detour entry and reconstructed stack) as the *PRD decompiled source*.

4.3 Partial recompilation

Partial recompilation must be able to independently recompile high-level source code and ensure its correct execution in the context of the patched binary. We focus on two key requirements. First, because our binary rewriting strategy appends any new binary content, (r3.1) it must operate even if the memory base address of new content is not known in advance. Second, because new binary content may be executed at any point during run-time, (r3.2) it must operate regardless of the state of the program's library (e.g., *glibc*) linking, relocation table, dynamic symbols, and global object constructors or destructors. To satisfy these requirements, partial recompilation creates *position-* and *execution-independent* code.

Although most compilers support position-independent code, support for simultaneous static linking of external library objects and positionally independent code in a single executable is relatively new. We use the `-static-pie` flag, first introduced in GNU version 8.4.0 [12] to generate a statically linked and position-independent shared object.[1] To simplify binary rewriting, our prototype also constrains the linker to place all sections in a single segment.

Execution independence is achieved primarily through the generation of highly constrained source code specified by *detouring with references* in PRD's decompilation stage, and then later resolved during binary rewriting. External libraries may require program startup initialization and tasks that complicate execution independence (e.g., *glibc* has a strict dependence on execution order that we cannot guarantee). Our prototype implementation takes a pessimistic approach to decompiled content, assuming that standard *glibc* functions may be required, and uses *dietlibc*, a small-footprint alternative [6].

Our approach satisfies both requirements: r3.1 (with position independence) and r3.2 (with execution independence).

[1]Clang strictly interprets the command-line arguments `-shared` and `-static-pie`, such that `-static-pie` takes precedence.

4.4 Binary Rewriting

Binary Rewriting composes the original binary and recompiled functions into a single binary that executes correctly, replacing the vulnerable function with recompiled content. We implemented our custom binary rewriting prototype to extract, add, and manipulate binary content using LIEF [74]. Because maintaining execution independence is more important than minimizing the binary size in our use case, we append the recompiled shared object content—the detour destination—as a new segment of the original binary. We use the updated symbol, section, and segment information to calculate relative addressing for the detour destination and for any required global references. Figure 2 illustrates the general case of such detouring with references. Finally, to satisfy implication (i1) (Section 4.2.1), we insert binary instructions to manipulate the stack, changing the effective function call to align with the detour entry prototype.

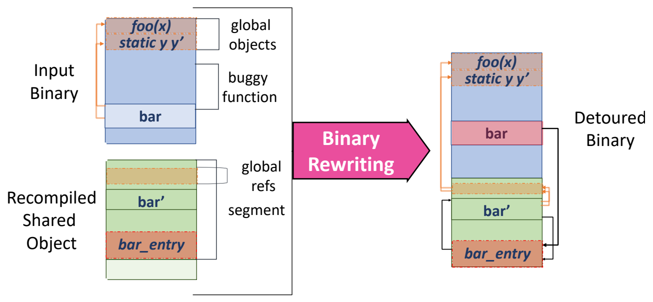


Figure 2: *Detouring with References.* The left identifies important input features for Binary Rewriting, the right shows its output. Symbol resolution is indicated by orange arrows, change in control flow by black arrows.

5 EXPERIMENTAL SETUP

PRD consists of several components, which we first independently and collectively in an end-to-end fully automated scenario. Specifically, our evaluation addresses the following questions:

RQ1. Decompilation: What fraction of decompiled programs and functions result in recompilable code?
RQ2. Partial recompilation: What fraction of decompiled code can be recompiled to be test-case equivalent to the original?
RQ3. BinREPARED: In an end-to-end scenario, how well does BinREPARED automatically mitigate vulnerabilities?
RQ4. Components: How effective is CGFL? When decompilation succeeds is the content usable and recompilable?
RQ5. Generality: Does PRD generalize to real-world defects, other languages, and performance constraints?

In this section, we describe our experimental setup to address these questions. Because we evaluate each PRD stage independently and in an end-to-end context, we use multiple datasets.

5.1 Benchmarks and Datasets

Our three benchmark datasets are: (1) the DARPA Cyber Grand Challenge C binaries ("CGC-C", 100 binaries, C language); (2) the DARPA Cyber Grand Challenge C++ binaries ("CGC-C++", 10 binaries, C++ language); and (3) case studies of two real-world programs with known vulnerabilities ("Case Study", 2 binaries, C/C++).

CVE	Vulnerability	Program (Release)	Comp. (Lang.)	LOC
CVE-2021-30472	stack-based buffer overflow	podofopdfinfo (PoDoFo-0.9.7)	g++ (C++)	47,414
CVE-2021-3496	heap-based buffer overflow	jhead (jhead-3.0.6)	Clang (C)	4,203

Table 1: "Case Study": Real-World Vulnerabilities.

5.1.1 CGC-C. At the binary level, the CGC dataset is a compelling example of the use cases we envision, where it is crucial to address a vulnerability quickly, and support or source may not be available. The DARPA 2016 Cyber Grand Challenge (CGC) [4] provides a dataset of binaries that each contain realistic vulnerabilities, with a testing framework [5]. We derived our dataset from a *Linux* variant of the CGC *challenge binaries* (CBs), *cb-multios* [75]. We verified all CGC CBs using a robust variant of the testing environment and found that 110 CGC CBs (100 from C source, 10 from C++) had at least one negative test that failed and nine positive tests that passed, averaging 1.7 negative tests and 100.4 positive tests each.

For our evaluations, we consider two features of each CB: *target*, a subset of functions targeted for decompilation, and *scenario*, the vulnerability scenarios to repair. When evaluating the vulnerable set of functions, our 100 C-based CBs contain a total of 190 applicable targets (some CBs have multiple vulnerable functions). When evaluating defects, these same CBs contain a total of 157 defect scenarios (some CBs have multiple vulnerabilities).

