Evolving Byte-Equivalent Decompilation from "Big Code"

Eric Schulte Jason Ruchti Matt Noonan David Ciarletta Alexey Loginov

***Abstract*—We introduce a novel technique for C decompilation that provides the correctness guarantees and readability properties essential for accurate and efficient malware analysis. Given a binary executable, an evolutionary search seeks a combination of source code excerpts from a "big code" database that can be recompiled to an executable that is *byte-equivalent* to the original binary. Byte-equivalence ensures that a successful decompilation fully reproduces the behavior, both intended and unintended, of the original binary. Moreover, the decompiled source is typically more readable than source obtained with existing decompilers, since it is generated from human-written source code excerpts. We present experimental results demonstrating the promise of this novel, general, and powerful approach to decompilation.**

I. Introduction

Recent widespread incidents have highlighted the significant threat malware poses to a secure internet. In a recent example, the WannaCry malware infected more than 230,000 computers in over 150 countries [33]. Reverse-engineering of WannaCry binaries was key to stopping the malware's spread [20]. In general, reverse-engineering and analysis of malware binaries is a critical first step in incident mitigation and recovery. Accurate decompilation allows the use of source-based program analysis tools, and enables security analysts to do their work more efficiently by reasoning at the source code level.

Traditional decompilation is fundamentally limited by a reliance on deterministic techniques that are often compiler-specific, optimization-specific, and Instruction Set Architecture (ISA)-specific; as a result decompilers are typically time-intensive to write and to evaluate for correctness. Compilers often generate obscure, idiosyncratic code that leverages deep semantic properties of the combination of source language and ISA. The reasoned reversal of such compiler idioms is more difficult than their initial implementation (and potentially undecidable), a problem exacerbated by the relatively small amount of development effort dedicated to decompilation technology compared to that applied to compiler development. To reverse this imbalance, we use evolutionary search to leverage compilers themselves for decompilation robust to compiler, ISA, and language developments.

Security analysis of binary applications requires trusted, accurate, and readable decompilation. Existing decompilers frequently produce decompilations that are not fully functionally equivalent, and fail to identify which portions of the decompiled source are accurate. Although recent academic work has begun to target correctness [30], [37], authors report that their decompilations often fail the weak semantic equivalence tests provided by the program's test suite.

We present the use of *byte-equivalence* of recompilation as a deterministically checkable guarantee of *full semantic reproduction* of the original binary, including both desired behavior as well as faults and vulnerabilities. Our technique performs an evolutionary search whose fitness function is based on byte-similarity with the original binary executable, with the success state being full byte-equivalence. This search leverages the compiler as a black box: existing compiler transformations do not need to be analyzed or reversed because they are applied directly when candidate decompilations are recompiled. This process iteratively improves a large population of candidate decompilations by applying source-to-source transformations that draw from a large database of human-written source code *excerpts*. The candidate population may be seeded with excerpts from this large external code database, as well as with the output of other decompilers. In the former case, those elements of the database whose compilation most resembles elements of the target are combined into new compiling candidate decompilations—in the limit, semantically equivalent source in the database quickly achieves *perfect* disassembly. In the latter case, our technique *improves* and *re-combines* existing decompiler output, driving them closer to byte-equivalence and continually incorporating new human-written code excerpts to improve fidelity and readability.

The resulting Byte-Equivalent Decompilation (BED) tool combines insights from a diverse set of recent advances in software engineering. Internet-scale "big code" databases are increasingly used to drive program analysis and synthesis [3], [24], [18]. Search based techniques are now an accepted tool for common software engineering and analysis tasks including program repair [16], [35], [17], program optimization [14], [28], and high performance fuzz testing [38], [39]. Compilers have been used for partial decompilation, to automatically generate assembly to compiler-IR translators [11]. Finally, essential to BED's success, a study of Gabel and Su [10] covering 430 million lines of source code revealed:

> a general lack of uniqueness in software at levels of granularity equivalent to approximately one to seven lines of source code [...] crossing both project and programming language boundaries.

We evaluate BED against a benchmark of small C programs exercising specific language constructs. We demonstrate the viability of the technique by achieving byte-equivalent decompilation of most programs. (The BED technique scales with function, not program, size so we expect our results to extend to larger programs.) Additional effort will be required for reliable

decompilation of real-world programs, specifically BED does not yet handle C structs.

We compare BED to the industry leading HEXRAYS decompiler [7] using a number of correctness and readability metrics. We find BED often outperform on such criteria, even when full byte-equivalence is not attained. When only partial byte-equivalence is attained, BED is able to identify lines of the decompiled source and regions of the original binary that do achieve full byte-equivalent recompilation. This ensures the utility of even partially byte-equivalent decompilation for security analysis by providing the guarantees of full byte-equivalent decompilation for large regions of the decompiled source.

Experimental Results. In an evaluation against a benchmark suite of small C source programs, the BED technique achieved byte-equivalent decompilations for 10 of the 19 programs. For those benchmark programs on which BED fails to achieve byte-equivalence, BED typically matches 81.23% of all machine-code instructions in the binary, but matches as much as 97.15% in optimized binaries when BED is seeded with output from the HEXRAYS decompiler.

We selected a suite of decompilation quality metrics (§IV-A4) with a focus on *readability* and *correctness*: properties essential to the use of decompilation to support program security analysis. BED outperforms the HEXRAYS decompiler on 10 of the 13 metrics (§IV-B3), notably producing significantly shorter decompiled C source.

Contributions. The main contributions of this paper follow.

- Introduction of BED's evolutionary decompilation technique, which leverages "big code" databases, the original compiler, and byte-equivalent recompilation as an objective function to achieve high quality decompilation. This novel technique is general across any source language, ISA, compiler, or set of compiler optimizations. All subsequent contributions enable this novel decompilation technique.
- Targeted search techniques to select, from a large source code database, excerpts which are likely to improve a currently non-byte-equivalent region of a candidate decompilation. This search is guided by similarity between the unmatched bytes from the original binary and the bytes associated with each source excerpt in the database.
- A byte-similarity based fitness function and selection technique capable of guiding evolution of the decompilation candidates toward byte-equivalence.
- Techniques of *targeted improvement* of evolved decompilation, including a quantifier free bit vector (QFBV)-based binary analysis technique capable of mining literals from stripped binaries.
- Methods of source code *recontextualization* to incorporate foreign code into candidate decompilations.
- Targeted mutations (see Table I) specifically targeting the acceleration of search for byte-equivalent decompilation.
- A technique of automatically responding to compiler warnings to fix compilation of evolved candidates.
- Source-to-source transformations capable of leveraging a large code database, together with a review of the resulting program search space and an empirical analysis of the robustness of source to source transformations.

II. EXAMPLE

The BED technique is illustrated in Figure 1. We walk through its decompilation for the small example shown in Figure 2a.

