Fast, Fine-Grained Equivalence Checking for Neural Decompilers

LUKE DRAMKO and CLAIRE LE GOUES, Carnegie Mellon University, USA
EDWARD J. SCHWARTZ, Carnegie Mellon University Software Engineering Institute, USA

Neural decompilers are machine learning models that reconstruct the source code from an executable program. Critical to the lifecycle of any machine learning model is an evaluation of its effectiveness. However, existing techniques for evaluating neural decompilation models are generally inadequate, especially when it comes to showing the correctness of the neural decompiler's predictions. To address this, we introduce CODEALIGN,[1] a novel instruction-level code equivalence technique designed for neural decompilers. We provide a formal definition of a relation between equivalent instructions, which we term an equivalence alignment. We show how CODEALIGN generates equivalence alignments, then evaluate CODEALIGN by comparing it with symbolic execution. Finally, we show how the information CODEALIGN provides—which parts of the functions are equivalent and how well the variable names match—is substantially more detailed than existing state-of-the-art evaluation metrics, which report unitless numbers measuring similarity.

CCS Concepts: • **Theory of computation → Program analysis**; *Logic*; • **Software and its engineering →** *Software notations and tools*.

Additional Key Words and Phrases: Program Equivalence, Alignment, Program Analysis

1 INTRODUCTION

Native decompilation is the process of reconstructing source code from a compiled executable. Decompilation is used for several security-related code maintenance tasks, including malware analysis, vulnerability research, and patching legacy software for which the corresponding source code is not available. Because a significant amount of information is discarded during compilation, including variables and their names and types, decompilation cannot fully be solved deterministically. Deterministic conventional decompilers focus on code semantics and are constructed using architectures similar to an optimizing compiler [Emmerik 2007], and while an improvement on machine code, still produce difficult-to-read reconstructed programs, because they do not attempt to recover elements like meaningful variable names and types.

As a result, a recent surge of interest has applied neural learning to either recover a specific feature such as variable names [Chen et al. 2022a; Lacomis et al. 2019; Pal et al. 2024; Xiong et al. 2023] or to decompile an entire program [Armengol-Estapé et al. 2024; Cao et al. 2022; Fu et al. 2019; Hu et al. 2024; Jiang et al. 2023; Katz et al. 2018; Liang et al. 2021; Tan et al. 2024]. The latter class of models are termed *neural decompilers*. Neural decompilers are probabilistic, sampling from a distribution of possible source code representations. In theory, they should be able to reconstruct the original source code in some cases. However, they can also hallucinate, and produce code that is not semantically equivalent to the original. Determining how often hallucinations occur is critical for knowing how much a given neural decompiler can be trusted.

[1]Tool available at https://github.com/squaresLab/codealign; experiment replication package at https://github.com/squaresLab/codealigneval

Authors' addresses: Luke Dramko, lukedram@cs.cmu.edu; Claire Le Goues, clegoues@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, PA, USA; Edward J. Schwartz, Carnegie Mellon University Software Engineering Institute, Pittsburgh, PA, USA, eschwartz@cert.org.

ACM 1557-7392/2025/10-ART
https://doi.org/10.1145/3772368

```
int str_list_match(list_t *list, char *text) {
    int match = 0;
    char *wchar = gettext_to_wchar(text);
    for (list_t *node = list; node;
         node = node->next) {
        match = match_node(node, wchar);
        if (match > match)
            match = match;
    }
    free(wchar);
    return match;
}
```

```
int str_list_match(list_t *list, char *str) {
    list_t *walk;
    int best_status = 0;
    char *wc_str = gettext_to_wchar(str);
    for (walk = list; walk; walk = walk->next) {
        int this_status = match_node(walk, wc_str);
        if (this_status > best_status)
            best_status = this_status;
    }
    free(wc_str);
    return best_status;
}
```

Fig. 1. A prediction by a machine learning model (left) and the reference solution (right). Some instructions in the prediction are equivalent to the reference; these are connected with boxes and lines. However, the prediction is subtly incorrect; it always returns the value of evaluating **`match_node`** on the *last* item of the list, rather than the highest one found. Equivalent instructions are connected with boxes and lines.

To perform such evaluations, researchers usually set up an experiment where the correct answer—i.e., the original source code—is available; the neural decompiler's predictions are compared against the original. However, program equivalence is an extremely difficult problem that is undecidable in general. Researchers have created a variety of domain-specific program equivalence techniques that meet the needs of particular use cases, balancing trade-offs of soundness, completeness, efficiency, and applicability. For example, formal, sound, but expensive and incomplete methodologies are used to prove that code produced by an optimizing compiler is equivalent to the original [Churchill et al. 2019; Gupta et al. 2018]. On the other hand, complete, widely applicable, and fast but very unsound methods can be employed to approximate equivalence in a loose way when evaluating machine learning models that generate code against a reference code snippet [Eghbali and Pradel 2022; Papineni et al. 2002; Ren et al. 2020; Tran et al. 2019; Zhou et al. 2023]. They use a variety of heuristics, such as syntactic or the lexical overlap between functions, to produce a score that reflects how similar the two functions are. Unfortunately, all of these existing techniques are either not applicable to neural decompilation or come with significant drawbacks.