5.1.2 CGC-C++. There are 10 C++ source programs and binaries in the DARPA CGC Dataset (averaging 1,138 LOC).

5.1.3 Case Study: Real-World Vulnerabilities. Finally, we also consider two real-world programs, podofopdfinfo (*PoDoFo-0.9.7* [11]) and jhead (*jhead-3.0.6* [10]), not specifically curated for automated repair or binary analysis, which have associated public security vulnerabilities (CVEs). These programs fit the minimum PRD requirement (the vulnerable methods are local to the binary and not resolved dynamically). Table 1 presents relevant information for each CVE vulnerability and corresponding program.

5.2 External Tools

PRD uses Hex-Rays IDA Pro 7.5 SP2 (Hex-Rays) [9], GCC 8.4.1, and `dietlibc`. When handling C++, we augment Hex-Rays with the Ghidra SRE Public 10.0.1 Release [8]. We use Valgrind for coarse-grained fault localization.

5.3 APR Tools

To study BinREPARED's ability to mitigate vulnerabilities automatically, we selected one APR tool that uses the CIL infrastructure (GenProg version 3.2) and one based on Clang (Prophet version 0.1). In Figure 1c we use the label APR_{PRD} but for clarity, we refer to the tool name (GenProg or Prophet).

When evaluating program variants, GenProg replaces standard compilation with PRD to correctly generate the combined and detoured binary file. Prophet, which replaces the compiler with custom scripts, relies on runtime environmental variables and dynamic libraries. We address this with a PRD-compatible build infrastructure. Ultimately, PRD operates seamlessly with both tools.

For Prophet, we used the default search parameters with `profile` as its localizer strategy. For GenProg, we consider these search strategies: the default genetic algorithm ("GA"), a deterministic search option focused more on static analysis ("AE") [77], and a search of single-edit repairs ("single-edit").

6 EMPIRICAL EVALUATION

6.1 RQ1: Decompilation

"What fraction of decompiled programs and functions result in recompilable code?" We studied all 100 C-based and 10 C++-based CBs from CGC. Placing no restrictions on how the binary is compiled or the original source code grammar, we considered each function independently in all CBs and asked how many could be decompiled at all by Hex-Rays. We had 9,876 total decompile targets of which 4,330 had unique names (3,119 were C-functions).

Only two of the CBs (2/110) could be decompiled completely (all associated functions), even when they were augmented with heuristics. Those two C-based CBs, `Palindrome` and `Palindrome2`, each contain only four functions. This result validates one of our assumptions.

However, 2,479 of the 3,119 C-functions (79.5%), when considered separately and independently, successfully generated recompilable decompiled output. For C++ binaries, the result is 129 of 1392 functions (9.26%). The vast majority of C-based CBs (86) had more than 75% of their local functions decompile successfully.

We find that only 2% of C-based binaries can be fully decompiled, but 79.5% of individually decompiled functions can be recompiled. This strongly supports our insight to use *partial*, rather than full decompilation.

6.2 RQ2: Partial Recompilation

"What fraction of decompiled code can be recompiled to be test-case equivalent to the original?" We evaluated this in two ways: the set of all functions for 110 CGC CBs (same 9,876 targets as Section 6.1) and a focused subset of developer-specified vulnerable functions for 100 C-based CBs (190 targets). In both cases, we applied PRD to individual functions, first generating and recompiling PRD decompiled code, then rewriting the binary to use the recompiled content (Section 4) and evaluating the resulting PRD binary against the binary's test-cases. The second evaluation includes an analysis of the dynamic failures for the PRD binaries which fail test-case equivalency. Note that (1) we do not limit the grammar of the original binary's source code: our targets include structs unions, arrays, floating points, and pointers; (2) we do not limit how the original program was optimized; (3) no APR transformations are applied (the identity transformation).

Table 2 shows the results for the first evaluation. It shows that although PRD successfully generates PRD decompiled code for 88.5% of all targets (8,742/9,876), 30% (2,969/9,876) fail to recompile. We analyzed these recompilation failures and identified 56 unique fail signatures. Our first recompilation result is that PRD successfully generates a test-case equivalent binary for 48.5% (4,790/9,876) of all targets. However, when decompilation obviously succeeds, PRD successfully generates a test-case equivalent binary for at least 84% of targets (4,790/5,697), which is a lower bound as decompilers may introduce issues discernable only at run-time.

For the second evaluation (Table 3), we focused on the set of critical functions, those associated with a vulnerability (i.e., the function changed in the official repair provided by DARPA). This second result shows that PRD generated a test-case equivalent binary for 57% (109/190) of all targets. Our third recompilation result is that only 3% (5/190) appeared to decompile but failed dynamically on test cases. One was missing a `__attribute__` (adding it produced a test-case equivalent binary), two were decompiler correctness errors. Only two were C-specific run-time issues (a non-decompiler weakness in our approach): dereferencing pointers (detour reference) to a multi-dimensional static array (external global symbol).

All other failures, 42% (79/190), are decompiler-related and pertain to input issues, decompiler failures, or decompiler output issues. The following static issues account for 76 of the 79 decompiler-related failures. For 12% (23/190), *decompilation* failed: one because the vulnerability was located in a global array, not a function; the remaining 22 failed because decompilation (Hex-Rays) failed. For 16% (31/190), *partial recompilation* failed because gcc failed to compile the PRD-produced decompilation output. For 12% (22/190), *binary rewriting* failed from misspecified (unbound) or undefined symbols for detouring with references, due to the misidentification of symbol reference names/types.

Result	Static / dynamic	Description	Count (C++)	Unique
success		test-case equivalent	4790 (23)	2181
failure	dynamic	inconsistent test-case results	907 (50)	520
failure	static	unbound or undefined symbols	76 (0)	76
failure	static	recompilation errors	2969 (809)	347
failure	static	decompilation failed	1134 (451)	670
TOTAL			9876 (1343)	

Table 2: PRD partial decompilation and recompilation for each function from both "CGC" benchmarks. *Count* indicates the raw number of functions; *unique* indicates the number of uniquely named functions, a code reuse indication.