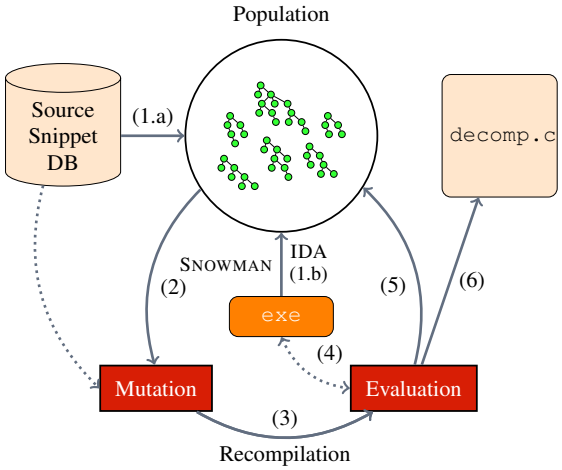

Fig. 1: BED system architecture.

The first step is to seed the candidate decompilation population. Candidates may be constructed from whole function excerpts from the source database that compile to bytes similar to those in the target binary (we call such candidates "*frankensteins*"), Figure 1 (1.a), or from the outputs of one or more existing decompilers (1.b). In the run producing this decompilation, the best candidate in the initial population consisted of a single function pulled from our experimental code database, which had approximately 55% byte-similarity with the target binary.

After a population of many such *frankenstein* candidate decompilations has been generated, BED evolves the population by iteratively applying both random and targeted source-to-source transformations that leverage the code database (2), recompiling the resulting candidates to produce new binaries (3), and then evaluating the fitness of the resulting binary using a byte-similarity fitness function (4). The evaluated candidates are returned to the population (5), and the technique proceeds using an evolutionary algorithm to guide the search towards increasing byte-similarity with the target in a process mimicking natural selection. When full byte-equivalence against the target is achieved the search terminates (6) returning the resulting byte-equivalent decompilation.

We present a number of techniques to improve the efficiency and accuracy of this search process. The application of source-to-source transformations is weighted to target those regions of candidate decompilations that currently fail to match the original. After each mutation, if the resulting individual fails to compile BED may make targeted changes to achieve

compilation. This is particularly useful when attempting to incorporate the output of existing decompilers, which often fails to compile. Once the individual has been compiled and the output compared against the target binary BED may also make targeted source transformations to match literals mined from the target binary. For example, the string literal on line 19 of Figure 2b and the `% 2` operation on line 11 of Figure 2b were both pulled directly from the target binary after binary-to-binary comparison.

At each generation a subset of individuals are selected for reproduction using lexicase selection (§ III-E); this maintains diversity by ensuring candidates recompiling to match regions of the original binary not matched by other candidates are not lost from the population.

```c
#include <stdio.h>
#define MAX 1000000
int main(void)
{
  int pprev = 0,
      prev = 1;
  int num = 0;
  int tot = 0;

  while (tot < MAX) {
    num = prev + pprev;
    prev = pprev;
    pprev = num;
    if (num % 2 == 0) {
      tot = tot + num;
    }
  }
  printf("%d\n", tot);

  return 0;
}
```

(a) Original source code.

```c
int main(void)
{
    int x = 0, y = 1;
    long int sum1 = 0, sum2 = 0;
    while (sum2 < 1000000) {
        sum1 = y + x;
        y = x;
        x = sum1;
        if (sum1 % 2 == 0) {
            sum2 = sum2 + sum1;
        }
    }
    printf("%d\n", sum2);
    return 0;
}
```

(b) BED byte-equivalent decompilation.

Fig. 2: Problem 2 from the Project Euler [1] set, shown in original source (Figure 2a), and BED evolved byte-equivalent decompilation (Figure 2b). HEXRAYS (circa 2015) decompiles this binary to over 250 lines of C. The compiler used automatically adds `#include <stdio.h>` when required.

The results of this technique are shown in Figure 2 along with the original source. The HEXRAYS decompilation of the same function demonstrates artifacts such as inclusion of C runtime functions and comments referencing stack offsets.

By contrast the BED decompilation in Figure 2b has similar size and form to the original, in Figure 2a, and could plausibly have been written by a human. Although the HEXRAYS decompilation of this function fails to compile without modification, the BED decompilation not only compiles, but matches the original program byte for byte.

III. OVERVIEW

The BED algorithm is shown in Figure 3. The computationally expensive tasks of evaluation, line 7, and reproduction, line 28, may both be parallelized across multiple threads.

The population may optionally be seeded, line 1, with the output of existing decompilers (§ IV-A5) via the *InitialPop*. Additional candidate decompilations may be synthesized from

Input: Target Binary, TBin : *Executable*
Input: Snippet Database, DB : *Database*
Parameters: *PopSize*, *CrossRate*, *MaxEvals*
Parameters: *InitialPop*, *NumFranks*, *FixLitChance*
Output: Decompilation of TBin

```
 1: let Pop ← InitialPop, EvalCounter ← 0
 2: do NumFranks times
 3:     let Pop ← Pop ∪ Frankenstein(DB, TBin)
 4: end do
 5: loop
 6:     let TmpPop ← ∅
 7:     for Candidate ∈ Pop do                          ▷ Evaluation
 8:         let CBin ← Compile(Candidate)
 9:         if CBin = Failed then
10:             Candidate ← FixCompilation(Candidate)
11:             CBin ← Compile(Candidate)
12:         end if
13:         let Diff ← null
14:         if CBin = Failed then
15:             Diff ← Disassembly(TBin)
16:         else
17:             Diff ← Evaluate(TBin, CBin)
18:         end if
19:         EvalCounter ← EvalCounter + 1
20:         if Fit = 0 then
21:             return Minimize(Candidate)
22:         else if EvalCounter ≥ MaxEvals then
23:             return Minimize(Best(Pop))
24:         end if
25:         TmpPop ← ⟨Candidate, Diff⟩
26:     end for
27:     Pop ← ∅
28:     while |Pop| < PopSize do                        ▷ Reproduction
29:         let p ← ⟨null, null, null⟩, p′ ← null
30:         if Random() < CrossRate then
31:             let p1 ← Select(TmpPop)
32:             let p2 ← Select(TmpPop)
33:             p ← Crossover(p1, p2)
34:         else
35:             p ← Select(TmpPop)
36:         end if
37:         let p′ ← Mutate(DB, p)
38:         Pop ← Pop ∪ {p′}
39:     end while
40:     GenCounter ← GenCounter + 1
41: end loop
```

Fig. 3: BED algorithm.

the code database, line 3, using the Frankenstein() method (§ III-A1). The algorithm's main loop, lines 5–41, iteratively evaluates, selects, and regenerates the population driving the candidate decompilations towards increasing byte-similarity with the target binary, *TBin*. During evaluation candidate decompilations are first compiled, line 8, and failed compilations are fixed when possible, line 10, (§ III-B2). Candidate compilations are then compared against *TBin*, line 17 using a disassembly-based fitness function (§ III-D) that returns the diff between *TBin* and *CBin*. This diff is later used to drive selection and to target mutation.

