Automatically Mitigating Vulnerabilities in Binary Programs via Partially Recompilable Decompilation

Pemma Reiter*, Hui Jun Tay*, Westley Weimer†, Adam Doupé*, Ruoyu Wang*, Stephanie Forrest*
* Arizona State University, † University of Michigan

Abstract— Vulnerabilities are challenging to locate and repair, especially when source code is unavailable and binary patching is required. Manual methods are time-consuming, require significant expertise, and do not scale to the rate at which new vulnerabilities are discovered. Automated methods are an attractive alternative, and we propose Partially Recompilable Decompilation (PRD) to help automate the process. PRD lifts suspect binary functions to source, available for analysis, revision, or review, and creates a patched binary using source- and binary-level techniques. Although decompilation and recompilation do not typically succeed on an entire binary, our approach does because it is limited to a few functions, such as those identified by our binary fault localization.

We evaluate the assumptions underlying our approach and find that, without any grammar or compilation restrictions, up to 79% of individual functions are successfully decompiled and recompiled. In comparison, only 1.7% of the full C-binaries succeed. When recompilation succeeds, PRD produces test-equivalent binaries 93.0% of the time.

We evaluate PRD in two contexts: a fully automated process incorporating source-level Automated Program Repair (APR) methods; and human-edited source-level repairs. When evaluated on DARPA Cyber Grand Challenge (CGC) binaries, we find that PRD-enabled APR tools, operating only on binaries, perform as well as, and sometimes better than full-source tools, collectively mitigating 85 of the 148 scenarios, a success rate consistent with the same tools operating with access to the entire source code. PRD achieves similar success rates as the winning CGC entries, sometimes finding higher-quality mitigations than those produced by top CGC teams. For generality, the evaluation includes two independently developed APR tools and C++, Rode0day, and real-world binaries.

1 Introduction

Fixing software bugs is challenging when vendor support is unavailable, source code is not provided, or rebuilding the software from source is infeasible. In these cases, bugs must be mitigated at the binary level, which is a tedious, complicated, and error-prone task that does not easily scale to the number of bugs plaguing today's software. At the binary level, the most compelling bugs are security vulnerabilities, which are important to address quickly after disclosure to reduce attack and exploitation.

One approach to this challenge is Automated Program Repair (APR). Despite active development on many different tools and techniques, most APR methods today operate only on source code [1]. Thus, an appealing alternative would lift the entire binary to source, to be analyzed, modified, and then recompiled to generate a patched binary. Unfortunately, current decompilation tools have scalability issues [2] or focus on readability rather than recompilability [3]–[5], often producing inaccurate or uncompilable results when applied to a whole binary [2].

Instead, this paper presents a hybrid approach to APR centered on the idea that for most (if not all) binary programs, partial analysis is sufficient for binary repair. Several insights guided our approach: fault localization need only identify a small set of functions relevant to the vulnerability; decompilers can lift a small set of functions and compatible types to recompilable source code; binary-source interfaces and binary rewriting can transform them into test-equivalent binaries, even when tools fail for full binaries; the set of decompiled binary functions provides sufficient context to enable source-level analyses and transformations, even when those methods operate only on source.

Practically, we present an effective mechanism that consolidates partial analysis content to achieve automated binary patching. To our knowledge, there are no such existing tools, partly because their underlying techniques are laborious, i.e., generating compatible binary content from source, altering compiled binaries with that content, and ensuring that the result remains executable are all challenging. Our fully functional prototype mitigates these difficulties by automating the burden of manually patching source into binaries. Specifically, it analyzes multiple abstractions and generates binary-source interfaces which enable the use of high-level source code to patch binaries. In this context, only the patched function offset and its referenced types and symbols (functions and variables) need recovery, significantly reducing the requirement of complete and sound type inference on binary code, an open research problem. Our source-based, binary-source patching method is called Partially Recompilable Decompilation (PRD), and its *source-based APR*-compatible extension to automated binary repair, BinrePaiReD.

Unlike reassemblers or binary recompilers, which operate on low-level software abstractions, PRD enables source-level analyses by leveraging the output of decompilers. Such 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 all functions local to a binary, we say a decompilation is *partial* when only a subset of binary functions are decompiled and compatible types recovered. This design choice mitigates some of the weaknesses and limitations of current decompilers and enables source-only analyses on binary code before decompilation achieves perfection.

PRD tackles the analytical and engineering challenges posed by source-level binary patching, with the objective of producing binaries with the same executional qualities as the original. This includes the ability to call and use symbols from the original binary from freshly compiled functions, accomplished by PRD via binary-source interfaces, and vice versa. Instead of recompiling (unavailable) source, PRD compiles decompiled functions (that may have been modified at the source level) and binary-source interfaces to produce binary patch content (*PRD decompiled code*). Since compilers do not support combining new content with non-object binary content, PRD links and locates new binary patch content and overwrites the original buggy function with a custom detour to a binary-source interface.

Our evaluation focuses on PRD applicability to APR, generality to real-world vulnerabilities, other languages (C and C++ x86), performance constraints, and the assumptions. Previous evaluations of decompiler output constrained source code by restricting grammar and types [2]. We evaluate decompiled code's recompilability and behavioral consistency without these restrictions. Because APR depends on fault localization, we study its effectiveness at identifying functions relevant to vulnerabilities. Our evaluation includes two independently developed and evaluated APR tools (Prophet [6] and GenProg [7]), and uses off-the-shelf tools (e.g., Hex-Rays and GCC). Our datasets include the DARPA Cyber Grand Challenge (CGC) [8], Rode0Day [9], and C and C++ programs from the MITRE CVE. Surprisingly, these APR tools performed well on the datasets despite being restricted to just a small set of functions, both for target repair locations and as sources of code for the repairs. Our implementation is extensible to other architectures and stripped binaries, assuming decompiler support.

To summarize, the main contributions of this paper are:

- PRD: A system to support repair of binaries using high-level source code when the complete source is unavailable for recompilation.
- The fully functional PRD prototype, the first to support source-level patching of a binary, including a description of the technical implementation
- An empirical validation of the individual assumptions that guided PRD's design: (1) that partial decompilation usually succeeds when full decompilation is infeasible; (2) that recompilation of decompiled functions usually produces test-equivalent binaries; and (3) that coarse-grained fault localization can localize faults to a small set of relevant functions. In particular, we find that even with no special grammar or compilation restrictions, 79.1% of individual functions can be successfully decompiled and recompiled (when only 1.7% of full binaries succeed). When recompilation succeeds, PRD produces test-equivalent binaries 93.0% of the time. Our coarse-grained fault localization method works well, containing the relevant function in 299/316 cases.
- PRD generalizes to multiple languages (C and C++) and compilers (Clang and GNU) with no significant impact to performance.
- An end-to-end evaluation of BinrePaiReD's ability to apply source-based APR tools to mitigate vulnerabilities in CGC binaries. We find that two APR tools, when used with PRD, mitigate vulnerabilities with success rates comparable to, sometimes exceeding, these same tools operating on the full source. They collectively mitigate 85 of 148 unique defects (20 that the winning cyber-reasoning system failed to patch) and provide code that humans can analyze and amend.

To further reproducible science, our prototypes, datasets, and our experimental results are available at https://github.com/AdaptiveComputationLab/FuncRepair.

2 Background

We briefly describe the techniques PRD uses to analyze and manipulate binary content, localize, and repair faults.

2.1 Binary Decompilation and Rewriting.

A binary program, or *binary*, refers to a structured executable file composed of encoded binary instructions. *Disassembling* is the process of lifting binary instructions to assembly; *decompilation* is the process of lifting lower-level abstractions (e.g., assembly) to high-level representations, (e.g., source). Since information about control flow structures, prototypes, variable names, and types is lost during compilation, decompilers must infer this content [5], [10], [11]. Such inference is unsound, often leading to unreadable output, incorrect results, or outright failures. *Binary rewriting* directly modifies a compiled binary file, retaining its ability to execute [12]. PRD uses a binary rewriting strategy that appends freshly compiled content to an existing binary, overwriting the original binary function with instructions to *detour* execution to the new content (Section 4).

