Evolving Exact Decompilation

Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, Alexey Loginov
GrammaTech
531 Esty St.
Ithaca, NY 14850
{eschulte,jruchti,mnoonan,dciarletta,alexy}@grammatech.com

Abstract—We introduce a novel technique for C decompilation that provides the correctness guarantees and readability proper-
ties essential for accurate and efficient binary 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 [34]. Reverse-engineering of WannaCry binaries was key to stopping the malware’s spread [22]. 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 source language and the target 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 compiler development. To reverse this imbalance, we use evolutionary search that leverages the compilers themselves, enabling decompilation across compiler, ISA, and source language.

Security analysis of binary applications requires trusted, accurate, and readable decompilation. Existing decompilers frequently produce decompilations that fail to achieve full functional equivalence with the original program, and cannot identify which portions of the decompiled source are accurate. Although recent academic work has begun to target correctness [31], [38], authors report that their decompilations often fail the weak semantic-equivalence tests provided by the program’s test suite.

We propose 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, using a fitness function that measures byte-similarity with the original binary executable. 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 initialized using randomly generated code, excerpts from this large external code database, or even using the output of other decompilers. Our technique then improves and re-combines the initial candidates, driving them closer to byte-equivalence while 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 [5], [26], [20]. Search based techniques are now an accepted tool for common software engineering and analysis tasks including program repair [18], [36], [19], program optimization [16], [29], and high performance fuzz testing [39], [40]. Compilers have been used for partial decompilation, to automatically generate assembly to compiler-IR translators [12]. Finally, essential to BED’s success, a study of Gabel and Su [11] covering 430 million lines of source code revealed:

This material is based upon work supported by the SPAWAR Systems Center Pacific under Contract No. N66001-13-C-4046.

Workshop on Binary Analysis Research (BAR) 2018
18 February 2018, San Diego, CA, USA
ISBN 1-891562-50-9
http://dx.doi.org/10.14722/bar.2018.23008
www.ndss-symposium.org
#include <stdio.h>
#define MAX 1000000
int main(void)
{
    int pprev = 0, prev = 1;
    int num = 0, tot = 0;
    while (tot < MAX) {
        num = prev + pprev;
        pprev = prev;
        prev = num;
        if (num % 2 == 0) {
            tot = tot + num;
        }
        printf("%d
", tot);
    }
    return 0;
}

(a) Original source code.

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
", sum2);
        return 0;
    }

(b) Byte-equivalent BED output.

Fig. 1. Problem 2 from the Project Euler [2] set, shown in original source (Figure 1a), and BED evolved byte-equivalent decompilation (Figure 1b). The C compiler automatically adds #include <stdio.h> when required.

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 benchmark programs. The BED technique can be parallelized across functions, suggesting that our results can extend to larger programs.

We compare BED to the industry leading HEX-RAYS Decompiler [9] using a number of correctness and readability metrics. We find BED often outperforms HEX-RAYS on such criteria, even when full byte-equivalence is not attained. Even 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 (Figure 3). This ensures the utility of BED’s output for security analysis by providing the guarantees of full byte-equivalence for large—and clearly delineated—regions of the decompiled source.

Experimental Results. In an evaluation against a benchmark suite of small C source programs, the BED technique achieved full byte-equivalent decompilations for 10 of the 19 programs. For those optimized benchmark programs on which BED fails to achieve byte-equivalence, BED typically matches >80% of all machine-code instructions in the binary, increasing to 97% when seeded with output from the HEX-RAYS 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. We use byte-equivalence of the resulting binary as a computable, strong proxy for correctness. BED outperforms the HEX-RAYS Decompiler on 10 of the 13 metrics (§IV-B2), notably producing significantly shorter decompiled C source.

Contributions. We describe BED’s novel 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 approach generalizes across source languages, ISAs, compilers, and optimizations.