Selection is performed using lexicase selection, line 35, (§ III-E); crossover is applied with chance, probability

$CrossRate$, line 30. Mutation is then applied yielding a new candidate decompilation.

The generation counter is then incremented, line 40, and the algorithm repeats with Pop now holding the newly generated candidate decompilations. The algorithm runs until either byte-equivalence is achieved, line 21, or the runtime budget of $MaxEvals$ is exhausted, line 41. In either case the resulting candidate decompilation is minimized to remove any elements of the evolved source that do not contribute to the compilation (§ III-F). The remainder of this section discuses each of these stages in greater detail.

A. Initial Population

The initial population of candidate decompilations evolved by BED can be formed in many ways. We evaluate two options in this work, described in the following subsections.

1) Frankenstein: Frankenstein individuals are composed of whole functions from the code database. In a pre-processing step function signatures are extracted from the original binary using an off-the-shelf static analysis tool for analyzing stripped binaries. Function boundaries are computed using recursive descent, and the likely signature of each function is extracted to produce a template C file consisting of function prototypes with empty bodies. The bytes for each identified function are then extracted from the target binary and are used to perform a byte-similarity search in the code database as described in § III-C5. The result is a list of functions from the database sorted by similarity to the target bytes. Multiple frankenstein candidate decompilations are then generated by replacing each empty body in the template C code with weighted random selections from these sorted lists, giving a higher probability of selection to those excerpts which most closely match the target bytes. The probability of the ith most closely matching Frankenstein being selected can be calculated using the formula $(1-d)^{i-1} * d$, where d is the decay rate, a value between 0 and 1. In our experiments, we found using a decay rate of 0.125 yielded optimal results.

2) Other Decompilers: Any available decompiler may be used to seed the initial population of candidate decompilations, jump-starting BED's evolutionary search. Candidates from multiple decompilers may be used in combination with frankenstein candidates to seed a single population. The decompilation seeds may then be combined (via crossover) and improved upon via the evolutionary process. The use of lexicase selection helps retain candidates that match unique portions of the target binary (see §III-E). This, critically, ensures uniquely useful source code from any source is retained in the population until it can contribute (via crossover) to the final evolved decompilation. We evaluate the impact of seeding the population with decompilation generated via the HEXRAYS decompiler.

B. Local Search

To improve the speed with which the BED technique converges on byte-equivalent decompilations we introduce two forms of local search, literal mining § III-B1 and compilation repair § III-B2. Each of these techniques may be used to directly improve the fitness of randomly evolved candidate decompilations. Evolutionary search techniques augmented with local search are called *memetic algorithms* and have been proven effective in many cases [19], [8].

1) Fixing Source Literals: Because no code database can contain all possible literals to be utilized in a program, we have developed a mutation to identify literals in the target binary and insert them directly into the relevant portion of the candidate source. Many of the literal values immediately available in program binaries, e.g. addresses, are not suitable candidates for source-level transformation. Also, the compiler may optimize source level operations such as multiplication and division in such a way that a literal value cannot be gleaned directly from the disassembly listing.

To address the first challenge, we identify instructions that contain references to program values. When an operand references the read-only data section of the executable, we attempt to find the literal value at the referenced location. For instance, in "`lea eax,0x80484e0`", if "`0x80484e0`" is the starting address of the string literal "`Hello, world!`", we offer this as a candidate replacement in the source that compiles to the relevant portion of the candidate recompilation. Similarly when we encounter an instruction such as `movsd xmm0,0x8048638`, we offer the floating point literal at `0x8048638` as a replacement.

To address the challenge of compiler optimization of common arithmetic expressions, our approach utilizes a quantifier free bit vector (QFBV)-based analysis. As an example, consider the source fragment `n = 12*i`; this may be represented at the machine level as the following sequence of instructions.

```
mov eax,edx
add eax,eax
add eax,edx
shl eax,0x2
```

After identifying instruction sequences in the target binary that match the general pattern of optimized multiplication or division, we transform each instruction sequence into a QFBV formula describing the collective effect on the machine state. The resulting formula can be mined for constant values; in the example above, we would find that register `eax` was scaled by a factor of 12.

Our targeted fixing of source literals works by annotating the target decompilation with mined source literals. During fitness evaluation we identify the "diff regions" in which the compiled candidate differs from the target binary, and use debugging information from the compilation to map those back to the candidate source. Into these source regions we inject the recovered literal strings and numerical values, operators, local variables, and function names. This technique leverages alignment information resulting from the evolutionary search to improve the efficacy of simple traditional decompilation techniques.

2) Fixing non-compiling source code: Although we make an effort to perform mutations that generate syntactically valid individuals, it is possible that a newly-generated individual will fail to compile. In this case, we apply a sequence of

compilation-fixing transformations by matching the compiler's error messages to known strategies for fixing compilation. For example, a program that references a library function without including an appropriate header file can cause an error of the form "implicitly declaring library function F"; on seeing such an error, we automatically check if a `man` page exists for F and, if so, extract any `#include` directives from that page.

Another common compilation fixer is applied when an error of the form "use of undeclared identifier X" appears. In this case, we insert a random declaration for the given identifier on a line just before the error occurred, supplying a randomly-chosen type for the new declaration. [1]

Our suite of compilation-fixing strategies includes techniques designed to counter known problems in contributing decompilers. This significantly increases the contribution of decompiler-generated candidates to the BED evolutionary search.

If none of our compilation-fixing strategies are successful, we simply delete the line on which the error occurred. The compilation-fixing strategies are iterated until either an error-free program is obtained, or until a maximum number of attempts has been made.

C. Source Transformations

1) Recontextualization: We perform mutations directly on the source text of our individuals, maintaining a map from AST nodes to source ranges. This approach means that regions of source code may require additional processing when injected into a new source location or individual. For example, imagine that we were to implement an "insert" mutation, using the code excerpt `x = x + 1`. At the chosen insertion point, it is very unlikely that a variable named `x` is already in scope; a naïve insertion would result in an individual that would not compile. To increase the changes that mutation will result in compilable source, we *recontextualize* the code by identifying all free identifiers and rebinding them to names that are in scope at the insertion point. The rebinding operation is applied uniformly across the mutation text, ensuring that all instances of `x` are re-bound to the same identifier in the mutated code.

When the mutation text involves a group of statements that span more than one scope, extra caution is required to find a valid rebinding of identifiers which respects all scopes, as illustrated by the crossover example in Figure 4.

2) Mutations: When modifying a candidate decompilation, we randomly apply either a basic mutation or a targeted mutation from Table I. The basic mutations are standard "cut" (remove an expression), "insert" (insert text), "swap", and "replace." Each standard mutations may be limited in the regions of the source code to which it is applied in a number of ways. Each restriction yields a new mutation. The available restrictions are as follows; to full statements only, [2] to source and target pairs with matching AST classes, with source text drawn from the code database, with source text drawn from elsewhere in the program text. In all cases, the source text is recontextualized (see § III-C1).