2.2 Fault Localization.

Fault localization (FL) finds likely locations of a vulnerability or bug by analyzing a program, dynamically or statically. In Spectrum-Based Fault Localization (SBFL), program spectra (characteristics) are obtained through analysis and used to identify likely locations of the fault. Spectra can include code coverage, data- or information-flow [13], [14], call sequences [15], or program counter samples [16]. SBFL produces suspiciousness scores and ranking (risk evaluation) [17] but does not consider historical development information [18]. To locate vulnerabilities, SBFL maps lower-level code elements, like statements and variables, to their execution, then applies metrics. Most methods require access to source, but we use only the binary and available test cases, calculating suspiciousness scores using multiple SBFL metrics. Specifically, we adapted a hybrid FL method, Rank Aggregation Fault Localization (RAFL) [19] to consolidate

the SBFL metrics using weighted ranks to identify the top-$\mathcal{K}$ (35%) suspicious functions [20]. We refer to our approach and its output as coarse-grained fault localization (CGFL).

For our purposes, CGFL does not need to be as precise as many FL applications, i.e., it does not need to pinpoint the exact lines of faulty code. To succeed, we only need to identify a set of relevant (likely buggy) functions. Although we use SBFL with function spectra, CGFL could leverage other FL methods, e.g., those tailored to specific vulnerability classes [21], assuming that the underlying mechanisms do not require access to source code and can successfully identify a set of suspicious functions. Similarly, if a vulnerability has already been disclosed and fixed in another version of the code, then the changed methods could replace CGFL.

2.3 Automated Program Repair.

Automated program repair (APR) methods generate patches for defects in software with minimal or no human intervention [22]. There are many popular methods (Gazzolla *et al.* provide a survey [23]), but most have adopted a *search-based* approach, defining transformation operators, e.g., different flavors of mutation, to manipulate existing code, and using test suites to validate repair correctness. Other methods use formal semantics either alone or in combination with mutation-based search. Recently, machine learning (ML) approaches based on neural machine translation [24] or large language models [25] have proliferated. Most current ML models require perfect fault location (source code line or function), and they are not appropriate for binary repair because the models are overwhelmingly trained on source code or natural language. However, PRD could be used in conjunction with ML tools, which would require training or fine-tuning with decompiled source code to improve effectiveness. We evaluate PRD using two independently developed mutation-based source-code repair tools: Prophet [6], GenProg [7], [26], and its deterministic variant "AE" [27].

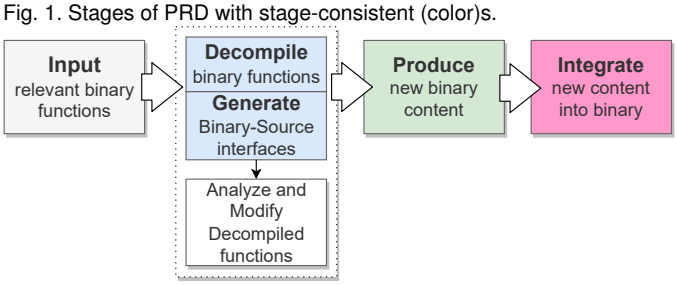
Listing 1. Decompiled code for `KPRCA_00018`'s `cgc_split`, generated by the Hex-Rays decompiler. Low readability of decompiled code does not limit APR tools.

```
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;
  g_srabbit->split_len = v2 + 1;/*BUG*/
  if( v2 > 1 ) return -1;       /*BUG*/
```

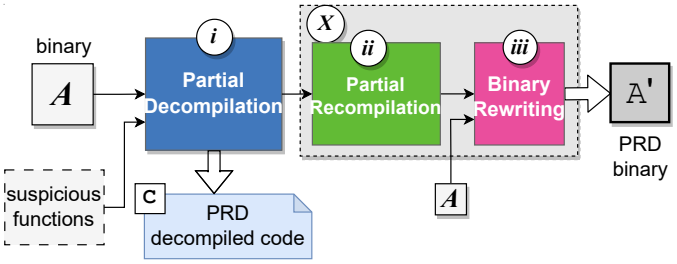
3 Motivating Example

To motivate our approach, consider `KPRCA_00018` (`Square_Rabbit`) from the DARPA CGC dataset, a casino-inspired game with an integer overflow vulnerability that crashes the program. Let's assume we need to prevent crashes, but the software is no longer supported, the source is unavailable, and direct binary fault localization and patching are not feasible [28]. However, we have recovered some tests for the buggy binary (*100* functional, one

Fig. 1. Stages of PRD with stage-consistent (color)s.



(a) Description of input requirements and high-level goals. Note that, at a minimum, vulnerability-relevant binary functions need to be identified for our automated binary patching framework.



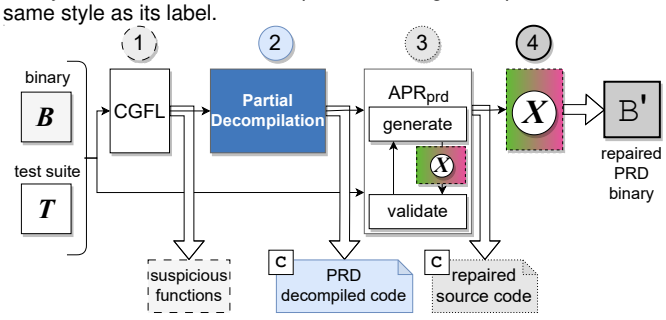
(b) The PRD method: ⓘpartial decompilation (blue), ⓘⓘpartial recompilation (green), ⓘⓘⓘbinary rewriting (rose).

crash-inducing) and have access to a source- and test-based APR tool. Our APR tool does not require the source code to be readable, but it must recompile and be test-equivalent to the original. To accomplish this, we assume Hex-Rays as our decompiler. First, we apply Hex-Rays to the binary which fails to generate recompilable output for all functions (see [29]). Although we cannot decompile the full binary, we *can* use PRD. By recovering compatible type definitions and decompiling functions individually, we can generate a large number of recompilable (87) and test-equivalent (76) functions from the binary.

Next, we localize the likely source of the problem to a small set of 14 suspicious functions from 90 total in the binary (fault localization), including the vulnerable `cgc_split` function, which is tied for the most suspicious.

The next step is generating a patched binary that preserves the expected functionality and repairs the vulnerability. We decompile the suspicious function set, recover

Fig. 2. The four stages of BinrePaiReD with PRD stage-consistent (color)s: ⓘCGFL - identifies a set of relevant binary functions; ⓘⓘpartial decompilation - generates PRD decompiled code, i.e., binary-source interfaces and decompiles binary functions; ⓘⓘⓘsource-level repair (APR_{prd}) - algorithms seeded with decompiled functions search for repairs; ⓘvpartial recompilation and binary rewriting - generates a repaired PRD binary when APR identifies a repair. Each stage's output is formatted in same style as its label.



compatible types (partial decompilation), and transform them, using PRD, into source that is both recompilable and compatible with binary rewriting via binary-source interfaces. To verify that the resulting binary is test-equivalent to the original, we recompile the generated code (partial recompilation), construct a new binary (binary rewriting), then check the result for test equivalence. Finally, we address the vulnerability by repairing the decompiled source, either manually or automatically with APR.

Listing 1 shows the buggy `cgc_split` function as decompiled by Hex-Rays (bug appears on lines 10–11: incrementing `g_srabbit->split_len` results in an integer overflow). Our APR tools successfully identify a developer-equivalent patch (i.e., moving `10` after `11`). Altogether, we obtain the ease and benefit of source-level APR, applied to a binary.

4 Partially Recompilable Decompilation

Figure 1 depicts PRD's architecture. PRD takes as input a small set of binary functions, which are processed in three interdependent stages, each customized to preserve the executional integrity of the original binary. CGFL identifies the relevant set of suspicious binary functions. Partial decompilation generates source code that is structured to facilitate recompilation (application binary interface, compiler idiosyncrasies, etc.), and partial recompilation creates the corresponding binary patch (*PRD recompiled code*). Finally, binary rewriting integrates new binary content into the existing binary, by appending a new segment and overwriting original functions with custom detours. PRD succeeds if its output binary is test-equivalent to the original binary (assuming no source-level modifications).

Our prototype implementation operates on x86 statically- and dynamically-linked Linux ELF executables. The following subsections discuss key PRD concepts: coarse-grained fault localization, partial decompilation, partial recompilation, and binary rewriting.

4.1 Coarse-Grained Fault Localization (CGFL)

To reiterate, we do not need to our FL to be fully precise (i.e., identify specific lines of suspicious code). Rather, we require FL that can identify a small set of functions which are likely to contain the vulnerability. Our approach to FL, called CGFL, focuses on this minimal requirement.