Evolutionary search is accelerated with techniques that mine useful excerpts from a large source code database. This excerpt search is guided by similarity between the unmatched bytes from the original binary and the bytes associated with each source excerpt in the database. Methods of source recontextualization are used to incorporate foreign code into candidate decompilations.

We also apply targeted improvement methods to decompilation candidates, such as injecting literals mined from the target binary, and a technique for automatically responding to compiler errors with additional edits to the candidate.

Software. BED is an application of GrammaTech’s open-source Software Evolution Library for program synthesis and repair [3]. Individual program edits are performed by the clang-mutate tool [1]. The source code database and similarity search were provided by the Pliny Database [42].

II. Example

We walk through the BED technique for the small example shown in Figure 1a, using the BED system architecture diagram Figure 2 for reference.

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 Figure 2 (1.a). We call such candidates “frankensteins”. Optionally, the initial population may be seeded with 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 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...
The results of this technique are shown in Figure 1 along with the original source. The HEX-RAYS 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 1b has similar size and form to the original, in Figure 1a, and could plausibly have been written by a human. Although the HEX-RAYS 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 4. 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 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, \(\text{TBin}\). 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 \(\text{TBin}\), line 17 using a disassembly-based fitness function (§III-D) that returns the diff between \(\text{TBin}\) and \(\text{CBin}\). 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 \(\text{Pop}\) 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 final output of the compiler (§III-F).

The remainder of this section discusses each of these stages in greater detail.

A. Creating an Initial Population

The initial population of candidate decompilations for BED can be formed in many ways. We chose to focus on two population-generating strategies:

1) Frankensteins: Frankensteins are composed of whole functions drawn from the code database. Function signatures and boundaries are extracted from the original binary using CodeSurfer®, for Binaries, GrammaTech’s static analysis platform for analyzing stripped binaries, producing a template C file consisting of function prototypes with empty bodies. The bytes for each identified function are then used to perform a similarity search in the code database as described in §III-C5. Multiple frankenstein candidate decompilations are then generated by replacing each empty body in the template C code with random
B. Local Search Techniques

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 convergence on byte-equivalent decompilations. Evolutionary search techniques augmented with local search are called memetic algorithms and have been proven effective in many cases [21, 10].

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, such as addresses, are not suitable candidates for source-level transformation. Also, the compiler may optimize operations such as multiplication and division in such a way that a literal value cannot be gleaned directly from the disassembly.

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", suppose "0x80484e0" is the starting address of the string literal “Hello, world!” We would offer this string as a candidate replacement for mutations in the corresponding region of the decompilation. 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, we mine constants from logical representations of binary fragments. As an example, consider the source fragment \( n = 12 \times k \); this may be represented at the machine level as the a sequence of instructions equivalent to \( n = k; n = (n + n + k) \times 2 \). After identifying instruction sequences in the target binary that match the general pattern of optimized multiplication or division, we generate a formula describing the sequence’s net effect on the machine state. The resulting formula can be mined for constant values; in the example above, we would find that \( k \) was scaled by 12.

Our targeted fixing of source literals works by annotating the target decompilation with mined source literals. During fitness evaluation we identify the regions where the compiled candidate differs from the target binary, and use debugging information from the compilation to map those back to the candidate source. The fixer can use this map to inject any literals mined from a binary region back into the corresponding source region.

2) Other Decompilers: Any available decompiler may also 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 frakensteins to seed a single population. We evaluate the impact of seeding the population with decompilation generated via the HEX-RAYS Decompiler.

B. Local Search Techniques

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 [21, 10].

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 ← Ø
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 ← Ø
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. 4. BED algorithm.
chosen type for the new declaration. This change is similar to the AddInit repair type in SPR [19].

Our suite of compilation-fixing strategies includes techniques designed to counter known problems in contributing decompilers. This significantly increases the utility 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.

The compilation-fixing strategies used by BED are available as part of the Software Evolution Library [3].