[1] This change is similar to the `Add Init` Repair Type in SPR [17].

[2] A "full statement" is defined as any immediate child of a block; these roughly correspond to expressions in the C grammar that end with a semicolon or are wrapped in braces.

```
int fib(int n) {
  int x = 0;
  int y = 1;
  while (n > 0) {
    int t = x;
    x = x + y;
    y = t;
  }
  return x;
}
```

```
int collatz(int m) {
  while (m != 1) {
    if (m % 2 == 0)
      m /= 2;
    else
      m = 3*m + 1;
    ++k;
  }
  printf("%d\n", k);
  return k;
}
```

Fig. 4: Crossover example. If we perform crossover by trading lines 6–10 of `fib` with lines 3–11 of `collatz`, we must ensure that `k` is not re-bound to `t`; otherwise, the `printf` statement would reference an out-of-scope variable.

Mutation	Description
fix-literals	Identify literals in the target binary and replace incorrect source literals.
promote-guarded	Promote statements from within compound statements, (i.e. `{...}`).
explode-for-loop	Decompose a `for` loop into a `while` loop.
coalesce-while-loop	Coalesce a `for` loop and preceding assignment into a `while loop`.
arith-assign-expansion	Expand arithmetic assignments (e.g. +=) into traditional assignments.
insert-excerpt-decl	Search the database specifically for a declaration to be inserted into the program.
rename-variable	Select a variable and replace with another in-scope variable.

TABLE I: Targeted mutations, designed to accelerate evolution and address problems identified empirically during evolutionary decompilation runs.

3) Crossover: All crossover operations are *homologous* meaning that the parents are aligned before crossover points are selected. We align parents by compiling each and matching their compiled machine code. A crossover point in one parent is mapped through the aligned machine code to a corresponding point in the other parent's source. Homologous crossover has been previously shown to encourage meaningful and productive crossover operations [21]. We perform two-point crossover, selecting two aligned points. The region between these points may open or close n scopes. In the second parent, we find a homologous region with the same value of n and swap these two regions as shown in Figure 4.

4) Diff-targeted mutation: We focus mutations towards the "broken" parts of the candidate decompilation. Specifically, we identify *bad statements* in a program as those source statements which compile to bytes belonging to some *diff-region*, as described in § III-D. We limit mutation targets to these bad statements with probability $TargetChance$ (Table III) otherwise we choose mutation targets uniformly at random from all statements.

5) Similarity search: To improve the quality of excerpts used in insertion and replacement mutations, we search our

excerpt database for the excerpts whose compiled form is most byte-similar to the region of the target binary corresponding to the point of mutation. This optimization leverages the *diff-region* identified by the fitness function (see § III-D). We search the database for the excerpts most similar to the target binary's disassembly in the *diff-region* using edit distance as our similarity metric. We then perform a weighted random selection from the sorted results, ensuring that those results from the fodder database which most closely match the *diff-region* are selected with a higher probability. [3]

D. Disassembly Fitness

To evaluate a candidate decompilation of the target binary, we use the edit distance between the target and candidate decompilations' disassembly listings as our fitness metric.[4] We assign a perfect fitness to each instruction in the target binary within a *common-region* (i.e., non *diff-region*) and a bad fitness value proportional to the size of the diff to each instruction in the target binary within a *diff-region*.

We normalize the disassembly listings before computing the diff. We remove irrelevant nop and other padding instructions meant to align code to even-address boundaries. We resolve addresses as described in § III-B1. Finally, we identify instructions that change the program counter, such as `jmp 0x8014040` or `call 0x8014080` and replace the address operand with the function name or ELF section name and offset to ensure differences in program alignment do not negatively impact fitness.

E. Lexicase Selection

Lexicase selection is a genetic programming technique shown to dramatically increase population diversity and long term evolutionary success in some cases [12]. With lexicase selection, the fitness of an individual is represented as a vector of independent test cases (instead of the traditional representation as a scalar sum of passed test cases). Selection is performed by iteratively filtering the population by best performance on a random ordering of the tests in this vector until a single candidate remains. By splitting the fitness calculation across multiple test cases, individuals that pass tests not widely passed by the remainder of the population are retained in the population with high probability, even when they fail many other tests. With a traditional fitness function such individuals are often lost from the population due to a low total number of test cases passed.

Each machine-code instruction in the target binary is a distinct fitness test case. With this approach, a candidate decompilation that matches a portion of the target binary unmatched by other members of the population has a high probability of being retained in the population. The source code from this candidate responsible for the unique binary matches then has many opportunities to be incorporated into other candidate decompilations via crossover. Our experimental results demonstrate the utility of lexicase selection (§ IV-B5).

[3]Thanks to *Anonymized*for search database implementation.

[4]Computed using a sequence comparison algorithm in Wu *et al.* [36].

F. Minimization

There are no protections against the evolutionary search accumulating unnecessary source code artifacts. In fact, through a phenomenon known as *bloat* [23], it is often beneficial to a candidate's long-term survival to accumulate unused, or *dead*, genetic material. To correct for bloat, and to ensure irrelevant code is not returned by the algorithm, we minimize the returned source code using *delta debugging* [40]. The delta debugging algorithm systematically removes both lines and ASTs from the program until a minimal set that compiles to the same binary as the original is found.

Before minimization we pass the source code through the `clang-format` utility [13]. Clang-format enforces uniform indentation and coding conventions improving readability, and breaks semantically distinct program elements into separate lines for easier removal via delta debugging.

IV. Experimental Results

Our experiments address several research questions.

- RQ1. **Evolution of byte-equivalence from big code.** We demonstrate the feasibility of evolving byte-equivalent code by evaluating the degree to which BED is able to achieve byte-equivalence with and without decompiler seeds against a benchmark set compiled with and without compiler optimizations (§ IV-B1).
- RQ2. **Impact of targeted mutations.** We demonstrate the utility of our targeted mutation operators for the evolution of byte-equivalence (§ IV-B2).
- RQ3. **Utility of evolved decompilation.** We evaluate the utility of evolved decompilation to support the analysis and rewriting of binary executables. This evaluation is performed through analysis of a number of *readability* and *correctness* metrics of both the *decompiled source* and the *recompiled binary* (§ IV-A4).
- RQ4. **Handling of compiler optimization.** We evaluate BED across multiple levels of compiler optimization.
- RQ5. **Impact of lexicase selection.** We evaluate the impact of lexicase selection versus standard tournament selection.

A. Methodology

Experimental evaluations were performed against the benchmark described in §IV-A1 using the BED parameters described in § IV-A2, and the Euler database described in § IV-A3.