Although PRD operates on a small number of *implicated functions*, binary repair cannot succeed unless the FL includes the vulnerable functions. This context leads to a distinct set of requirements from those of most FL methods (Section 2.2): (**r1.1**) does not require source code or the ability to recompile, (**r1.2**) prioritizes functions for decompilation, and (**r1.3**) identifies vulnerable functions. For performance, we add two more requirements: (**r1.4**) avoids functions that cannot support our detouring and (**r1.5**) minimizes overhead to obtain function set. Later analyses can use finer-grained FL to pinpoint suspicious statements.

CGFL uses *Valgrind*'s [30] *callgrind* to trace the program under different unit tests (satisfying r1.1–2,r1.5). Valgrind can efficiently provide function level traces, allowing us to identify function-level spectra. To address requirement r1.4 (detour support), we implemented a screening algorithm that reduces the probability of overruns and, for statically-linked programs, culls standard library functions. To avoid detour overruns, we set a minimum function size that supports at least four references per detour. This eliminates very small functions with large calltrees (Section 4.2.1).

For each such function, we calculate suspiciousness scores using five state-of-the-art SBFL metrics (Tarantula [31], Ochiai [32], op^2 [33], Barinel [34], D^2 [35]). Using these scores, RankAggreg identifies the top-$\mathcal{K}$ suspicious functions [20] (satisfying r1.3: This implicates vulnerable functions in the top 35% 94.6% of the time, which is sufficient for our purposes and is consistent with current decompiler capabilities [2]). This generates CGFL, a list of suspicious functions consolidated from the individual SBFL metrics. Using CGFL can reduce the size of search space by up to 95% in later analyses.

4.2 Partial decompilation

This section discusses partial decompilation, including what is required to use its output in PRD and salient implementation details of our prototype. Figure 3 outlines the high-level execution flows that PRD enables between binary content and PRD recompiled code.

4.2.1 Constraints on PRD recompiled code

Because the original binary headers and tables are left intact, standard interpreter initializations (constructors, symbol resolutions and relocations) are bypassed in PRD recompilation. We ensure that the PRD fulfills these two requirements: (**r2.1**) does not require the interpreter and (**r2.2**) references global, local, and external symbols in the original binary.

4.2.2 Binary-Source Interfaces

We analyze both the original and decompiled content, then generate two complementary binary-source interfaces: *unbound symbol* and *detour*. Shown in Listing 2, these interfaces allow decompiled code to reference symbols (like callbacks) from the original binary, regardless of binding state, by address (respective table entry or relative location). The unbound symbol interface manages dynamic interactions with symbol tables, procedure linkage tables (PLT), relocation tables (REL), and global offsets tables (GOT). Note, position-independent code (PIC) requires the `ebx` register to contain the GOT value, used during calls via PLT GOT pointers. In essence, this interface wraps callbacks with code that enables the expected dynamic linking behavior for function symbols. Similarly, the detour interface manages all required symbols and registers used in the function and serves as the entry point for the recompiled function. Binary rewriting adds instructions to resolve the relative locations of these symbol functions and variables to their offset or table entry (see Section 4.4). Together, they ensure that the resulting recompiled code executes in a manner consistent with the original.

Listing 2 shows an example detour interface with added `void*` parameters, harboring values for `ebx` and three references: `cgc_receive`, `cgc_memcpy`, and `cgc_calloc`. Because `cgc_memcpy` and `cgc_calloc` are local symbols at computable locations, they can be invoked as callbacks from the decompiled code. Symbol `cgc_receive` has an unknown binding

state; so, we create an unbound symbol interface to allow the decompiled code to call it using its table entry (PLT). In Listing 2, lines 8 of the original binary's function call and lines 10-13 of the detour interface have diverged, where det_cgc_read_line has added four void* parameters). Thus, a single jump instruction, e9, will not suffice for *detour* F'_3, the instructions overwriting the original binary function (F_3).

Divergence from the original function call to the detour interface has some implications: (**i1**) our detour interface is not compatible with the original function call; (**i2**) the stack state is not consistent upon return from decompiled function; (**i3**) added symbol references incur a byte-cost which may overrun the original function. Section 4.4 explains how PRD's binary rewriting phase handles (i1) and (i3). PRD decompilation addresses (i2) by inserting stack-correcting inline assembly before the detour interface's return. Relevant to (i3), when multiple functions are decompiled and aggregated, the minimum set of required references for any entry function is the union of its calltree's references.

This approach satisfies requirements **r2.1** (the binary-source manages references) and **r2.2** (the decompiled code uses the original binary's symbols).

Listing 2. Example **binary-source interfaces** for a detour interface ($\texttt{Bin-Src}_{\texttt{dec}}$(det_cgc_read_line)) and local symbol callbacks ($\texttt{Bin-Src}_{\texttt{local}}$(cgc_calloc,cgc_memcpy)).

```
// Bin-Src(local) : typedef function ptr + local variable
typedef void * (*pcgc_memcpy)(void *, const void *,
     cgc_size_t);
pcgc_memcpy cgc_memcpy = NULL;
// Bin-Src(local) : typedef function ptr + local variable
typedef void * (*pcgc_calloc)(cgc_size_t);
pcgc_calloc cgc_calloc = NULL;
// Decomp : Decompiled Function Declaration
cgc_ssize_t  cgc_read_line(int fd, char **buf);
// Bin-Src(dec) : Detour Interface
cgc_ssize_t  det_cgc_read_line(
    /* Bin-Src params*/ void* EBX, void* mycgc_receive,
     void* mycgc_calloc, void* mycgc_memcpy,
    /* Decomp params */ int fd, char * * buf
){
    cgc_ssize_t retValue;
    origPLT_EBX = (unsigned int) EBX;
    z__cgc_receive = (pcgc_receive)(mycgc_receive);
    cgc_calloc = (pcgc_calloc)(mycgc_calloc);
    cgc_memcpy = (pcgc_memcpy)(mycgc_memcpy);
    retValue = cgc_read_line( fd, buf);
    asm( "mov    -0xc(%ebp),%eax\n\t"
         "mov    -0x4(%ebp),%ebx\n\t" "nop\n\t"
         "add    $0x14,%esp\n\t" "nop\n\t"
         "pop %ebx\n\t"
         "pop %ebp\n\t"
         "pop %ecx\n\t"
         "add $0xc,%esp\n\t"
         "push %ecx\n\t"
         "ret\n\t"
    ); /* stack-correcting ASM - removes Bin-Src params*/
    return retValue;
}
```

Listing 3. Example **binary-source interfaces** for external symbol callbacks ($\texttt{Bin-Src}_{\texttt{plt}}$(cgc_receive)), continued from Listing 2.

```
// Bin-Src(plt) : PLT register
unsigned int origPLT_EBX = NULL;
// Bin-Src(plt) : typedef function ptr
typedef int (*pcgc_receive)(int s_0, int s_1, int s_2, int
     s_3);
pcgc_receive z__cgc_receive = NULL;
// Bin-Src(plt) : Unbound Symbol interface
int cgc_receive (int s_0,int s_1,int s_2,int s_3) {
    pcgc_receive lcgc_receive = z__cgc_receive;
    int ret;
    unsigned int localEBX;
    unsigned int localorigPLT_EBX = origPLT_EBX;
    asm ( "movl %[LOCALEBX],%%ebx\n\t"
          "movl %%ebx,%[PLT_EBX]\n\t"
          :[LOCALEBX] "=r"(localEBX)
          :[PLT_EBX] "r"(localorigPLT_EBX)
          : "%ebx");
    ret = lcgc_receive(s_0,s_1,s_2,s_3);
    asm ( "movl %%ebx,%[LOCALEBX]\n\t"
          :[LOCALEBX]"=r"(localEBX));
    return ret;
}
```