C. Evolutionary Search

1) Recomposition: We perform mutations directly on the source text of our individuals, necessitating some cleanup when regions of source code are injected into new contexts. 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 naive 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.

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 5.

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.” The targeted mutations are specialized mutations designed specifically for decompilation. Tuning the set of targeted mutations lets us inject domain-specific knowledge into the evolutionary search. Each basic kind of mutation can be modified by the kind of program region where it should be applied, or the source of program text to use. The available modifiers are:

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

In all cases, the source text is recontextualized (see § III-C1).

3) Crossover: All crossover operations are homologous: 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 [23]. 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 5.

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

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 \textit{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 \textit{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 code database which most closely match the \textit{diff-region} are selected with a higher probability. In our experiments, the Pliny Database [42] was used for code search.

\[1\] 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.
D. Measuring 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. 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.

To normalize the disassembly listings, we remove irrelevant padding instructions used for alignment purposes. We also resolve addresses as described in §III-B1. Finally, we identify instructions that change the program counter, such as jmp

\texttt{0x8014040} or \texttt{call 0x8014080}, and replace the address operand with the section name and offset to ensure differences in program alignment do not negatively impact fitness.

E. Updating the population with Lexicase Selection

Lexicase selection is a genetic programming technique shown to dramatically increase population diversity and long-term evolutionary success in some cases [13]. 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 tend to be retained in the population, 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 that has discovered a correct decompilation for a unique portion of the target binary has a high probability of being retained in the population, even if they have lower overall correctness. The novel code from this candidate responsible will then have many opportunities to be incorporated into other candidate decompilations via crossover.

F. Final cleanup

There are no protections against the evolutionary search accumulating unnecessary source code artifacts. In fact, through a phenomenon known as bloat [25], it is often beneficial to a candidate’s long-term survival to accumulate unused, or dead, genetic material. To compensate for bloat we minimize the final source code using delta debugging [41], systematically removing lines and expressions from the program until a minimal set that compiles to the same binary is found.

Before minimization we pass the source code through the \texttt{clang-format} utility [14] to enforce uniform indentation, improve readability, and to break semantically distinct program elements into separate lines to aid removal via delta debugging.

\footnote{\textsuperscript{2}Computed using a sequence comparison algorithm in Wu \textit{et al.} [37].}

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
Program & LOC & Description \\
\hline
LCTHW [32] & & \\
\hline
hw1 & 5 & Hello world \\
hw3 & 9 & Formatted printing \\
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 \\
hw16 & 88 & Pointers to functions \\
\hline
Euler [2] & & \\
\hline
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 \\
\hline
\end{tabular}
\caption{BENCHMARK PROGRAMS, Learn C The Hard Way (LCTHW) is a C example-based tutorial that provides simple programs illustrating features of the C language [32]. The Project Euler problems are a selection of answers to the Project euler programming challenge [2].}
\end{table}

IV. EXPERIMENTAL RESULTS

Our experiments address five primary research questions:

- **Q1: 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 against a benchmark set. (§IV-B1).

- **Q2: 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).

- **Q3: Performance on optimized binaries.** We evaluate BED across multiple levels of compiler optimization.

- **Q4: Impact of targeted mutations.** We evaluate the utility of our targeted mutation operators for the evolution of byte-equivalence (§IV-B4).

- **Q5: Impact of lexicase selection.** We evaluate the impact of lexicase selection versus standard tournament selection.

A. Methodology

Experimental evaluations were performed against the benchmarks in §IV-A1 using the parameters described in §IV-A1, 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 [32] are taken from an example-based C language tutorial. Each selection demonstrates a specific facet of the language. The Project Euler programs [2] are taken from a popular set of programming challenges. As “big code” projects such as DARPA’s MUSE (http://corpus.museprogram.org/) continue to mature we anticipate the availability of increasingly relevant sample code.
2) Parameters: The experimental parameters used during our BED runs are listed in Table III. Runs were performed at both -O0 and -O2 optimization levels.