1) Benchmarks: The benchmark programs in Table II were selected to demonstrate a wide range of features of the C language. Specifically, the Learn C The Hard Way (LCTHW) programs [31] are taken from an example-based C language tutorial. Each selection demonstrates a specific facet of the language. The Project Euler programs [1] are taken from a popular set of programming challenges and are used to investigate the effect of a highly relevant database (see §IV-A3). As "big code" projects such as DARPA's MUSE [5] continue to mature we anticipate the availability of increasingly relevant sample code.

[5]http://corpus.museprogram.org/

Program	LOC	Description
LCTHW [31]		
hw1	5	Hello world
hw3	9	Formatted printing
hw6	19	Types of variables
hw7	23	More variables, some math
hw8	32	Sizes and arrays
hw9	40	Arrays and strings
hw10	20	Arrays of strings, looping
hw11	22	While loop & boolean exprs.
hw12	17	If else if else
hw13	45	Switch statement
hw14	32	Writing and using functions
hw15	43	Pointers dreaded pointers
hw18	88	Pointers to functions
Euler [1]		
Euler-1	12	Multiples of 3 and 5
Euler-2	19	Even Fibonacci numbers
Euler-3	48	Largest prime factor
Euler-4	40	Largest palindrome product
Euler-5	15	Smallest multiple
Euler-6	48	Sum square difference

TABLE II: Benchmark programs. Learn C The Hard Way (LCTHW) is a C example-based tutorial that provides simple programs illustrating features of the C language [31].The Project Euler problems are a selection of answers to the Project Euler programming challenge [1].

2) Parameters: The following parameters shown in Table III were used to configure BED in our experimental evaluations. Runs were performed at both O0 and O2 optimization levels.

Parameter	Value
compiler	clang
flags	`-m32 -g -OX`
$CrossRate$	$\frac{1}{4}$
$MaxEvals$	131072
$PopSize$	1024
$TargetChance$	$\frac{3}{4}$

TABLE III: BED experimental parameters.

3) Code Database: The Project Euler [1] database consists of 688 posted solutions to Project Euler problems. [6] We were able to identify and extract over 80k unique source ASTs from this corpus.

4) Metrics: To evaluate BED's performance, we consider two evaluation criteria: readability and byte similarity to the target binary. To compute the byte similarity, we take the number of matched instructions over the total number of instructions within the original and evolved binary. In addition to measuring byte similarity to the target binary, we apply the readability metrics listed in Table V to the decompiled source code. The first eleven readability metrics all count source features that inhibit readability, such as the number of `goto` statements, and thus a smaller number is preferable. The last readability metric measures direct matches against the original source, and thus a larger number is preferable.

5) Decompilers: We perform an experimental comparison against the HEXRAYS decompiler circa 2015. An attempted comparison against recently published decompilers was not possible. Neither Dream [37] nor Phoenix [30] could provide either the running decompiler nor the decompiled C from published experiments. [7]

B. Results

1) RQ1: Evolution of Byte-equivalence: By combining solutions found both with and without compiler optimizations enabled, BED is able to achieve byte-equivalence for 4 of 19 example programs when run without decompiler-generated seeds and 10 of 19 when the HEXRAYS decompiler is used to seed candidate populations. Detailed results are shown in Table IV. In those cases where byte-equivalent decompilation is not achieved the resulting decompilation is often *close to* the original program, often achieving byte-equivalence for whole functions if not the whole program.

In unsuccessful runs the BED technique is able to highlight those portions of the source which are compiling to non-byte-equivalent portions of the resulting binary. This mapping is using debugging information added during re-compilation to map failing regions in the binary to failing regions of source. In optimized binaries this mapping is not complete. In practice non-byte-equivalent portions are small, as we achieve 81.23% byte-similarity with the target binary in our experiments. In terms of LoC, for non-optimized binaries, BED fails to achieve byte-equivalent recompilation for 11.79% of all lines without decompiler seeds and 8.59% of all lines when decompiler seeds are used. Thus even runs that fail to achieve full byte equivalence still provide useful source for reverse engineering and analysis for the majority of the program.

This high level of byte-similarity indicates the promise of the BED technique. With structure mutation and a larger code database, BED could plausibly achieve byte-equivalent decompilation of most arbitrary C programs. Such a result would revolutionize the analysis and rewriting of non-obfuscated binary programs. [8]

2) RQ2: Impact of targeted mutations: The success rates for various mutation types when acting on a population in an evolutionary run appear in Figure 5.

For each mutation, we assess the fitness of an individual before and after the mutation, recording whether the mutation caused fitness to improve, remain the same, or become worse. If the mutated individual could not compile, we applied the compilation-fixing strategies (§III-B2); if the fixed source still did not compile, the individual was classified as "dead."

All mutations showed a 1–10% probability of resulting in a fitness improvement, but showed radically different chances

[6]The Project Euler corpus was first collected and organized by Seth Pollen and Ben Welton under the supervision of Prof. Ben Liblit.

[7]We contacted the authors of both tools seeking either executable decompilers or static decompilations from prior published results.

[8]The BED technique may also be applicable to obfuscated programs when the obfuscation tool is available. In this case the obfuscation tool itself simply replaces the black-box compiler.

Program	BED			BED Optimized			BED W/Deco.			BED Optimized W/Deco.		
LCTHW [31]	Orig	Final	Evals	Orig	Final	Evals	Orig	Final	Evals	Orig	Final	Evals
hw1	70.13	100.00	5120	76.92	100.00	3072	100.00	100.00	0	100.00	100.00	0
hw3	66.22	100.00	7168	77.11	100.00	10,229	78.21	100.00	4225	100.00	100.00	0
hw6	40.34	82.67	131,072	27.54	84.39	131,072	51.73	86.68	131,072	86.55	95.36	131,072
hw7	32.44	87.33	131,072	41.86	100.00	68,608	56.11	93.04	131,072	82.71	100.00	14,465
hw8	33.52	78.64	131,072	34.86	91.95	131,072	70.25	88.13	131,072	74.40	99.26	131,072
hw9	17.68	80.77	131,072	35.69	77.64	131,072	63.19	86.38	131,072	74.07	100.00	33,921
hw10	47.98	84.64	131,072	46.07	99.15	131,072	66.67	94.25	131,072	94.06	98.31	131,072
hw11	51.92	83.85	131,072	43.44	95.44	131,072	64.52	94.15	131,072	94.01	97.44	131,072
hw12	48.50	90.79	131,072	43.65	89.13	131,072	71.57	98.73	131,072	95.33	100.00	7297
hw13	37.87	71.71	131,072	29.52	48.10	131,072	47.37	69.18	131,072	67.87	99.18	131,072
hw14	55.24	79.26	131,072	46.29	50.21	131,072	72.92	86.23	131,072	59.90	97.33	131,072
hw15	31.99	60.40	131,072	27.61	61.56	131,072		N/A			N/A	
hw18	53.68	66.53	131,072	39.26	38.98	131,072		N/A			N/A	
Euler [1]												
Euler-1	59.84	92.31	131,072	36.43	99.10	131,072	74.31	95.31	131,072	100.00	100.00	0
Euler-2	59.29	100.00	72,703	62.11	81.53	131,072	77.12	100.00	6273	81.25	100.00	0
Euler-3	46.42	82.94	131,072	36.11	46.80	131,072	52.61	89.03	131,072	49.68	64.68	131,072
Euler-4	41.24	69.01	131,072	36.21	49.44	131,072	75.56	84.55	131,072	100.00	100.00	0.00
Euler-5	52.00	98.10	131,072	30.64	95.67	131,072	68.25	98.10	131,072	100.00	100.00	0.00
Euler-6	58.63	97.89	131,072	36.90	70.79	131,072	77.71	100.00	101,505	100.00	100.00	0.00
total		84.57			77.89			91.99			97.15	