Result	Failure type	Static / dynamic	Description	Count
success			test-case equivalent	109
failure	decompiler	static	recompilation errors	31
failure	decompiler	static	decompilation failed	22
failure	decompiler	static	unbound or undefined symbols	22
failure	decompiler	static	no actual function in vulnerable-set	1
failure	decompiler	dynamic	functional errors in decompilation	2
failure	decompiler	dynamic	decompiler missed __attribute__	1
failure	c-lang	dynamic	reference is multi-dimen array	2
TOTAL				190

Table 3: PRD partial decompilation and recompilation success on one developer-specified vulnerable function for each C-based DARPA CGC CB. *static* indicates errors found during PRD binary generation; *dynamic* indicates run-time errors.

	"AE"		GenProg "single edit"		GenProg "GA"		Prophet	
	Full	PRD	Full	PRD	Full	PRD	Full	PRD
Total	137	157	129	157	94	157	79	157
Completed	122	137	113	129	67	94	79	79
Repairs	45	**69**	48	**69**	32	**51**	**57**	52

Table 4: Full-source (baseline) vs. PRD-enabled comparison using APR. *Full* refers to full-source as APR input and *PRD* is end-to-end with APR and PRD decompiled source. We report the number of scenarios that produced a plausible mitigation, as well as the *total* scenarios that *completed* within 8 hours.

These results show that when decompilation succeeds, with very few exceptions (2/190), PRD produces patched binaries that are test-case equivalent to the original. PRD provides a solid foundation for source-level transformations.

6.3 RQ3: Mitigating vulnerabilities with BinREPARED

"To what degree can off-the-shelf source-level APR tools, in an end-to-end setting, succeed at mitigating vulnerabilities in binaries?" Because APR tools do not always succeed, we compared the success rate for APR tools applied to the actual source code of the binary to the success rate for the same APR tool applied to the PRD decompiled source code.

We evaluate on the 157 defect scenarios from the C-based CGC benchmark suite. As a baseline, when given access to the full original source (not normally available in our use case) and limited to an 8-hour run-time, GenProg "GA" produces 32 candidate repairs, "AE" and "single-edit" do slightly better (45 and 48, respectively), and Prophet performs much better with 57.

6.3.1 End-to-end BinREPARED Result. Our *primary BinREPARED result* (Table 4), shows that PRD-enabled repair, operating only on binaries, *performs as well as and sometimes better than* full-source repair within the 8-hour search budget: PRD-supported algorithms find 51–69 plausible patches, while the full-source baselines find 32–57. Prophet performs slightly better with access to the original source (57 vs. 52) while the GenProg variants perform better in the PRD setting (51–69 vs. 32–48). Collectively, our PRD-enabled APR tools mitigated 85 of the 148 completed, unique scenarios (including 20 CBs that the winning CRS, Mayhem, did not patch). Overall, we find that the success rate for off-the-shelf tools operating on binaries via BinREPARED is consistent with and sometimes exceeds that for the same tools operating on full-source ($p<0.0004$, proportions z-test).

This evaluation features an end-to-end use of BinREPARED. CGFL implicates a subset of the binary, which is the partially decompiled, providing adequate information for the APR algorithm (which then uses its own fine-grained fault localization and other static and dynamic analyses). The source-level patches are partially recompiled and composed with original binary content (e.g., using detouring with references).

6.3.2 Repair Quality. APR tools sometimes find repairs that overfit the test suite [33, 73] without addressing the root cause of the problem. We do not improve or worsen that orthogonal concern here, instead finding that BinREPARED inherits the repair quality of its underlying APR method. In our use case, disrupting an exploit quickly is valuable, even if the repair is not completely general. For example, this is particularly true in the context of the CGC, where participants competed with one another under time constraints. In the following, we focus on GenProg results for simplicity; Prophet results are similar.

First, in cases where GenProg failed to produce a mitigation, the required edit was usually out of scope for the algorithm. For example, some official repairs change `struct` fields or variables which were defined but never referenced in the source. Other repair failures involved special constant values or comparators, a known weakness of this tool (cf. [56]).

Second, we consider the mitigations that GenProg did find. In most of cases, multiple solutions were found, so we randomly sampled 5–10% of the unique mitigations and examined their C representations (recall from Section 3 that while our BinREPARED repairs binaries, it retains source-level patches, facilitating such analyses). We do not find statistically significant difference in the rate of overfitting between the full-source APR baseline and PRD-enabled APR applied to binaries. As a case study, we describe one example, which mitigates the vulnerability but does not address it generally (overfits), and then describe a second example that fully corrects the defect in a more general way.

Lower-Quality Mitigation. For the `<KPRCA_00013>` CB's first vulnerability, GenProg mutation `a(1178,1056)` passes all tests and successfully mitigates it. In this edit, the "`(`" character is pushed onto the operator stack in a loop. In the next iteration, an error is flagged because the top of the stack is "`(`". Although this patch does not address the official *Off-by-one error* or *Use of uninitialized variable* vulnerabilities for this CB, it does mitigate this particular exploit, preventing control of the next heap block's heap metadata.

Higher-Quality Repair. The `<NRFIN_00076>` CB's first vulnerability simulates a vulnerability that is introduced when a programmer commits unfinished code. The program incorrectly increments `*results` in a frequent function (*Incorrect Pointer Scaling*), leading to the use of an invalid pointer (*Untrusted Pointer Dereference*). One mitigation that GenProg found is mutation `d(4)`, which correctly deletes the problematic code and eliminates the pointer vulnerabilities.

Our **primary BinREPARED result** shows that in an end-to-end scenario, using PRD to apply a source-level APR tool to binaries produces results that *are consistent with and sometimes better than* using those same techniques on the corresponding source. This is true both in terms of the rate at which vulnerabilities are mitigated and in terms of repair quality (overfitting).