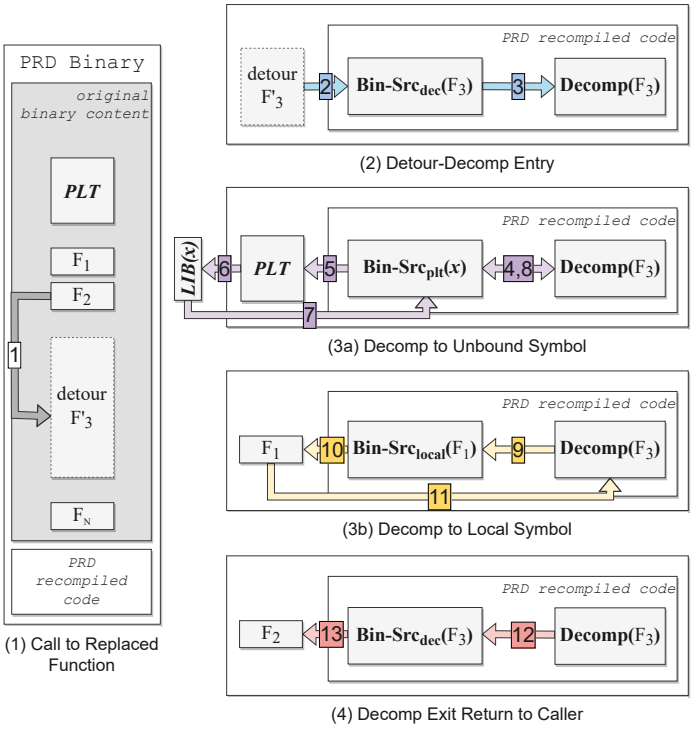


Fig. 3. Program Execution Flows for patching F_3, with each flow between original and new binary content outlined by way of numbered arrows. $F_{1...N}$ refer to the original binary's functions and its *PLT*. PRD recompiled code consists of *Decomp*(F_3), the decompiled content for function F_3, and *Bin-Src*$_{..}$, binary-source interfaces. Subscripts *dec* specify the detour interface to decompiled function, *plt* the unbound symbol interface for a dynamically resolved symbol, and *local* the interface for a local symbol, whose location is resolved. See Listings 2-3 for code examples.

4.2.3 *Decompilation.*

Here, we outline our use of decompilers and their output.

Function-specific Decompilation. For PRD to support binary repair, not all binary functions need to be decompiled. Instead, we apply decompilers to a relevant subset generated with CGFL (Section 4.1). The decompiled output is left intact, with the exception of a small set of decompiler-specific keyword substitutions required for APR and subsequent recompilation (e.g., replacing DWORD with unsigned int).

Compatible Type Recovery. Similarly, it is not necessary to recover exact types from the binary, only *compatible types* are needed. For example, a struct foo may contain many fields with different types, but only one of the fields (e.g., foo.bar where bar is an unsigned int) is used in a decompiled function. PRD only needs to know the offset and inferred type of bar. This significantly reduces the requirement for complete and sound type inference on binary

code, an open research problem. To accomplish compatible type recovery, we leverage the decompiler's type inference system to reconstruct the necessary types. Notably, although decompilers can fail to recover all types, PRD can succeed if only the referenced compatible types are defined. Since types can be nested, PRD decompilation resolves the definition order for the compatible type definitions.

4.2.4 Implementation

Our prototype is implemented in Python with LIEF [36] and uses the Hex-Rays Decompiler, supplemented with Ghidra to support C++-decompilation when Hex-Rays recompilation fails. The decompilers generate the initial decompilation, and our custom IDAPython script obtains corresponding local type and function declarations.

PRD substitutes common primitives such as `_DWORD` and decompiler-specific definitions, resolves the definition order for types, generates the required binary-source interfaces and resolves the minimum set of required symbols. These rule-based transformations are a best-effort heuristic to produce informative decompilation. To reorient the stack to support detouring with references, we analyze the initial recompiled PRD source, generate inline assembly that reconstructs the stack on detour exit, then inserts the assembly snippet in the detour interface (see lines 20-28 of Listing 2).

4.3 Partial Recompilation

Partial recompilation recompiles PRD decompiled code respecting r2 (Section 4.2.1). Because our binary rewriting strategy appends new binary content, it must operate even if the memory address of the new content is not known in advance. To satisfy these requirements, PRD creates position-independent, statically linked content. Although `PIC` is ubiquitous, support for both static linking and `PIC` in a single shared object is recent ($\geq$ GNU 8.4.0) and not behaviorally consistent across all compilers. We generate our object with these flags and custom linker script, placing all sections in a single segment. While default partial recompilation uses `GCC`, we support APR profiling (which often leverages `libc` functions not always available in the original binary) with *dietlibc*, a small-footprint libc. Our approach supports executional requirements and r2.1.

4.4 Binary Rewriting

Binary rewriting composes the original binary and PRD-decompiled source into a single binary that executes correctly. Our prototype analyzes, extracts, adds, and manipulates binary content to satisfy the requirements r2.1-2. To handle (i1) (Section 4.2.1), we overwrite the existing binary function by adding instructions that, if any symbols are required, change the effective function call to align with the detour interface, and a jump to the detour interface, i.e., *detour F'3* from Figure 3. To accomplish this, we analyze the binary content to generate instructions for each required reference, resolving each symbol (function or variable) to its respective offset or table entry, adhering to calling convention. These added instructions incur a byte-cost for each additional reference which may overrun the original function (Section 4.2.2) (i3).

5 Experimental Setup

Our evaluation of PRD and its application in APR (BinrePaiReD) includes evaluations of underlying assumptions, end-to-end fully automated scenarios, and two real-world case studies (Section 6.5.2). Specifically, our evaluation addresses the following research questions:

RQ1. Does CGFL identify function(s) relevant to a given vulnerability?

RQ2. Without any restrictions, how often is decompiled code recompilable?

RQ3. Is decompiled code behaviorally consistent to original binary functions?

RQ4. How effective is BinrePaiReD at mitigating vulnerabilities?

Case Study: Generality. Does PRD generalize to real-world vulnerabilities, other languages, and performance constraints?

Next, we describe our datasets and experimental setup.

5.1 Benchmark Datasets

Table 1 shows the four benchmark datasets used in the evaluation: DARPA Cyber Grand Challenge C binaries (`CGC-C`) and C++ binaries (`CGC-C++`); Rode0day 19.11 (`Rode0day`); and vulnerable, real-world programs (`CVE Case Study`).

5.1.1 CGC-C and CGC-C++

The DARPA 2016 Cyber Grand Challenge (CGC) provides a dataset of challenge binaries (CBs), each containing realistic vulnerabilities, and a testing framework. We derived our CGC datasets from a Linux variant, *trailofbits* [37], verified these CBs using a robust variant of its testing environment: Python3, instrumentation support, extended signal handling. This identified 110 valid CGC CBs (100-`CGC-C` and 10-`CGC-C++`), each with at least nine passing positive and one negative tests (defect scenario). We consider *target*s for each CB, i.e., a set of relevant functions for decompilation, totaling 190 as some binaries have multiple vulnerable functions.

5.1.2 Rode0day

Rode0day 19.11 inserts hundreds of bugs into Linux binaries, each a collection of stripped binaries and example test inputs. We recompile to ensure that symbol names exist in the binary and generate unit tests from provided inputs (see Table 1). For our evaluations, we consider both functions and the injected bugs.

5.1.3 Real-World Case Study

We also evaluated the viability of manually generating source-level changes on real-world programs. These three programs were not specifically curated for APR or binary analysis, two taken from public security vulnerabilities (CVEs), `podofopdfinfo` [38] and `jhead` [39], and a raw binary obtained from a docker image, `/bin/bash`. Because the vulnerable methods are local symbols, PRD can repair the binary, a more difficult task than repairing libraries.

TABLE 1
Evaluation Dataset Features. Release, number of binaries, number of defect scenarios, average and minimum number of behavioral (pos) tests, compiler-toolchain and source language, average lines of code (LOC). Line coverage % values are reported for average, std-dev, and median for Curated Datasets and CGFL.

	Curated Dataset			Real-World Case Study		
id	CGC-C	CGC-C++	Rode0day	CVE 2021-30472	CVE 2021-3496	/bin/bash
release	trailofbits	trailofbits	19.11	PoDoFo 0.9.7	jhead 3.0.6	i386/ubuntu 18.04
bins	100	10	4	1	1	1
defect scenarios	157	10	927	6	1	
avg(min) pos	104(9)	89.8(13)	1.5(1)	36	68	
Compiler	GNU	GNU	GNU	GNU	Clang	GNU
Lang	C	C++	C	C++	C	C++
avg. LOC	37,670	1,138	84,725	47,414	4,203	
line-cov						
%avg	66	75	13.8			
%std-dev	29.9	13.2	4.1			
%med	77	79	13.6			

5.2 External Tools

PRD uses Hex-Rays IDA Pro 7.5 SP2 (Hex-Rays), and GCC 8.4.1 with dietlibc used for APR. When handling `C++` for APR, we augment Hex-Rays failures with Ghidra 10.0.1.