TABLE IV: Evolution of byte-equivalence. The columns list the performance of BED in achieving byte-equivalence. The entries are formatted as *Orig Final*, *Evals* where *Orig* is the byte similarity to the target binary of the best candidate in the original population, *Final* is the greatest byte similarity achieved over the course of the run, and *Evals* is the total number of fitness evaluations performed during the run. For the first two results columns, the initial population did not include decompiler seeds, while in the last two columns, the initial population included seeds from the HEXRAYS decompiler. In the first and third columns, compiler optimizations were not enabled (-O0), while in the second and fourth columns, compilation was performed with optimizations enabled (-O2). Where the HEXRAYS decompiler did not generate compiling seeds for the evolutionary process, N/A is shown.

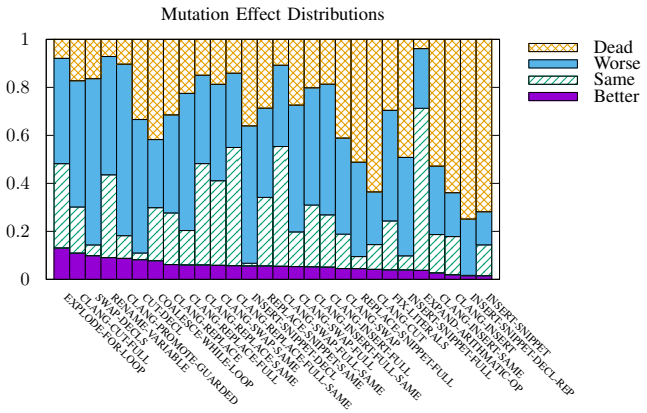

Fig. 5: Mutation success rates by type

of producing neutral candidate decompilationss. Generally, mutations that preserve the AST class and only operate on full statements performed the best, creating neutral candidate decompilationss—decompilations with the same fitness as before mutation—35–55% of the time, compared to a 15–20% rate of neutral candidate decompilations generation for the unrestricted versions of those mutations. The generation of candidate decompilations which are as fit as but not identical to their ancestors advances evolutionary search by increasing both population diversity and the explored fraction of the fitness landscape. Neutral offspring are beneficial to evolutionary search [34].

3) RQ3: Utility of evolved decompilation: Our readability and byte-equivalence metrics, described in §IV-A4, are shown in Table V. In comparison to the HEXRAYS decompiler, the decompilations generated by BED included fewer `gotos`, casts, variable declarations, live ranges, macros, typedefs, and significantly shorter total decompiled code—all key metrics for accessing source code readability. Additionally, BED decompilations achieved a higher degree of byte-similarity with the target binary and BED always generated compiling decompilations.

4) RQ4: Handling of compiler optimization: At higher levels of optimization, the mapping between source ASTs and machine code is less direct as source is elided or transformed significantly. As a result, the targeting of mutations to diff-inducing source ASTs becomes a significant challenge.

Despite this, BED performs well at higher levels of optimization, and indeed performs better with optimizations enabled for some benchmark programs. At higher levels of optimization, the space of valid C source code that compiles to the same bytes is significantly larger than without optimizations enabled. For instance, consider the example HW10 decompilation given in 6. In the resultant decompilation, the loop in the original

Metric	BED	HexRays
Byte similarity	81.23	69.16
# gotos	0	10
# casts	18	118
# variables	219	262
# scopes	216	211
# live ranges	185	262
# dead assignments	34	0
# macros	8	3116
# typedefs	0	608
# characters	22,346	263,835
# lines	1040	8342
# ASTs †	4723	5616
# matched literals	287	398

† AST count does not include macros and types

TABLE V: Byte-similarity and readability metrics.

program has been elided and replaced with a sequence of printf statements which compiles, with optimization, to the same bytes as the original; this would not be the case with optimizations disabled.

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i = 0;
  // go through each string in argv
  // why am I skipping argv[0]?
  for(i = 1; i < argc; i++) {
    printf(``arg %d: %s\n'', i, argv[i]);
  }
  // let's make our own array of strings
  char *states[] = {
    ``California'', ``Oregon'',
    ``Washington'', ``Texas''
  };
  int num_states = 4;
  for(i = 0; i < num_states; i++) {
    printf(``state %d: %s\n'', i, states[i]);
  }
  return 0;
}
```

(a) Original source code.