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 IV 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 HEX-RAYS Decompiler circa 2015. An attempted comparison against recently published decompilers was not possible; we were unable to obtain executables or experimental data sets for the published results in Dream [38] and Phoenix [31].

B. Experimental results

1) 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 HEX-RAYS Decompiler is used to seed candidate populations. Detailed results are shown in Table V. In those cases where byte-equivalence is not achieved, the resulting decompilation is close to the original program, often achieving byte-equivalence at the function granularity.

In non-byte-equivalent 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. In practice non-byte-equivalent portions are small, as we achieve 81% byte-similarity with the target binary in our experiments. In terms of correctly decompiled lines of code for non-optimized binaries, BED fails to achieve byte-equivalent recompilation for 12% of all lines, improving to 9% of all lines when decompiler seeds are used. Thus, runs that fail to achieve full byte-equivalence still provide useful source for reverse engineering and analysis for the majority of the program. The high level of byte-similarity observed in these experiments indicates the promise of the BED technique.

2) Utility of evolved decompilation: Our readability and correctness metrics, described in § IV-A4, are shown in Table IV. We use byte-equivalence of the resulting binaries as a computable proxy for correctness that does not depend on pre-existing high-coverage test suites, formal specifications, or other externally-provided data.

In comparison to the HEX-RAYS 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 achieved a higher degree of byte-similarity with the target binary, while also producing re-compilable source.

3) Performance on optimized binaries: 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-ingroducing 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 3. In the resultant decompilation, the loop in the original program has been elided and replaced with a sequence of printf statements that compiles, with optimization, to the same bytes as the original; this would not be the case with optimizations disabled.

4) Impact of targeted mutations: The success rates for various mutation types when acting on a population in an evolutionary run appear in Figure 6.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler</td>
<td>clang</td>
</tr>
<tr>
<td>flags</td>
<td>-m32 -g -Ox</td>
</tr>
<tr>
<td>CrossRate</td>
<td>½</td>
</tr>
<tr>
<td>MaxEvals</td>
<td>131072</td>
</tr>
<tr>
<td>PopSize</td>
<td>1024</td>
</tr>
<tr>
<td>TargetChance</td>
<td>½</td>
</tr>
</tbody>
</table>

TABLE III. BED EXPERIMENTAL PARAMETERS.

<table>
<thead>
<tr>
<th>METRIC</th>
<th>BED</th>
<th>HEX-RAYS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte similarity</td>
<td>81.23</td>
<td>69.16</td>
</tr>
<tr>
<td># gotos</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td># casts</td>
<td>18</td>
<td>118</td>
</tr>
<tr>
<td># variables</td>
<td>219</td>
<td>262</td>
</tr>
<tr>
<td># scopes</td>
<td>216</td>
<td>211</td>
</tr>
<tr>
<td># live ranges</td>
<td>185</td>
<td>262</td>
</tr>
<tr>
<td># dead assignments</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td># macros</td>
<td>8</td>
<td>3116</td>
</tr>
<tr>
<td># typedefs</td>
<td>0</td>
<td>698</td>
</tr>
<tr>
<td># characters</td>
<td>22,346</td>
<td>263,835</td>
</tr>
<tr>
<td># lines</td>
<td>1040</td>
<td>8342</td>
</tr>
<tr>
<td># ASTs</td>
<td>4723</td>
<td>5616</td>
</tr>
<tr>
<td># matched literals</td>
<td>287</td>
<td>398</td>
</tr>
</tbody>
</table>

† AST count does not include macros and types

TABLE IV. BYTE-SIMILARITY AND READABILITY METRICS. For each metric in the BED column, an average was computed over both optimized and unoptimized test cases, without the use of decompiler seeding.