BinrePaiReD, uses Valgrind for CGFL and supports two APR tools: Prophet (v0.1, Clang-based) and GenProg (v3.2, CIL-based). We used Prophet's default search algorithm with its *profile* localizer and three of GenProg's: genetic algorithm ("GA"), deterministic search ("AE") [27], and a "single-edit" repair search. When evaluating program variants, GenProg substitutes standard compilation with PRD. Prophet substitutes the compiler with custom scripts, relying on environmental variables and dynamic libraries (each is compatible with PRD). Ultimately, PRD operates seamlessly with both APR tools.

6 Empirical Evaluation

In this section, we present the results of our evaluation and explain how they answer the research questions. The first three subsections evaluate underlying assumptions and the remaining three evaluate performance.

6.1 RQ1: Does CGFL identify function(s) relevant to the vulnerability?

After confirming compatibility with our implementation, we used the three dataset's test-cases as stimuli and used annotations for ground truth (i.e., vulnerable functions that should be implicated).

For all binaries we studied, the results show that CGFL output contains at least one ground-truth function for 95 of 100 `CGC-C`, 8 of 10 `CGC-C++`, and 196 of 206 `Rode0day` , which succeeds despite having few tests (Section 5.1.2).

Although CGFL achieved 94.6% success under our criteria, we observed three failure types that can be easily explained or mitigated. First, 14 binaries (4 `CGC`/10 `Rode0day`) did not exercise any vulnerable function in negative tests (f.1). Second, 10 were in the first three ranks, but ties affected their selection (f.2), a common failure in SBFL metrics. Third, one involved a buggy reimplementation of a libc function (f.3). Although (f.1) is out of scope for both SBFL and test-driven APR, (f.2) is readily mitigated with improved tests or increaseing $\mathcal{K}$ (Section 4.1). Finally, (f.3) is a consequence of our simple heuristic which screens out known library functions and could be replaced with more sophisticated screening.

TABLE 2
RQ2 (success rates for Type Recovery, Decompiler and Basic Recompilation) and RQ3 (success rates for PRD Recompilation and Test-equivalency) evaluation results, including failure rates from Type Errors (proportion of all recompilation errors). Reported numbers are per function, except Type Recovery, which is per binary. *Basic* and *PRD Recompile* are without/**with**% typing errors.

	RQ2: Partial decompilation				RQ3: Partial recompilation	
Dataset	Success			Failure	Failure	Success
	Type Recovery	Decompiler	Basic Recompile	Typing Errors	Typing Errors	PRD Recompile	Test Equivalent
CGC-C	54.0%	99.9%	89.5/**79.1%**	54.6%	56.6%	89.9/**79.4%**	92.1%
	54/100	7060/7067	5585/6237	823/1507	823/1455	5605/6237	5161/5605
CGC-C++	0.0%	87.8%	72.7/**34.5%**	80.3%	78.7%	70.4/**33.3%**	97.8%
	0/10	1099/1251	379/521	578/720	578/732	367/521	357/365
Rode0days	50.0%	100.0%	51.5/**22.4%**	81.5%	82.5%	71.8/**30.8%**	96.4%
	2/4	3297/3297	738/1432	1864/2289	1880/2280	1016/1416	979/1016
Total	49.1%	98.6%	71.2/**45.3%**	72.3%	73.4%	85.5/**47.8%**	93.0%

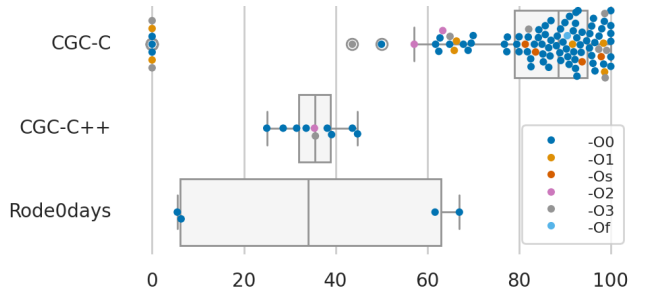


Fig. 4. Basic recompilation success rate per binary, with optimization.

> Even when few tests are available, coarse-grained fault localization works well in practice and contained the relevant function in 299/316 cases.

6.2 RQ2: How often is decompiled code recompilable?

We studied the `CGC-C`, `CGC-C++`, and `Rode0day` datasets with no restrictions on compilation or on the source code grammar. We evaluated four aspects of decompilation: type recovery, decompiler success, basic recompilability, and full recompilability. For type recovery we considered each binary independently, reporting success only if *all* specified types were recovered. Decompiler and recompilability evaluations considered each function independently and asked how many could be decompiled by Hex-Rays and recompiled. If Hex-Rays failed, then decompiler failed for that function. Basic recompilation was measured per function and fails if any dependency is not fully defined, i.e., issues with compatible type recovery, prototype recovery, or decompilation. We evaluated basic recompilation on the raw decompiler output for function prototype recovery and function content. However, the raw recovered type output is rarely well-ordered, so we used PRD-transformed types for assessing Typing Errors (Section 4.2.3).

Table 2 and Figure 4 demonstrate that decompilation is surprisingly successful and that basic recompilation is affected more often by type failures rather than by optimization level. In total, 22.4% to 79.1% of functions succeeded at basic recompilation while only two complete binaries were successfully recompiled (`CGC-C`: Palindrome, Palindrome2).

> We find that only 1.7% of binaries are fully recompilable, but up to 79.1% of individually decompiled functions are recompilable. This strongly supports the insight to use *partial*, instead of full decompilation.

TABLE 3
APR Comparison: Full-source (baseline) vs. BinrePaiReD with PRD decompiled code (PRD). We report the number of scenarios that produced a plausible mitigation (*mitigated*), the *total* number that the APR tool successfully launched its search, as well as the number which the tool *completed* its search within 8 hours.

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

6.3 RQ3: Is decompiled code behaviorally consistent to original binary functions?

Each PRD binary was evaluated for test equivalency using the provided test-cases (Table 1) and compared to the original binary. Any behavioral disparity is considered a failure, and proof-of-vulnerability tests must both mitigate the vulnerability and be test equivalent.

The results (Table 2) show that PRD's recompilations are consistent with basic recompilation (Section 6.2), i.e., PRD decompilation transformations generally do not introduce errors and are similarly impacted by typing errors. PRD recompilation sometimes outperforms basic recompilation, because e.g., if the decompiler recovers a static global variable's initial values, PRD only needs its reference and type. The results also show that over 92% of PRD's binaries were test-equivalent to the original. We also evaluated the behavior of each PRD binary, finding that only two binaries, the same two from RQ2, could be fully reconstructed and generate test-equivalent PRD binaries for all functions.

When recompilation succeeds, PRD produces patched binaries that are test-equivalent to the original. PRD provides a solid foundation for source-level transformations.

6.4 RQ4: How effective is BinrePaiReD at mitigating vulnerabilities?

We consider two scenarios: Plausibility, i.e., can APR tools leverage partially decompiled content to find repairs?; and Content, i.e., does the form and quality of decompiled affect the success of APR tools?

6.4.1 Plausibility.

We compared the success rate for APR tools applied to the actual source code that produced the binary (baseline) to the success rate for the same APR tool applied to PRD partially decompiled code. Our primary evaluation features an end-to-end use of BinrePaiReD, depicted in Figure 2). CGFL implicates the relevant functions, which are individually decompiled to create PRD decompiled code, which is input to the APR algorithm. To discover repairs, the APR algorithm uses PRD tools to apply candidate source-level patches to the original binary evaluates the resulting PRD binary's behavior using the provided tests. Unlike BinrePaiReD, the baseline has access to the complete source.

Using the 157 defect scenarios from the `CGC-C` dataset, we assessed baseline and PRD-enabled APR algorithms, limiting each run to 8 hours (Table 3). The results show that PRD-enabled repair, operating only on binaries, performs as well as and sometimes better than full-source APR: 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-supported APR tools mitigated 85 of the 148 scenarios (including 20 that the winning CGC cyber-reasoning system, Mayhem [40], did not patch). Overall, we find that the success rate for repairs generated by BinrePaiReD are as good as those produced by the same tools operating on full-source ($p < 0.0004$, proportions z-test). Because the baselines have access to the complete source code, we expected that BinrePaiReD would succeed less often because it has much less source code to use to help generate repairs, e.g., less code that can be copied as part of a mutation. Although surprising, it can be explained by the fact that that decompiled source code is a better fit with typical APR operators than original source code [41], which allows the APR operators to take more advantage of code elements.