```c
int main(int argc, char** argv) {
  for (int d = 1; argc * 1 > d; d++) {
    printf(``arg %d: %s\n'', d, *argv);
  };
  int d;
  d = 0;
  printf(``state %d: %s\n'', d, ``California'');
  d = 2;
  printf(``state %d: %s\n'', 1, ``Oregon'');
  printf(``state %d: %s\n'', d, ``Washington'');
  printf(``state %d: %s\n'', 3, ``Texas'');
}
```

(b) BED decompilation.

Fig. 6: The tenth problem from the Learn C The Hard Way [31] programming tutorial shown in original source (Figure 6a) and BED evolved near byte-equivalent decompilation (Figure 6b) with compiler optimizations enabled.

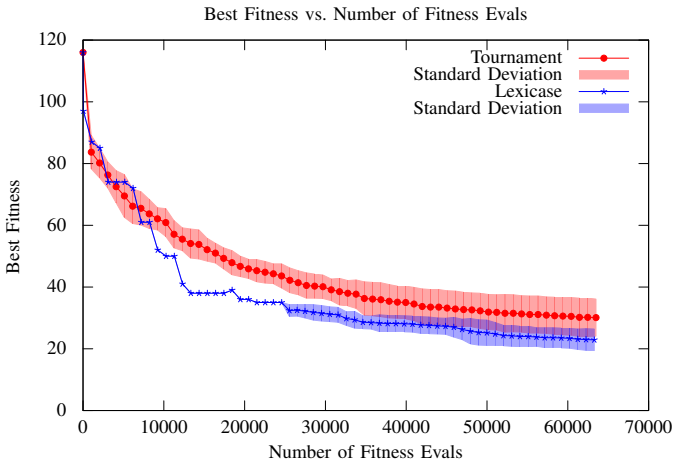

Fig. 7: Comparison of the best fitness in the population as a function of number of fitness evaluations performed using lexicase and tournament selection. Numbers shown are averages of 15 decompilations of the Euler-2 benchmark program.

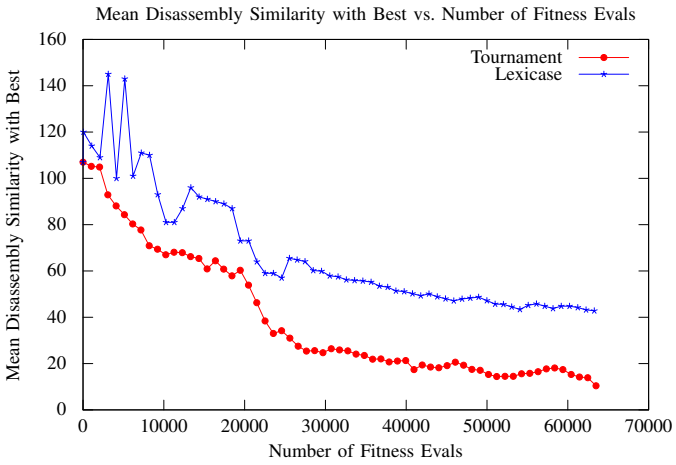

Fig. 8: Comparison of lexicase and tournament selection on diversity of Euler-2 decompilations. 15 decompilations were performed for each style. A value of zero implies no diversity.

5) RQ5: Impact of lexicase Selection: To test the impact of lexicase selection on overall fitness and population diversity, we compared performance versus standard tournament selection. In tournament selection, overall fitness is given as a single scalar value, and the best individual from a population sample is chosen to reproduce.

Our experiments with lexicase selection demonstrate a significant improvement over tournament selection. As shown in Figure 7, lexicase selection achieves higher levels of fitness in the best individual in the population. Additionally, by measuring diversity as the average edit distance between the best individual and a randomly selected individual's disassembly listings, we see in Figure 8 that lexicase selection enables a greater diversity of individuals to remain present.

V. Limitations

Unknown compiler and flags. We assume the compiler and flags used to compile the original binary are known. This is typically not the case, however recent work has demonstrated the feasibility of automatically learning this information [26]. Alternately the compiler and flags could be added to each candidate decompilation and evolved as part of the BED search.

Available code database. BED is limited by the quality of the available database of source code. We anticipate the emergence of large, high quality, easily searched databases of collected source code. Such databases will be critical for the extension of this technique to real-world programs.

Runtime and large search spaces. The amount of time taken by BED can vary dramatically. Heuristic search provides no guaranteed limits of runtime. The efficiency of evolutionary search techniques is typically a result of the size of the search space and the quality of the fitness function used. Although we present many specific techniques to improve efficiency, the size of the search space is daunting and the results presented required many thousands of fitness evaluations.

C structure declarations. The benchmark programs, mutation strategies, and decompilation results presented herein do not include consideration of C structures. We believe generalizing our technique to support mutations of structure declarations is a matter of implementation and does not fundamentally affect the implementation or the performance of the BED technique. However, we note that handling of structures will allow evaluation of BED on larger programs.

Partial byte-equivalence. This work demonstrates the promise of evolutionary search as a decompilation technique. We achieved 81.23% byte similarity over our experimental benchmark set. With the improvements suggested in § VII, the increased engineering efforts suggested above, and larger or more focused code databases, we believe a significantly higher fraction of programs could reach full byte-equivalence. Even partial byte-equivalent decompilation is useful for security analysts, especially as BED marks the specific lines of source and regions of the binary which fail to achieve byte-equivalence so only those specific regions require binary analysis.

VI. Related Work

Phoenix. Schwartz *et al.* [30] note that previous work rarely evaluates *correctness*, meaning the preservation of behavior of the original binary. They note that decompilation for security requires both correctness and high-level abstractions. They address these needs by using a *structural analysis* algorithm that includes *semantics preserving structural analysis* to remove non-semantics-preserving schemata and *iterative control-flow structuring* to selectively remove edges from the control flow graph in support of subsequent schema application.

Correctness of Phoenix's output was assessed by comparing the recompiled binary's behavior to the original program on a suite of tests. This approach to correctness depends on having a test suite with high coverage, which may not be possible for legacy or third-party binaries.

Dream. Yakdan *et al.* [37] introduce decompilation using a *pattern-independent* control-flow structuring algorithm capable of recovering control constructs from the original source. Their goals are similar to those of this work in that they target decompilation to support security analysis by humans and using source code level tools; both of which require *structured* code without a surfeit of GOTOs. Against the GNU coreutils programs they boast smaller code and fewer `gotos` than Phoenix, however they don't address *correctness*.

To decompile abnormal control-flow structures such as multi-entry loops, Dream is able to introduce fresh variables to act as tags, controlling the execution of particular portions of the loop body. Although this transformation successfully restructures many abnormal loops, it introduces further complexity to arguments that the decompiled source is semantically equivalent to the original code as the introduced variables may affect the observable program behavior.

SmartDec. SmartDec [9] targets decompilation to C++ code. In addition to standard type inference and dataflow analyses, it includes specialized algorithms designed to reconstruct specific C++ idioms, including virtual function tables, class hierarchies, exceptions, and constructors and destructors.

SmartDec is open source and has been forked by the SnowMan project; SnowMan augments SmartDec's decompilation with improved parsing of ELF and PE files, better calling-convention detection, and a larger set of patterns for reconstructing expressions.

JSNice. Raychev *et al.* [25] introduce a novel approach for predicting identifier names and type annotations in JavaScript code. Using an extensive corpus of existing JavaScript, they build a probabilistic graphical model of program properties using conditional random fields. The resulting tool, JSNice, is able to predict correct names for 63% of identifers in obfuscated JavaScript programs and to correct type annotations in 81% of cases.

Mutational Robustness. Software functionality has been found to be surprisingly robust to random perturbations [29]. This robustness and the resultant *neutral spaces* of multiple diverse implementations of a single specification are thought to be central to software's amenability to genetic improvement [27]. Our investigation of the effectiveness of mutations in § IV-B2, perhaps unsurprisingly, identifies similar robustness of byte-similarity of compilation to source modification. Similarly our breakdown of binary impact by type of modification extends and agrees with similar work analyzing the functional impact of source-to-source transformations [2], [17].

VII. Future Work

Decompilation (lifting) to LLVM-IR. The low level virtual machine (LLVM) [15] intermediate representation has become a popular target for program analysis and rewriting. The BED technique is not inherently specific to C, and may be directly re-targeted to the lifting of binaries to any program representation

for which a compiler exists, including LLVM-IR. We believe the BED technique will more effectively evolve byte-equivalent LLVM-IR than it does byte-equivalent C. This is because of the relative simplicity of LLVM-IR as compared to the C language, leading to a greatly reduced search space. This effort would entail the following straightforward steps.

1) Re-compilation of our code database extracting LLVM excerpts associated with each source and machine-code pair.