6.4 RQ4: BinREPARED Components

"How effective are our coarse-grained fault localization and decompilation algorithms?" Having established that BinREPARED works as well in end-to-end scenarios, we now consider the relative effects of particular algorithmic steps.

6.4.1 CGFL. We evaluated the hypothesis that our CGFL algorithms identify an appropriate vulnerable subset. While the end-to-end evaluation shows that it works effectively for in practice, we also evaluate it independently, first confirming that all CGC CBs, both C and C++, are compatible with our CGFL implementation.

We used the developer-provided patched functions as ground truth (i.e., vulnerable functions that should be implicated) for the 110 CBs. CGFL successfully identified (within the first three ranks) at least one ground-truth function for 95 of the 110 binaries and found all ground-truth for 74.

When CGFL failed to identify a vulnerable function, we observed these failure types: (a) 4 binaries did not exercise any vulnerable function in any negative test, (b) 10 were in the first three ranks, but ties with other functions impacted their selection, and (c) 1 binary involved a reimplemented `malloc`.

While CGFL succeeds more than 85% with our criteria, CGFL failures can be readily explained or mitigated. For failure type (a), the specific problems (function not exercised in test content or repair outside of a function body) cannot be addressed by SBFL or by APR. Failure type (b) is common in SBFL metrics, with large ties with same rank values. If better test content is not available, then this could be readily mitigated by increasing the size of $\mathcal{K}$, per Section 4.1). Finally, failure type (c) is a result of our simple heuristic to screen out system functions. These results suggest that even with minimal spectra, CGFL performs well when coupled with good negative test content (e.g., repeatable exploits).

6.4.2 Decompilation. To assess the impact of our informative decompilation approach (using off-the-shelf tools and heuristics), we studied a random sample of 21 CGC CBs. Given the associated negative tests, these 21 binaries correspond to 30 vulnerability scenarios, and cover 24 distinct Common Weakness Enumerations (CWE), a wide range of security classes.

Table 5 shows the details for those 30 scenarios, together with summary information about code and binary characteristics of the original and PRD-generated binaries. We consider access to the full source of the program (our baseline), the PRD-provided decompiled source for implicated methods, and a bound in which the CGC-provided source is used for the same implicated methods ("ideal" decompilation).

We observe that with full access to the source (our baseline), GenProg and Prophet separately produce 28 candidate patches, with our PRD-generated binaries they produce 18, and with ideal decompilation they produce 34. This echoes the RQ1 finding that state-of-the-art decompilation tools have room to improve in end-to-end usage scenarios.

In addition, the "CB LOC" and "PRD LOC" offer a partial explanation for how decompilation can perform comparably to full source: the decompilation applies only to the implicated functions (reducing effective LOC by 95%), while the full source provides the entire source code to the APR tool. This supports our insight that coarse-grained fault localization, if accurate at implicating a relevant subset, can help downstream stages (such as decompilation or program analysis/transformation).

We find that our coarse-grained fault localization algorithm works well in practice (identifying a relevant function in 95/110 of cases). We observe that decompilation can have a strong impact on quality, but weaknesses of current decompilation tools are mitigated by fault localization filtering.

6.5 RQ5: Generality

"Does our technique generalize to real-world defects, other languages, or performance constraints?" Our previous results establish generality across defect types and APR algorithms, but for a binary-facing technique it is also important to consider multiple source languages, execution overhead, and real-world (i.e., larger, non-benchmark) programs and defects.

6.5.1 Application to C++. While we cannot directly evaluate our APR algorithms on C++ (an unsupported language), we can show that PRD can partially decompile and recompile binaries produced from C++. In this assessment we use the DARPA CGC CB benchmark set of 10 C++ binaries. This involves the limited use of a secondary decompiler, Ghidra [8], aiding C++-decompilation to overcome known Hex-Rays issues. We applied PRD to the 149 functions identified by CGFL and successfully generated 36 test-case equivalent PRD-binaries including C++ class methods. These successes cover 7 of the 10 C++-sourced binaries. Recompilation failures (94) dominated our C++ failures, with test (10) and decompilation (9) failures lagging, similar to our results from Section 6.2.

These results indicate that the PRD framework is not only extensible to C++ binaries, but also that there is the potential to use PRD to repair C++ binaries with C-based APR tools.

6.5.2 Performance Analysis. We consider two performance aspects for PRD binaries: run-time and compile-time.

First, using `perf stat`, we compared the run-time performance of 100 PRD binaries to their counterparts over 5,163 tests. The observed performance was not statistically significant (user: $p < 0.970$; system: $p < 0.277$, two-tail t-test).

Second, to study a different notion of overhead relevant to APR algorithms, we sampled 25 CGC CBs and respective single-detour PRD binaries, generating each 25 times. Generating a PRD-binary is statistically less expensive for the tested APR tools than compiling the binary from source (user:$p < 0.0$; system: $p < 8.4845e^{-192}$, two-tail t-test).

These results indicate that the PRD framework does not induce a performance overhead on APR tools (either in test case evaluation on running binaries or in the production of binaries for candidate patches).

6.5.3 Real-world Vulnerabilities. In addition to the DARPA CGC Dataset, we used two real-world programs that contain known vulnerabilities (CVEs) to evaluate PRD's applicability and the effectiveness of CGFL on real vulnerabilities (see Table 1).

Consider `CVE-2021-30472` [2] for podofopdfinfo [11]. We used the developer's recommended example PDFs and the CVE vulnerability input, *bug4*, as tests. The exploit hits a vulnerability in `PoDoFo::PdfEncryptMD5Base::ComputeEncryptionKey` ("f1") before exercising another vulnerability in a second function, `PoDoFo::PdfEncryptMD5Base::ComputeOwnerKey` ("f2").