6.4.2 Case Studies: Repair Quality

It is well-known that APR tools can find repairs that overfit the test suite [42], [43] without addressing the root cause of the problem. We do not improve or worsen that orthogonal concern here. Instead, we find that BinrePaiReD inherits the repair quality of its underlying APR method. In our use case, quickly disrupting an exploit is valuable, even if the repair is not completely general. In the following, we focus on the GenProg results for simplicity, but Prophet's results were similar.

First, in cases where GenProg failed to produce a mitigation, the required edit was usually out of scope, e.g., changes to struct fields or unused variables. Other failures involved special constant values or comparators, which are known tool weakness (cf. [44]). Second, when GenProg does find a successful mitigation, it often finds multiple solutions. We randomly sampled 5–10% of the solutions and examined their `C` representations (recall from Section 3, BinrePaiReD retains source-level patches, facilitating such analyses). We do not find significant differences in the rate of overfitting between the full-source APR baseline and BinrePaiReD. As a case study, we describe one example of overfitting and one example that generates a general solution.

Lower-Quality Mitigation. For `KPRCA_00013`'s first vulnerability, GenProg's single mutation solution passes all tests and successfully mitigates the POV. In this edit, the "`(`" character is pushed onto the operator stack in a loop, and the next iteration flags an error since the top of the stack is "`(`". Although this patch does not address the intended vulnerability it successfully blocks this particular exploit, preventing control of the next heap block's heap metadata.

Higher-Quality Mitigation. `NRFIN_00076`'s defect simulates a vulnerability introduced when a programmer commits unfinished code. The program incorrectly increments `*results` in a frequent function, leading to the use of an invalid pointer. The repair found by Genprog correctly deletes the problematic code, eliminating the pointer issues.

In an end-to-end scenario, BinrePaiReD with PRD support can use a source-level APR tool to repair binaries with results that *are consistent with and sometimes better*

than those using the same techniques on the full source. This is true both in terms of the rate at which vulnerabilities are mitigated and in terms of repair quality.

6.4.3 How Does Partial Decompilation Affect APR?

To assess the impact of decompilation on overall results, we studied a random sample of 21 CGC CBs and their 30 scenarios. We consider three variations: access to the full program source (*baseline*), the PRD-provided decompiled source for implicated methods (*PRD*), and (*exact*) where the CGC-provided source is used for the same implicated methods, i.e., the exact source function replaces the decompiled function. GenProg and Prophet separately produce 28 candidate patches with the *baseline*, 18 with *PRD*, and 34 with *exact* decompilation. This result echoes the RQ2 finding that state-of-the-art decompilation tools have room to improve in end-to-end usage scenarios.

Additionally, the difference in *lines of code* (LOC) offers helps explain why exact decompilation can perform comparably to full source. Since decompilation applies only to the implicated functions (reducing effective LOC by up to 95%). This supports our insight that CGFL, if accurate at implicating a relevant subset, can help downstream stages, including decompilation and program analysis/transformation. current decompilers' weaknesses are mitigated by fault localization filtering.

Decompilation quality can strongly affect BinrePaiReD, but current decompilers' weaknesses are mitigated by CGFL fault localization filtering.

6.5 Case Study: Generality

The previous results demonstrate PRD's effectiveness, but for a binary-facing technique it is also important to consider multiple languages, real-world programs and defects, and execution overhead.

6.5.1 Application to `C++`.

PRD can partially decompile and recompile binaries produced from `C++` (Table 2) for individual functions. Because Prophet and GenProg don't support C++, we cannot directly compare to baseline APR performance. Rather, we conduct an end-to-end study of the `CGC-C++` examples in our dataset, using each example's CGFL results as the decompile set. For each binary, we applied PRD to its CGFL, totaling 149 functions (including decompiled `C++` class methods). Although test-equivalent PRD binaries were generated for 7 of the 10 binaries, recompilation failures (94) dominated test (10) and decompilation (9) failures, similar to the results of Section 6.3. These results show that the PRD framework is extensible to `C++` binaries and can potentially support automated repair of `C`-like binaries, if a `C`-based APR tool were available. Because APR tools overwhelmingly favor other languages than `C++`, these results also improve the applicability of `non-C++` APR tools.

6.5.2 Real-world Vulnerabilities.

We detail PRD and CGFL effectiveness for two real-world programs that contain known vulnerabilities (Table 1).

CVE-2021-30472: For tests, we used the developer's recommended example PDFs and the CVE input, `bug4` for `podofopdfinfo`. This exploit hits a vulnerability in `PdfEncryptMD5Base::ComputeEncryptionKey` (`f1`) before the `PdfEncryptMD5Base::ComputeOwnerKey` (`f2`) vulnerability. We applied CGFL, and all 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 both methods. Because Hex-Rays incorrectly lifted the stack canary using x86-64 content, we manually corrected this problem by adding equivalent x86 inline assembly. Using the `GCC-G++` compiled binary and this content, PRD successfully generated a test-equivalent binary for `f2`, and mitigated the vulnerability by modifying `f1`. Essentially, the decompiler generated a large number of local variables for `f1`, which changed the stack in a way that closed the vulnerability.
CVE-2021-3496: We used the same process for `jhead`. CGFL implicated the function, `ProcessMakerNote`, as rank 3 of 42. Notably, it inlines the reported buggy function `ProcessCanonMakerNoteDir`. PRD successfully generated a test-equivalent binary from the Clang binary.

By manually applying the necessary bug fixes (less than the lines were changed), we produce PRD-binaries that both pass all tests, resolving the reported bugs for both CVEs, successfully applying PRD to two real-world issues.
Source-less Patching: We next applied PRD to a `bash` binary obtained directly from a docker image, successfully patching the *builtin* function, `false`, thus demonstrating applicability to a `GNU-C++` binary without access to the source code.

6.5.3 Performance Analysis.

Using *perf stat*, we measured runtime and compile-time performance for PRD binaries. For runtime, we compared 100 PRD binaries to their counterparts over 5,163 tests and found no statistical difference in their performance ((user: $p < 0.970$; system: $p < 0.277$, two-tail t-test)). For compile time, an overhead relevant to APR algorithms, we sampled 25 CGC CBs and their respective single-detour PRD binaries, generating each 25 times. We found that generating a PRD binary is statistically less expensive than compiling the binary from source (user:$p < 0.0$; system: $p < 8.4845e^{-192}$, two-tail t-test). These results suggest that PRD does not induce a performance overhead on APR tools.

PRD can be applied to `C++` binaries and real-world CVEs. The ability to produce test-equivalent binaries for the `C++` examples we studied is comparable to the rate we observed for `C`. We find that both CGFL and PRD are successful on the real-world CVEs we tested. We find negligible overhead associated with producing and running PRD binaries.

7 Discussion

After initial analysis to identify a relevant set of vulnerable functions for binary patching, PRD supports both manual and automated mitigation of binary-level exploits. PRD enables source-level modifications of lifted content, and then generates a new binary that executes the modified code instead of the original.

Test suite quality can affect the accuracy and precision of our FL at the binary level. However, Our CGFL evaluation of Rode0days achieved a 95% success rate despite a severely limited test suite. By identifying a small function set, our CGFL strategy overcomes current limitations of decompilers, reducing LOC to be decompiled by an average of 95% (Section 6.4.3) over the complete binary.

It is surprising that the BinrePaiReD results we observed were consistent with those obtained with APR that has access to full source, since the methods we considered both obtain the seeds of repairs ("fix" locations) from the rest of the program [6], [7]. Our subsequent analysis showed that decompiled code structure is often more consistent with APR operators than the original source [41]. It is also likely that by restricting the number of vulnerable functions the size of the search problem is reduced in a way that aids the APR search, a topic for future research.

Although decompiled code is less readable than the original source, it is more readable than assembly (for manual analysis) and not an impediment at all for APR. PRD enables C-based APR tools to be extended to C++ binaries with decompiling into C-like source (Section 6.5.1).