2) Implementation of an evolve-able representation for LLVM software objects analogous to our representation of C software objects.

Given the popularity of LLVM as a representation supporting program analysis and rewriting, multiple tools exist for lifting to LLVM [32], [6]. These tools typically do not achieve byte equivalent recompilation and exhibit cumbersome recompilation artifacts, including explicit representations of activation records on the stack. The BED technique has the opportunity to significantly improve the state of the art for LLVM lifting enabling the automated analysis and modification and recompilation of COTS binaries.

Multi-objective fitness functions. Multi-objective fitness function can explicitly target additional desirable properties aside from byte similarity. These might include the following.

Readability Metrics. In fact any of the readability metrics from §IV-A4 could explicitly be added to the BED fitness function.

Test Suites. Although full byte-equivalence entails functional equivalence, it is possible that use of existing program test suites, when present, to guide the search could improve the overall efficiency. Functional correctness may serve as a useful proxy for byte equivalence, improving the fitness distance correlation of BED's fitness and the efficiency of BED's search for full byte equivalence.

Machine learning for local decompilation. The large code corpora used by BED could also be used to train local machine-learning techniques to automatically *learn* a decompilation function mapping short sequences of compiled machine code to C source code decompilation. For example, recurrent neural network (RNN) encoder-decoder models have had great success in translating between natural languages [5]. Adapting such models to translate small program segments from machine code to C could subsequently be used to improve the evolutionary decompilation process by directly translating the diff inducing regions of the target binary to C source.

Structured code search and completion. Existing systems for structured source code search [22] and source code completion [4] could be applied to candidate decompilations. Such tools are intended to aid developers in the authoring of code by searching a large code database for excerpts which are similar or likely to complement existing code. BED could leverage such techniques by using candidate decompilations to drive code retrieval and suggestion systems to generate new code or modifications to be applied back to the candidate decompilation. Such techniques could accelerate the decompilation process and result in increasingly natural decompiled source code.

VIII. Conclusion

We present a novel and general Byte-Equivalent Decompilation (BED) technique that works via the evolutionary recombination and recompilation of source excerpts from a "big code" database. We present experimental results demonstrating BED's ability to evolve readable C source code that compiles to an exact byte-for-byte match against target binaries. Byte-equivalent recompilation of decompiled source code is a deterministically checkable guarantee of *full semantic reproduction* of the original binary, including both desired behavior as well as faults and vulnerabilities. The genesis of the decompiled code in human-written excerpts leads to human-readable decompiled source.

We present a number of techniques that make byte-equivalent decompilation possible, including the use of existing decompilers to seed the search, an effective fitness function to guide the search, the use of byte-similar retrieval from a code database, the use of binary-difference-based mutation targeting, processes for fixing failing compilations and for pulling literal values from the original binary, and the application of lexicase selection. BED is a novel approach to decompilation that promises to achieve readability and full semantic equivalence in a manner which is "future proof" against new source languages, ISAs, compilers, and compiler optimizations.

IX. Acknowledgments

(*Anonymized*)

References

[1] Project Euler. http://projecteuler.net/.

[2] Benoit Baudry, Simon Allier, Marcelino Rodriguez-Cancio, and Martin Monperrus. Automatic software diversity in the light of test suites. in-submission-2015.

[3] Pavol Bielik, Veselin Raychev, and Martin Vechev. Programming with" big code": Lessons, techniques and applications. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[4] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

[5] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[6] ARTEM DINABURG and ANDREW RUEF. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[7] Chris Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[8] Emad Elbeltagi, Tarek Hegazy, and Donald Grierson. Comparison among five evolutionary-based optimization algorithms. *Advanced engineering informatics*, 19(1):43–53, 2005.

[9] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. SmartDec: Approaching C++ decompilation. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 347–356. IEEE, 2011.

[10] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Foundations of Software engineering*, pages 147–156. ACM, 2010.

[11] Niranjan Hasabnis and R Sekar. Automatic generation of assembly to ir translators using compilers.

[12] Thomas Helmuth, Lee Spector, and James Matheson. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 2014.

[13] D. Jasper. clang-format: Automatic formatting for C++, 2013. http://llvm.org/devmtg/2013-04/jasper-slides.pdf.

[14] William B. Langdon and Mark Harman. Evolving a CUDA kernel from an nVidia template. In *Congress on Evolutionary Computation*, pages 1–8, 2010.

[15] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[17] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[18] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312. ACM, 2016.

[19] Peter Merz and Bernd Freisleben. A genetic local search approach to the quadratic assignment problem. In *Proceedings of the 7th international conference on genetic algorithms*, pages 1–1, 1997.

[20] Lily Hay Newman. How an accidental 'kill switch' slowed friday's massive ransomware attack. https://www.wired.com/2017/05/accidental-kill-switch-slowed-fridays-massive-ransomware-attack/, May 2017.

[21] Peter Nordin, Wolfgang Banzhaf, and Frank D Francone. 12 efficient evolution of machine code for cisc architectures using instruction blocks and homologous crossover. *Advances in genetic programming*, 3:275, 1999.

[22] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *Software Engineering, IEEE Transactions on*, 20(6):463–475, 1994.

[23] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by John. R. Koza).

[24] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, New York, NY, USA, 2015. ACM.

[25] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 111–124. ACM, 2015.

[26] N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (FSE 2010)*, pages 21–28. ACM, 2010.

[27] Eric Schulte. *Neutral Networks of Real-World Programs and their Application to Automated Software Evolution*. PhD thesis, University of New Mexico, Albuquerque, USA, July 2014. https://cs.unm.edu/~eschulte/dissertation.

[28] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 639–652. ACM, 2014.

[29] Eric Schulte, Zachary Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, pages 1–32, 2013.

[30] Edward J Schwartz, J Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium*, page 16, 2013.

[31] Zed A. Shaw. Learn C the hard way. http://c.learncodethehardway.org/book/.

[32] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 52–61. IEEE, 2013.

[33] Gayle Swenson. Bolstering government cybersecurity lessons learned from wannacry. https://www.nist.gov/speech-testimony/bolstering-government-cybersecurity-lessons-learned-wannacry, 2017.

[34] Andreas Wagner. *Robustness and Evolvability in Living Systems*. Princeton University Press, 2013.

[35] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366. IEEE, 2013.

[36] S. Wu, U. Manber, G. Myers, and W. Miller. An o(np) sequence comparison algorithm. *Inf. Process. Lett.*, 35(6):317–323, September 1990.

[37] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. 2015.

[38] Michal Zalewski. american fuzzy lop, 2015. http://lcamtuf.coredump.cx/afl/.

[39] Michal Zalewski. american fuzzy lop technical "whitepaper". Technical report, 2015.

[40] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Foundations of Software Engineering*, pages 253–267, 1999.