Evaluating CGFL on podofopdfinfo was simple: our CGFL approach (based on *callgrind*) was applied directly to our test baseline.

DARPA CGC CB name	Fn id	POV	POLL tests	CB LOC	PRD LOC	PRD funcs	# AST nodes	Prophet (full)	GenProg (full)	Prophet (PRD)	GenProg (PRD)	Prophet (ideal)	GenProg (ideal)	Mayhem
KPRCA_00013	0	pov_1	44	2,360	1,917	7	349	✓	✓	✓	✓	✓	✓	
KPRCA_00013	0	pov_2		-	-			✓	✓	✓	✓	✓	✓	
NRFIN_00041	0	pov_1	84	1,138	510	8	207		✓		⊥			✓
CROMU_00027	4	pov_2	200	7,502	83	3	129	⊥						
CROMU_00027	4	pov_3		-	-			⊥	✓					
CROMU_00027	4	pov_4		-	-			⊥						
CROMU_00027	4	pov_5		-	-			⊥						
KPRCA_00010	1	pov_1	99	1,920	744	14	185	✓	✓	✓			✓	✓
KPRCA_00009	0	pov_1	100	8,397	107	3	11	✓	✓	✓	⊥	✓	✓	
KPRCA_00009	0	pov_2		-	-			✓	✓	✓	⊥	✓	✓	
KPRCA_00009	0	pov_3		-	-			✓	✓	✓	⊥	✓	✓	
CROMU_00001	0	pov_1	101	43,394	126	3	41		⊥	+	+	✓	✓	✓
YAN01_00010	0	pov_1	100	228	123	3	14		✓	+	+		✓	✓
NRFIN_00075	0	pov_1	87	882	178	1	892	✓	⊥	+	+	✓**	✓**	
CROMU_00033	0	pov_1	100	988	105	1	53			✓	✓	✓	✓	
CROMU_00032	0	pov_1	101	1,151	313	3	134			+	+	✓		✓
NRFIN_00035	0	pov_1	100	6755	140	1	757		✓		⊥			
KPRCA_00017	0	pov_1	29	1,225	162	3	50				⋈			
KPRCA_00017	0	pov_2		-	-			✓		✓	⋈	✓	✓	
KPRCA_00019	0	pov_1	100	1,399	216	3	49	✓	⊥	+	+	✓	✓	
KPRCA_00019	0	pov_2		-	-			✓	⊥	+	+	✓	✓	
CROMU_00037	0	pov_1	101	52,123	124	3	26	✓	⊥	✓	✓	✓	✓	✓
CADET_00001	0	pov_1	99	208	97	4	33		✓					-
KPRCA_00073	0	pov_1	93	1,918	129	3	17			✓	⊥	✓		
KPRCA_00060	0	pov_1	51	1,051	244	3	74	✓		✓	⊥	✓	✓	-
NRFIN_00076	0	pov_1	100	1,811	52	3	8		⊥	+	+	✓	✓	-
KPRCA_00007	0	pov_1	32	1,631	111	3	16	✓	✓	✓		✓		
NRFIN_00020	0	pov_1	92	720	139	6	57				⊥			
NRFIN_00020	0	pov_2		-	-				✓		⊥			
CROMU_00010	0	pov_1	100	15,270	101	1	2	✓	✓	✓	✓	✓		✓
Repairs (Total)		(30)						14 (26)	14 (24)	13 (23)	5 (11)	18 (30)	16 (30)	

Table 5: Experimental evaluation of BinREPARED on 30 scenarios (a scenario is `<CB,Func,POV>`). Each row corresponds to one scenario; *CB LOC* and *PRD LOC* refers to the lines-of-code in the DARPA source code and decompiled source, respectively; *# AST nodes*, the size of the decompiled CIL-AST;*(full)* indicates full-source APR results;*(prd)*, results using PRD decompiled source; *(ideal)*, results using Ideal decompilation. We use + for decompiler errors; ⊥ for APR tool issues; ⋈ for APR profiling failures tht clearly expose vulnerability; ✓ for successful repairs; ✓ where PRD application mitigated the exploit; practice binaries are indicated by - .**

All five SBFL metrics identified "f1" in rank 1 (10 ties) of 265 with "f2" at rank 147—echoing the bug precedence.

We successfully applied PRD to podofopdfinfo on both vulnerable methods: "f1" and "f2". This required a minor change associated with *glibc*. Because the Hex-Rays decompiler comments out GCC libc function prototypes, we added the necessary content as well as references to our detour entries. Hex-Rays also decompiled the "stack canary" incorrectly—we manually fixed this stack protection with equivalent inline assembly. After that, using the identity transformation, PRD successfully generated a test-case equivalent binary for "f2", while "f1" improved on test-case equivalency addressing the specific CVE vulnerability, detouring correctly from the G++-compiled original binary to our GCC-8 recompiled content.

Now, consider `CVE-2021-3496` [3] for jhead [10]. We applied the same methods to jhead: CGFL to test script (Travis-CI) and PRD to the binary. The first 3 functions identified were ErrFatal, ErrNonFatal, and ProcessMakerNote. The ProcessMakerNote function inlines the developer-reported buggy function *ProcessCanonMakerNoteDir*: our CGFL implicated the correct code as rank 3 (no ties) of 42.

Applying PRD to jhead required another minor *glibc*-associated change. Because the `ld` linker may resolve global external symbols at runtime, we cannot guarantee that all global symbols are statically initialized before PRD-generated code uses them, and thus translate `putc` into `putchar`. With this simple transformation, PRD successfully generated a test-case equivalent binary, detouring correctly from the Clang-compiled original binary to our GCC-8 recompiled content. Using transformations that apply a bug fix, BinREPARED produces a binary that passes all provided test cases as well as defeats the CVE-reported bugs for both jhead and podofopdfinfo.