7.1 PRD Limitations and caveats.

Tools that use whole-program analysis, such as symbolic execution used by Angelix [45], and interpreted languages cannot be supported by BinrePaiReD and PRD. Although our implementation focuses on 32b ELF and System-V, PRD is compatible with other binary formats and ABIs, assuming that calling conventions are upheld, decompiler support is available, and engineering effort. Because PRD only uses symbol names to map symbols to their entries and decompiled outputs, PRD is compatible with stripped binaries, assuming decompiler support and some engineering. PRD is compatible with ASLR binaries but doesn't handle self-modifying or self-checking binaries. PRD supports recursive functions and local function pointers, but may require specialization for other binary constructs, e.g., extending debug to PRD recompiled code. In binary rewriting, stack-unwinding could be supported by updating call frame information tables; variable support can be extended to thread-local storage and copy relocations by referencing their respective ELF structures. The most serious challenge for BinrePaiReD arises from the limitations of current decompilers (e.g., standard C content is not always generated). In our evaluation, failure to generate test-equivalent PRD binaries was usually caused by decompilation failures, and our transformations are limited to currently known and mitigated decompiler weaknesses. Our evaluations also found some brittleness, particularly with compiler-instrumented binaries and type recovery. The motivating example (Section 3) and the end-to-end repair results (Section 6.4) demonstrate the full pipeline with Hex-Rays.

7.2 Decompilation failures.

Although decompilation techniques have improved dramatically, 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 decompilation will enhance PRD's generality and quality.

7.3 Unsound decompilation results.

Decompilers can generate decompiled code that changes the semantics of the original binary. While determining the equivalence of two arbitrary binaries is undecidable in general, this problem could be addressed by requiring byte-level equivalence between the original binary code and the recompiled, unpatched code [4]. Our evaluation strategy resembles Equivalence Modulo Input testing [46] and assumes adequate coverage. We validated generated binaries against existing test cases (coverage in Table 1). Although they are sufficient for FL and APR, test generation could improve coverage and confidence in equivalency checks.

8 Related Work

Three topics most relevant to PRD and BinrePaiReD are: binary patching and rewriting, APR, and binary code decompilation.

8.1 Binary Patching and Rewriting

There are two main binary rewriting approaches: dynamic and static. Dynamic binary rewriting, or dynamic binary instrumentation, inserts user code at specified binary locations at runtime, e.g., Pin [47], Valgrind [30], and DynamoRIO [48]. These techniques can introduce prohibitively high overhead and are unused for production binary patching. Static binary rewriting techniques, like Egalito [49] and LIEF [36], perform code transformation and relocation before execution with much lower runtime overhead than dynamic methods. These methods are more suitable for generic tasks like binary patching or control-flow integrity enforcement. Ramblr [50] and ddisasm [51] convert binary code into assembly 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 [52]. These methods do require additional effort to craft repairs to patch binaries. CGC finalists used either in-place binary editing or reassembly to apply patches [40], [53], [54]—PRD could easily have been used in these contexts. No existing solutions transform binary code to usable high-level source.

8.2 Automated Program Repair

APR is an established research area [22], [23], [55], and the vast majority of tools and methods operate directly on source code. Some exceptions are Schulte *et al.*'s papers [56]–[58] and Orlov and Sipper's early explorations with Java bytecode [59], [60], both of which operate directly on lower-level representations without lifting to source. Other tools, e.g., RSRepair [61], Kali [62], and SPR [63], are all compatible and would require modifications similar to those we made for our tested tools. Other tools, like CodePhage [64] and CodeCarbonCopy [65], are compatible with PRD but would require more extensive modifications. Similarly, ML source-based repair methods, like CoCoNut [66] or VulRepair [25], may be compatible with PRD. These methods are largely trained on human-generated source code (may not be available), rely on external evaluations for repair correctness, and require perfect fault localization, i.e., the buggy location is annotated in the input. While Angelix [45] operates on source code, it uses symbolic

execution and therefore is incompatible with our approach. OSSPatcher [67] targets third-party, open-source libraries for automatic binary patching, but it requires both source and source-based patches.

8.3 Binary Code Decompilation

Progress in binary code decompilation relies on advances in binary code extraction, (control flow) structural analysis, and 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 [51], [68], [69]. Structural analysis has progressed significantly: Schwartz *et al.* reduced the number of `goto` statements [70]; and Yakdan *et al.* proposed pattern-independent control-flow structuring to eliminate `goto` statements and improve readability [5]. Due to their requirement in code coverage (e.g., [11], [71], [72]), decompilers often use static analyses or type inference. Recent progress in decompilation enabled the *recompilation* of decompiled code—considered impossible by most researchers until recently. For example, Liu *et al.* show that the output of modern C decompilers is generally recompilable [2] when grammar and types are restricted. Similarly, Harrand *et al.* present a method that mitigates Java decompiler failures by merging outputs [73]. Both examples confirm that decompilers sometimes generate incorrect output.

9 Conclusion

Security-critical vulnerabilities that arise after software is deployed must be addressed quickly, even when full 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 [74]. We present a new way to patch binaries when recompiling from complete source is not an option. Although full-source decompilation remains a challenge, we show that it generates recompilable code for most functions. By focusing on only the vulnerable functions, state-of-the-art decompilers can produce recompilable code that is amenable to source-level code repair tools. We introduce CGFL to identify a set of suspicious functions, partial decompilation to lift relevant sections of the binary to source where repairs are developed and applied, and finally generate a PRD binary addressing the problem. Our implementation and datasets are available at https://github.com/AdaptiveComputationLab/FuncRepair.

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

Acknowledgment

We would like to thank our anonymous reviewers for their valuable feedback. The authors gratefully acknowledge the partial support of NSF (CCF 1908633, OAC 2115075), DARPA (FA8750-19C-0003, N6600120C4020), and the Santa Fe Institute.

References

[1] M. Monperrus, "The living review on automated program repair," HAL/archives-ouvertes.fr, Tech. Rep. hal-01956501, 2018.

[2] Z. Liu and S. Wang, "How far we have come: testing decompilation correctness of C decompilers," in *Intl. Symposium on Software Testing and Analysis, ISSTA*. ACM, 2020.

[3] M. Botacin, L. Galante, P. de Geus, and A. Grégio, "Revenge is a dish served cold: Debug-oriented malware decompilation and reassembly," in *Reversing and Offensive-Oriented Trends Symposium*. ACM, 2020.

[4] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Loginov, "Evolving exact decompilation," in *Workshop on Binary Analysis Research*. Internet Society, 2018.

[5] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations," in *Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2015.

[6] F. Long and M. C. Rinard, "Automatic patch generation by learning correct code," in *SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. ACM, 2016.

[7] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, 2012.

[8] J. Song and J. Alves-Foss, "The darpa cyber grand challenge: A competitor's perspective," *IEEE Security Privacy*, 2015.

[9] "Rode0day," , accessed: 2022-07-20.

[10] A. Gussoni, A. D. Federico, P. Fezzardi, and G. Agosta, "A comb for decompiled C code," in *Asia Conf. on Computer and Communications Security, ASIA CCS*. ACM, 2020.

[11] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *Programming Language Design and Implementation, PLDI*. ACM, 2016.

[12] J. R. Larus and E. Schnarr, "EEL: machine-independent executable editing," in *Programming Language Design and Implementation, PLDI*. ACM, 1995.

[13] W. Masri, "Fault localization based on information flow coverage," *Softw. Test. Verification Reliab.*, 2010.

[14] R. P. A. de Araujo and M. L. Chaim, "Data-flow testing in the large," in *Intl. Conf. on Software Testing, Verification and Validation, ICST*. IEEE Computer Society, 2014.

[15] H. Zhu, T. Peng, L. Xiong, and D. Peng, "Fault localization using function call sequences," *Procedia Computer Science*, 2017, intl. Congress of Information and Communication Technology (ICICT2017).

[16] E. M. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, "Automated repair of binary and assembly programs for cooperating embedded devices," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 2013.

[17] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, 2009.

[18] T. Hirsch, "A fault localization and debugging support framework driven by bug tracking data," in *Intl. Symposium on Software Reliability Engineering Workshops, ISSRE Workshops*. IEEE, 2020.

[19] M. Motwani and Y. Brun, "Automatically repairing programs using both tests and bug reports," *CoRR*, 2020.

[20] S. Lin, "Rank aggregation methods," *Wiley Interdisciplinary Reviews: Computational Statistics*, 2010.

[21] S. Shen, A. Kolluri, Z. Dong, P. Saxena, and A. Roychoudhury, "Localizing vulnerabilities statistically from one exploit," in *ASIA CCS: ACM Asia Conf. on Computer and Communications Security*. ACM, 2021.

[22] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, 2019.

[23] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Software Eng.*, 2019.

[24] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Intl. Conf. on Software Engineering, ICSE*. ACM, 2022.

[25] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Q. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Foundations of Software Engineering*. ACM, 2022.

[26] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Intl. Conf. on Software Engineering, ICSE*, 2012.

[27] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Intl. Conf. on Automated Software Engineering, ASE*. IEEE/ACM, 2013.