3The Project Euler corpus was compiled by Seth Pollen and Ben Welton under the supervision of Prof. Ben Liblit.
of producing neutral candidate decompilations. Generally, mutations that preserve the AST class and only operate on full statements performed the best, creating neutral candidate decompilations—decompilations with the same fitness as before mutation—35–55% of the time, compared to a 15–20% rate of neutral candidate 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 known to be beneficial to evolutionary search [35].

5) 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. Threats to Validity

Unknown compiler and flags. We assume the compiler and flags used to compile the original binary are known. This is less restrictive than it may seem; recent work has demonstrated the automatic detection of such information [27]. Alternately, the compiler and flags could be added to each candidate decompilation and co-evolved as part of the BED search.

Available code database. BED’s output quality is affected by 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. The size of the search space is daunting and the results presented required many thousands of fitness evaluations. On the other hand, search can be performed one function at a time, suggesting a simple divide-and-conquer approach to scaling up BED.

Table V. 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 H-X-RAYS Decompiler. In the first and third columns, compiler optimizations were not enabled (−RAYS), while in the second and fourth columns, compilation was performed with optimizations enabled (−RAYS). Where the H-X-RAYS decompiler did not generate compiling seeds for the evolutionary process, N/A is shown.

![Mutation Effect Distributions](image)

Fig. 6. Mutation success rates by type
VI. RELATED WORK

**Phoenix.** Schwartz et al. [31] 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.

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.

**JSNice.** Raychev et al. [26] 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 identifiers and correct type annotations in 81% of cases on obfuscated code.

**Mutational Robustness.** Software functionality has been found to be surprisingly robust to random perturbations [30]. 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 [28]. Our investigation of the effectiveness of mutations, 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 [4], [19].

VII. FUTURE WORK

**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 additional effort, but does not fundamentally affect the implementation or the performance of the BED technique. Aggregate type declarations could be co-evolved with the population.

**Lifting to LLVM-IR.** The BED technique is not inherently specific to C; for any language \( P \) with a black-box translator \( T : P \rightarrow \text{Binary} \), BED methods give a lifting function \( L : \text{Binary} \rightarrow P \), such that \( T(L(x)) \approx x \) for all binaries \( x \).

The low level virtual machine (LLVM) intermediate representation [17] has become a popular target for program analysis and rewriting. The discussion above suggests that BED could be applied to LLVM-IR instead of C to evolve liftings from machine code to LLVM-IR. We conjecture that BED may work particularly well for this task, due to the relative simplicity of LLVM-IR as compared to the C language. The application of BED to LLVM lifting has the potential to significantly out-perform the current state of the art [33], [8], which exhibit cumbersome recompilation artifacts including explicit representations of activation records on the stack.

**Multi-objective fitness functions.** Multi-objective fitness functions can explicitly target additional desirable properties aside from byte similarity. These might include the following: **Readability Metrics.** Readability metrics (e.g. from §IV-A4) could explicitly be added to the BED fitness function. **Test Suites.** It is possible that leveraging program test suites to guide the search could improve the overall efficiency. Functional correctness may serve as a useful proxy for byte-equivalence, accelerating BED’s evolutionary search.

**Machine learning for local decompilation.** Recurrent neural network (RNN) encoder-decoder models have had great success in translating between natural languages [7]. The large code corpora used by BED has been used to train RNN encoder-decoder models to automatically learn a decompilation function translating short phrases of compiled machine code bytes to C source code decompilation [15]. Such models could subsequently be used to improve the evolutionary decompilation process by directly translating the diff inducing regions of the target binary into C source which could replace the source currently compiling to the bytes of the diff inducing region.
Structured code search and completion. Existing systems for structured source code search [24] and source code completion [6] 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 source code is a deterministically checkable guarantee of full semantic equivalence with 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 output.

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 both readability and full semantic equivalence in a manner which is “future proof” against new source languages, ISAs, compilers, and compiler optimizations.

Acknowledgment

The authors would like to thank David Melski for proposing the original idea behind BED.

References