Despite the limitation of RAFL and the required manual intervention (readily automated), in these case studies our CGFL and PRD applied successfully to two real-world issues.

We demonstrate the applicability of PRD to C++: our rate of producing test-equivalent binaries for our C++ benchmarks, 24.2%, is comparable to our rate for C. We find no run-time overhead for producing or running PRD-produced binaries. Finally, on the two real-world programs with real-world CVEs, we find that both CGFL and PRD are successful on 2 out of 2 real-world issues, increasing confidence in our systematic CGC evaluation.

7 DISCUSSION

With an initial coarse-grained analysis identifying a relevant function set, and the decompilation of a small function set, PRD enables manual and automated mitigation of binary-level exploits.

Once validated, the patch is recompiled, and our binary rewriting produces a unified binary that executes the patched functions in place of the originals. The current limitations of decompilers are addressed by identifying a small function set, which increases decompilation accuracy and subsequent recompilability. On average, this approach reduced the lines-of-code (LOC) by an average of 95% from CB source to decompiled source (see Section 6.4.2). Although decompiled source is not as readable as the original (e.g., variable names and high-level control-flow structures are missing), it is much less verbose and more readable than assembly code.

Readability is not an impediment for APR, as evidenced by the similar success rates we observed for full-source vs. BinREPARED repairs. Additionally, APR algorithms limited to C-source can apply to C++ generated binary content through its decompilation into C-like source. These results are encouraging in terms of using source-level repair methods in this context.

7.1 Limitations and Caveats

PRD applicability and compatibility. BinREPARED and PRD are not applicable for tools that use whole-program analysis (such as the use of symbolic execution by Angelix [46]) or interpreted languages. While our implementation focuses on 32b ELF and System-V Application Binary Interface (ABI), PRD is compatible with other ABIs and binary formats if their calling conventions are upheld. We do not handle self-modifying or self-checking binaries. Relevant to security, our prototype *is* compatible with ASLR binaries.

Decompilation failures. Although binary decompilation techniques have achieved impressive results, modern decompilers still struggle to generate satisfactory results when the binary (1) is compiled from a non-C language, (2) contains manual assembly code, or (3) contains self-modifying or obfuscated code. Improvements in binary decompilation techniques also enhance PRD's generality and BinREPARED's quality.

Unsound decompilation results. Decompilers do not always generate decompiled code that preserves the semantics of the original binary. While determining the equivalence of two arbitrary binaries is undecidable in general, this problem can be addressed by requiring byte-level equivalence between the original binary code and the recompiled (but unpatched) code [66]. We validate the partially-recompiled binary against existing test cases, assuming adequate coverage.

Prototype limitations. The most serious challenge to our prototype arises from limitations of current decompilers. When we fail to generate test-case equivalent PRD binaries, the vast majority of failures (79 of 81) were caused by decompilation issues. The two remaining are technical limitations with language-specific references. The example in Section 3 and the end-to-end repair results demonstrate the full pipeline for the Hex-Rays decompiler. As our limited use of Ghidra indicates, PRD can be configured to accommodate other decompilers as needed.

8 RELATED WORK

Our approach and evaluation rely on recent progress in three major research areas: APR, binary code decompilation, and binary patching and rewriting.

Binary Patching and Rewriting. There are dynamic and static binary rewriting techniques. Dynamic binary rewriting, also known as dynamic binary instrumentation, inserts user code at specified locations in the target binary at runtime, e.g., Pin [41], Valgrind [52], and DynamoRIO [7]. Because these techniques must translate and relocate every block during runtime, they introduce prohibitively high overhead and are not used in production for binary patching.

Static binary rewriting techniques perform code transformation and relocation before executing the target binary. They have much lower runtime overhead compared to their dynamic counterparts and suit generic binary patching tasks, such as binary patching and control-flow integrity enforcement.

Ramblr [76] and ddisasm [22] convert binary code into assembly code that can later be reassembled into a new binary. E9Patch performs in-binary byte editing in AMD64 binaries to allow insertion of a few chunks of code [21]. Egalito [78] enables a similar binary transformation mechanism as LIEF [74]. CGC finalists used either in-place binary editing or reassembly to apply their patches [16, 53, 70]. But none of the existing solutions transform binary code to high-level representations. In comparison, BinREPARED uses partial recompilation to enable APR or manual repair on decompiled C-code.

Automated Program Repair. Automated Program Repair (APR) is a mature research area. Three recent surveys [23, 24, 48] review more than a decade of this work. Most APR techniques work on intermediate representations instead of on binary content. Some exceptions are Schulte *et al.*'s [64, 67] executable work and Orlov and Sipper's early work [57, 58] on Java bytecode. Since Java bytecode is interpreted by JVM and not aligned with PRD, we omit their discussion. Similarly, Angelix [47], which uses whole-program symbolic execution, is not compatible with PRD.

As mentioned earlier, other APR tools could be used in place of the two we tested. Closely related tools, such as RSRepair [59], Kali [60], and SPR [39] would require only similar modifications to those we made for the tools we tested. Other tools that are compatible to PRD but would require more extensive modification include CodePhage [72] and CodeCarbonCopy [71], and recent methods that use machine learning [42, 43].

OSSPatcher [20] targets third-party, open-source libraries for automatic binary patching. However, it requires source code and source-based patches, while we use decompilation to generate high-level source code. Our implementation extends naturally to libraries by standard hooking techniques.

Binary Code Decompilation. The quality of binary code decompilation relies on advances in (a) binary code extraction, (b) (control flow) structural analysis, and (c) type inference. Binary code extraction on non-obfuscated binaries is equivalent to control flow graph recovery, where state-of-the-art approaches work in a compiler-, platform-, and architecture-agnostic manner with high precision [15, 22, 61]. Structural analysis has progressed significantly: Schwartz *et al.* reduced the number of `goto` statements by applying iterative refinement and semantics-preserving analysis [68]; Yakdan *et al.* proposed pattern-independent control-flow structuring to eliminate `goto` statements and improve readability [81]. Decompilers often use static analyses or type inference due to their intrinsic requirement in code coverage (e.g., [36, 44, 55, 80]).