[28] Y. Hu, Y. Zhang, and D. Gu, "Automatically patching vulnerabilities of binary programs via code transfer from correct versions," *IEEE Access*, 2019.

[29] P. Reiter, "https://github.com/AdaptiveComputationLab/FuncRepair" .

[30] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Programming Language Design and Implementation, PLDI*. ACM, 2007.

[31] J. Jones, M. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Intl. Conf. on Software Engineering, ICSE*, 2002.

[32] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Intl. Conf. on Software Engineering, ICSE*, 1994.

[33] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, 2011.

[34] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "Spectrum-based multiple fault localization," in *Intl. Conf. on Automated Software Engineering, ASE*. IEEE/ACM, 2009.

[35] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Trans. Reliab.*, 2014.

[36] R. Thomas, "Lief-library to instrument executable formats," 2017.

[37] trailofbits, "trailofbits/cb-multios: Darpa challenges sets for linux, windows, and macos," , 2017, accessed: 2021-08-07.

[38] "Podofo download — sourceforge.net," , accessed: 2021-07-28.

[39] "Matthias-wandel/jhead," , accessed: 2021-07-28.

[40] T. Avgerinos, D. Brumley, J. Davis, R. Goulden, T. Nighswander, A. Rebert, and N. Williamson, "The mayhem cyber reasoning system," *IEEE Secur. Priv.*, 2018.

[41] P. Reiter, A. M. Espinoza, A. Doupé, R. Wang, W. Weimer, and S. Forrest, "Improving source-code representations to enhance search-based software repair," in *GECCO: Genetic and Evolutionary Computation Conf.* ACM, 2022.

[42] E. K. Smith, E. T. Barr, C. L. Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Foundations of Software Engineering*. ACM, 2015.

[43] X. D. Le, F. Thung, D. Lo, and C. L. Goues, "Overfitting in semantics-based automated program repair," in *Intl. Conf. on Software Engineering, ICSE*. ACM, 2018.

[44] V. P. L. Oliveira, E. F. de Souza, C. L. Goues, and C. G. Camilo-Junior, "Improved representation and genetic operators for linear genetic programming for automated program repair," *Empir. Softw. Eng.*, 2018.

[45] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: scalable multiline program patch synthesis via symbolic analysis," in *Intl. Conf. on Software Engineering, ICSE*. ACM, 2016.

[46] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM Sigplan Notices*, 2014.

[47] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Programming Language Design and Implementation, PLDI*. ACM, 2005.

[48] "Tutorial: Building dynamic instrumentation tools with dynamorio," in *Intl. Symposium on Code Generation and Optimization (CGO 2011)*, 2011.

[49] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 2020.

[50] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," in *Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2017.

[51] A. Flores-Montoya and E. M. Schulte, "Datalog disassembly," in *USENIX Security Symposium*. USENIX Association, 2020.

[52] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Programming Language Design and Implementation, PLDI*. ACM, 2020.

[53] Y. Shoshitaishvili, A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, C. Salls, N. Stephens, R. Wang, and G. Vigna, "Mechanical phish: Resilient autonomous hacking," *IEEE Security and Privacy*, 2018.

[54] A. Nguyen-Tuong, D. Melski, J. W. Davidson, M. Co, W. H. Hawkins, J. D. Hiser, D. Morris, D. Nguyen, and E. F. Rizzi, "Xandra: An autonomous cyber battle system for the cyber grand challenge," *IEEE Secur. Priv.*, 2018.

[55] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, 2018.

[56] E. M. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, "Automated repair of binary and assembly programs for cooperating embedded devices," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 2013.

[57] E. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in *Int. Conf. on Automated Software Engineering*. ACM, 2010.

[58] E. M. Schulte, W. Weimer, and S. Forrest, "Repairing cots router firmware without access to source code or test suites: A case study in evolutionary software repair," in *Genetic and Evolutionary Computation Conf.* ACM, 2015.

[59] M. Orlov and M. Sipper, "Genetic programming in the wild: evolving unrestricted bytecode," in *Genetic and Evolutionary Computation Conf., GECCO*. ACM, 2009.

[60] M. Orlov, "Evolving software building blocks with FINCH," in *Genetic and Evolutionary Computation Conf.* ACM, 2017.

[61] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Intl. Conf. on Software Engineering, ICSE*. ACM, 2014.

[62] Z. Qi, F. Long, S. Achour, and M. C. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Intl. Symposium on Software Testing and Analysis, ISSTA*. ACM, 2015.

[63] F. Long and M. C. Rinard, "Staged program repair with condition synthesis," in *Foundations of Software Engineering*. ACM, 2015.

[64] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Programming Language Design and Implementation, PLDI*. ACM, 2015.

[65] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard, "Codecarboncopy," in *Foundations of Software Engineering*. ACM, 2017.

[66] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Intl. Symposium on Software Testing and Analysis, ISSTA*. ACM, 2020.

[67] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries," in *Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2019.

[68] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in COTS binaries," in *IEEE/IFIP Intl. Conf. on Dependable Systems and Networks, DSN*. IEEE Computer Society, 2017.

[69] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *European Symposium on Security and Privacy, EuroS&P*. IEEE, 2017.

[70] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *USENIX Security Symposium*. USENIX Association, 2013.

[71] Z. Xu, C. Wen, and S. Qin, "Learning types for binaries," in *Intl. Conf. on Formal Engineering Methods, ICFEM/FMSE*. Springer, 2017.

[72] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, "Typeminer: Recovering types in binary programs using machine learning," in *Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA*. Springer, 2019.

[73] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "Java decompiler diversity and its application to meta-decompilation," *J. Syst. Softw.*, 2020.

[74] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram, "How do fixes become bugs?" in *Foundations of Software Engineering*. ACM, 2011.

Pemma Reiter Pemma Reiter received the B.S. in Computer Engineering from Virginia Tech in 2001 and the M.Sc. in Computer Science from Arizona State University in 2019, where she is currently a Ph.D. candidate. Before joining ASU, she worked at Intel Corp. 2001-2017 as a pre-silicon validation, design, and firmware engineer, and technical lead for System-on-a-Chip products. Her research interests focus on program representation at multiple abstraction levels with a goal of improving software quality, tools, and human understanding.

Hui Jun Tay Hui Jun Tay is a PhD student at Arizona Student University, SEFCOM. They graduated with a B.S./M.S. in Electrical and Computer Engineering from Carnegie Mellon University in 2015/2016. Before pursuing their PhD, they worked for DSO National Laboratories from 2016-2019 as a Computer Security Researcher in the field of embedded security. Hui Jun's current research interests include firmware analysis, symbolic execution and automated program analysis.

Westley Weimer Westley Weimer received the BA degree in computer science and mathematics from Cornell University, and the MS and PhD degrees in computer engineering from the University of California, Berkeley. He is currently a professor of computer science with the University of Michigan. His main research interests include static and dynamic analyses to improve software quality and fix defects, as well as medical imaging and human studies of programming.

Adam Doupé Dr. Adam Doupé is an Associate Professor in the School of Computing and Augmented Intelligence at Arizona State University. He is also Director of the Center for Cybersecurity and Trusted Foundations in the Global Security Initiative at Arizona State University and the co-Director of the Laboratory of Security Engineering For Future Computing (SEFCOM). He plays CTFs with Shellphish, and as a Founding Member of the Order of the Overflow hosted the DEF CON CTF (Quals and Finals) from 2018–2021. His research focuses on automated vulnerability analysis, web security, binary analysis, mobile security, network security, underground economies, cybercrime, hacking competitions, and human factors of security.

Fish Wang Dr. Ruoyu "Fish" Wang is an Associate Professor in the School of Computing and Augmented Intelligence at Arizona State University. He is Associate Director of the Center for Cybersecurity and Trusted Foundations in the Global Security Initiative at Arizona State University and the co-Director of the Laboratory of Security Engineering For Future Computing (SEFCOM). He is a long-time Shellphish CTF player and a member of Nautilus Institute, the DEF CON CTF (Quals and Finals) organizer since 2022. His main research interest is binary analysis, including but not limited to, automated reverse engineering, vulnerability discovery, exploit generation, and decompilation.

Stephanie Forrest Stephanie Forrest is a Professor in the School of Computing and Augmented Intelligence at Arizona State University, where she directs the Biodesign Center for Biocomputation, Security and Society. Her interdisciplinary research focuses on the intersection of biology and computation, including cybersecurity, software engineering, and biological modeling.