Rapid progress in decompilation has enabled the *recompilation* of decompiled code—deemed impossible by most researchers until very recently. For example, Liu *et al.* showed that the output of modern decompilers is generally recompilable after applying automated syntax-level transformations [38]. They also confirmed that decompilers make mistakes during decompilation and may generate incorrect output.

9 CONCLUSION

Security-critical vulnerabilities that arise after software is deployed must be addressed quickly, even when recompilation is not possible. Further, 15–25% of sampled post-release operating system bug fixes are reported to have end-user visible impacts such as information corruption [82]. We present a new way to patch binaries when recompiling from source is not an option. While it cannot yet replace full-source, we show that decompilation generates recompilable code for most functions. By focusing on only the vulnerable functions, state-of-the-art decompilation can produce recompilable code that is amenable to source-level code repair tools. BinREPARED uses CGFL to identify a buggy function set, partial decompilation to lift part of the binary to source, where repairs are developed and applied, then generates a unified binary addressing the problem. Our implementation and datasets are available at *git@github.com:pdreiter/FuncRepair.git*.

Today's tools are better at finding vulnerabilities than they are at patching them. We hope that BinREPARED will improve that capacity by leveraging recent advances in source-level APR. Although APR is an active area of research and used in industry, that potential has not been equally realized for binary code. BinREPARED using PRD helps to address these shortfalls.

REFERENCES

[1] [n.d.]. CVE-CVE. https://cve.mitre.org/index.html. Accessed: 2021-07-28.
[2] [n.d.]. CVE-CVE-2021-30472. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-30472. Accessed: 2021-07-28.
[3] [n.d.]. CVE-CVE-2021-3496. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3496. Accessed: 2021-07-28.
[4] [n.d.]. Cyber Grand Challenge. https://www.darpa.mil/program/cyber-grand-challenge. Accessed: 2021-08-01.
[5] [n.d.]. *CyberGrandChallenge/samples*. https://github.com/CyberGrandChallenge/samples Accessed: 2020-07-30.
[6] [n.d.]. diet libc - a libc optimized for small size. https://www.fefe.de/dietlibc/. Accessed: 2020-07-29.
[7] [n.d.]. DynamoRIO. https://dynamorio.org. Accessed: 2021-08-07.
[8] [n.d.]. Ghidra. https://ghidra-sre.org/. Accessed: 2021-07-28.
[9] [n.d.]. Hex Rays. https://www.hex-rays.com/products/decompiler/. Accessed: 2021-01-27.
[10] [n.d.]. Matthias-Wandel/jhead. https://github.com/Matthias-Wandel/jhead. Accessed: 2021-07-28.
[11] [n.d.]. PoDoFo download | SourceForge.net. https://sourceforge.net/projects/podofo/. Accessed: 2021-07-28.
[12] [n.d.]. Using the GNU Compiler Collection (GCC): Top. https://gcc.gnu.org/onlinedocs/gcc-8.4.0/gcc/index.html#SEC_Contents. Accessed: 2020-07-29.
[13] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792.
[14] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2009. Spectrum-based multiple fault localization. In *International Conference on Automated Software Engineering*. IEEE, 88–99.
[15] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic Function Detection in Binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 177–189.
[16] Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, Alex Rebert, and Ned Williamson. 2018. The Mayhem Cyber Reasoning System. *IEEE Security Privacy* 16, 2 (Mar 2018), 52–60.
[17] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to recognize functions in binary code. (2014), 845–860.
[18] Marcus Botacin, Lucas Galante, Paulo de Geus, and Andr Grgio. 2019. RevEngE is a dish served cold: Debug-Oriented Malware Decompilation and Reassembly. In *Reversing and Offensive-oriented Trends Symposium*. ACM, 112.
[19] Roberto Paulo Andrioli de Araujo and Marcos Lordello Chaim. 2014. Data-flow testing in the large. In *International Conference on Software Testing, Verification and Validation*. IEEE, 81–90.
[20] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries. In *Network and Distributed System Security Symposium*. Internet Society.
[21] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Programming Language Design and Implementation*. ACM, 151–163.
[22] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog Disassembly. In *USENIX Security Symposium*. arXiv:1906.03969 http://arxiv.org/abs/1906.03969
[23] L. Gazzola, D. Micucci, and L. Mariani. 2017. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* (2017), 1–1.
[24] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (Nov 2019), 56–65.
[25] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. 2020. A Comb for Decompiled C Code. In *ACM Asia Conference on Computer and Communications Security*. ACM, 637–651.
[26] Thomas Hirsch. 2020. A Fault Localization and Debugging Support Framework driven by Bug Tracking Data. In *International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 139–142.
[27] Yikun Hu, Yuanyuan Zhang, and Dawu Gu. 2019. Automatically patching vulnerabilities of binary programs via code transfer from correct versions. *IEEE Access* 7 (2019), 28170–28184.
[28] Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2005. System V application binary interface. (2005).
[29] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *International conference on Software engineering*. IEEE, 191–200.
[30] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *International Conference on Software Engineering*. IEEE, 467–477.
[31] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 628–639.
[32] James R. Larus and Eric Schnarr. 1995. EEL: machine-independent executable editing. In *Programming language design and implementation (PLDI '95)*. Association for Computing Machinery, 291–300.
[33] Xuan Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* (Mar 2018).
[34] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*. IEEE, 3–13.
[35] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.
[36] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Network and Distributed System Security*. http://dblp.uni-trier.de/db/conf/ndss/ndss2011.html
[37] Shili Lin. 2010. Rank aggregation methods. *Wiley Interdisciplinary Reviews: Computational Statistics* 2, 5 (2010), 555–570.
[38] Zhibo Liu and Shuai Wang. 2020. How far we have come: testing decompilation correctness of C decompilers. In *International Symposium on Software Testing and Analysis*. ACM, 475–487.
[39] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. ACM Press, 166–178.
[40] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 298–312.
[41] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming language Design and Implementation (PLDI '05)*, Vol. 40. 190.
[42] Thibaud Lutellier, Lawrence Pang, Viet Hung Pham, Moshi Wei, and Lin Tan. 2019. ENCORE: Ensemble learning using convolution neural machine translation for automatic program repair. *arXiv preprint arXiv:1906.08691* (2019).
[43] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. (2020), 101–114.
[44] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. 2019. TypeMiner: Recovering Types in Binary Programs Using Machine Learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Vol. 11543 LNCS. 288–308.
[45] Wes Masri. 2010. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability* 20, 2 (2010), 121–147.

[46] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering*. 691–701.

[47] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering*. ACM Press, 691–701.

[48] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (Jan 2018), 1–24.

[49] Martin Monperrus. 2018. *The Living Review on Automated Program Repair*. Technical Report hal-01956501. HAL/archives-ouvertes.fr.

[50] Manish Motwani and Yuriy Brun. 2020. Automatically repairing programs using both tests and bug reports. *arXiv preprint arXiv:2011.08340* (2020).

[51] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 1–32.

[52] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6, 89–100.

[53] Anh Nguyen-Tuong, David Melski, Jack W. Davidson, Michele Co, William Hawkins, Jason D. Hiser, Derek Morris, Ducson Nguyen, and Eric Rizzi. 2018. Xandra: An Autonomous Cyber Battle System for the Cyber Grand Challenge. *IEEE Security Privacy* 16, 2 (Mar 2018), 42–51.

[54] Matthew Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. In *Programming Language Design and Implementation*. 27–41. http://arxiv.org/abs/1603.05495

[55] Matthew Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. In *Programming Language Design and Implementation*. ACM Press, Santa Barbara, CA. arXiv:1603.05495 http://arxiv.org/abs/1603.05495

[56] Vinicius Paulo L. Oliveira, Eduardo Faria de Souza, Claire Le Goues, and Celso G. Camilo-Junior. 2018. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering* 23, 5 (Oct 2018), 29803006.

[57] Michael Orlov. 2017. Evolving software building blocks with FINCH. In *Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. Association for Computing Machinery, 1539–1540.

[58] Michael Orlov and Moshe Sipper. 2009. Genetic programming in the wild: evolving unrestricted bytecode. In *Genetic and Evolutionary Computation Conference (GECCO '09)*. Association for Computing Machinery, 1043–1050.

[59] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *International Conference on Software Engineering*. ACM Press, 254–265.

[60] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, 24–36.

[61] Rui Qiao and R Sekar. 2017. Function interface analysis: A principled approach for function recognition in COTS binaries. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 201–212.

[62] Nathan E. Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2010. Extracting compiler provenance from program binaries. In *Workshop on Program analysis for software tools and engineering (PASTE '10)*. Association for Computing Machinery, 21–28.

[63] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. 2013. Automated repair of binary and assembly programs for cooperating embedded devices. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 317–328.

[64] Eric Schulte, Stephanie Forrest, and Westley Weimer. 2010. Automated program repair through the evolution of assembly code. In *International Conference on Automated Software Engineering (ASE '10)*. Association for Computing Machinery, 313–316.

[65] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. [n.d.]. Evolving Byte-Equivalent Decompilation from Big Code. ([n. d.]), 12.

[66] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. 2018. Evolving Exact Decompilation. In *Workshop on Binary Analysis Research*. Internet Society.

[67] Eric M. Schulte, Westley Weimer, and Stephanie Forrest. 2015. Repairing COTS Router Firmware without Access to Source Code or Test Suites: A Case Study in Evolutionary Software Repair. In *Genetic and Evolutionary Computation Conference (GECCO Companion '15)*. Association for Computing Machinery, 847–854.

[68] Edward Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. 2013. Native x86 Decompilation using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *USENIX Security Symposium*. USENIX, 353–368. https://www.usenix.org/node/180374

[69] Francisco Servant and James A Jones. 2012. WhoseFault: automatic developer-to-fault assignment through fault localization. In *International conference on software engineering*. IEEE, 36–46.

[70] Yan Shoshitaishvili, Antonio Bianchi, Kevin Borgolte, Amat Cama, Jacopo Corbetta, Francesco Disperati, Audrey Dutcher, John Grosen, Paul Grosen, Aravind Machiry, Chris Salls, Nick Stephens, Ruoyu Wang, and Giovanni Vigna. 2018. Mechanical Phish: Resilient Autonomous Hacking. *IEEE Security and Privacy* 16, 2 (2018), 12–22.

[71] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. 2017. CodeCarbonCopy. In *Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, 95–105.

[72] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, 43–54.

[73] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, 532–543.

[74] Romain Thomas. 2017. LIEF: Library to Instrument Executable Formats. https://lief.quarkslab.com/. Accessed: 2021-08-07.

[75] trailofbits. [n.d.]. trailofbits/cb-multios: DARPA Challenges Sets for Linux, Windows, and macOS. https://github.com/trailofbits/cb-multios. Accessed: 2021-08-07.

[76] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *Network and Distributed System Security Symposium*.

[77] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 356–366.

[78] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-agnostic binary recompilation. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 133–147.

[79] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308.

[80] Zhiwu Xu, Cheng Wen, and Shengchao Qin. 2017. Learning types for binaries. In *International Conference on Formal Engineering Methods*. Springer, 430–446.

[81] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. February (2015), 8–11.

[82] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 2636.

[83] Hui Zhu, Tu Peng, Ling Xiong, and Daiyuan Peng. 2017. Fault Localization Using Function Call Sequences. *Procedia Computer Science* 107 (2017), 871–877.