Evaluating neural decompilers offers unique challenges that break assumptions made by existing sound equivalence techniques. Ironically, code produced by decompilers, including neural decompilers, usually can't be compiled. This is because external symbols that compilers assume exist, like functions and global variables, are unavailable. Further, even if the code could be compiled, there are rarely any test suites available in situations where decompilation is used. (Tests can be used as a proxy for equivalence themselves, or for performing trace alignment [Churchill et al. 2019]). Generating tests automatically is theoretically possible but difficult in practice, especially when dealing with sophisticated data structures that occur in real code (e.g. a binary search tree is a tree structure with an ordering invariant), or which depend on state (e.g. an open socket). In addition, in malware analysis, an important use case for decompilation, executing pieces of the (malware) program can be dangerous. Finally, evaluating a machine learning model often involves processing hundreds or thousands of predictions, so runtime performance is important.

On the other end of the spectrum, unsound but fast approaches traditionally used to evaluate code predicted by machine learning, such as CodeBLEU [Ren et al. 2020], and CodeBERTScore [Zhou et al. 2023], use various heuristics to approximate program equivalence; for Fig. 1, their values are 0.595 and 0.905, respectively. However, it is not clear how to interpret these scores; they are unitless. (These numbers should not be interpreted as proportions.) Further, it is not the case that a sufficiently high score indicates equivalence, as we show in Section 5.

Beyond equivalence of entire code fragments, it is often desirable to know what *parts* of the neural decompiler's predictions are equivalent to the reference. We illustrate with an example; Fig. 1 shows such a pair of functions. The non-control-flow instructions that are equivalent are connected with boxes and lines. The reference code

searches through a linked list to find the maximum value. The generated code correctly manages memory for a temporary object (the `gettext_to_wchar` and `free` calls are equivalent) and correctly interates through the list (the `->` operations are equivalent), albeit with slightly different syntax and variable names. However, the model had a hallucination: it uses just a single variable to represent both of the current and maximum value. The `match > match` and `match = match` are not equivalent to `this_status > best_status` and `best_status = this_status`. As a result, the generated function always returns the value of the last element in the list.

In this work, we introduce CODEALIGN, an instruction-level equivalence technique designed for neural decompilation. CODEALIGN produces an *equivalence alignment*: a relation of equivalent instructions between two functions. Intuitively, two instructions are equivalent if and only if the result of executing those instructions is the same for all inputs. (For a formal definition see Section 2). The boxes and lines in Fig. 1 illustrate an equivalence alignment. This granular, low-level equivalence representation allows for detailed analyses of the results. For example, to measure how well the variable names in generated code match those in the reference, one can identify the correspondence between variables in the generated code and variables in the reference code. With this correspondence, the generated variable names can be compared to the reference names. For instance, in Fig. 1, `walk` and `node` are equivalent; they both store the current node while iterating over the list. However, their names are not similar, which could be confusing. CODEALIGN's equivalence alignments make it easy to determine which variables correspond.

While neural decompilation is a challenging domain, the nature of the neural decompilation affords opportunities as well. A key insight that enables CODEALIGN is that the decompilation and original code should be *implementations of the same algorithm*. Compilation is a lossy, many-to-one function: textually distinct source code can map to the same sequence of assembly instructions, especially with optimizations. A neural decompiler therefore cannot deterministically match the original source code all of the time. Frequently, there will be differences in variable name and (irrelevant) statement ordering; expressions may also be broken up differently. However, the assembly instructions necessarily preserve the algorithm itself accuracy, and so a neural decompiler should be able to reproduce the algorithm in some form. A mark of an incorrect neural decompilation is one in which the algorithm is different.

We contribute the following:

- the CODEALIGN tool which builds alignments from C (and has some support for Python). CODEALIGN is available at https://github.com/squaresLab/codealign and a replication package for the experiments at https://github.com/squaresLab/codealigneval
- an equivalence alignment generator based on symbolic execution for comparison
- a demonstration of CODEALIGN's utility in evaluating both the correctness and variable name quality of code produced by a neural decompiler against a reference

2 DEFINITIONS

CODEALIGN operates on pairs of functions. We model each function as a sequence of instructions p_j. Each instruction operates on values v_i, where each value is either an argument to the function, a constant, or the result of executing another instruction. These values are called the instruction's *operands*. The class of computation that an instruction performs is defined by its *operator*, such as `+`, `>`, and `strcmp`. As an example from Fig. 1, `this_status > best_status` is an instruction; `this_status` and `best_status` are its operands; and `>` is its operator. All of the items in boxes in Fig. 1 are operators.

Value-instruction Binding. Without loss of generality, we define an equivalence alignment in terms of SSA-form code [Alpern et al. 1988; Rosen et al. 1988]. In SSA form, variables are statically immutable: they are assigned to at most once during the program. We borrow the term *value* from the SSA literature (and in particular the LLVM

compiler infrastructure) to refer to any object that can function as the operand to an instruction. This includes SSA-style immutable static variables, but also constants and the enclosing function's arguments.

A value binding is equivalent to the assignment of an instruction to an SSA immutable static variable. Let f be a function, let $v \in f$ be a value, and let p_j be an instruction. We say that v_j is bound to p_j if v_j is the result of executing p_j.

Equivalent Values. A common way to define functions as semantically equivalent is to say that two functions are equivalent if they have the same output for every input. Formally, $f = g \iff \forall i, f(i) = g(i)$. We use a similar definition to define equivalent values. Let v be a value and let $v^{f(i)}$ be the *dynamic* value of v when function f is executed on input i. Then

$$v_j = v_k \iff \forall i, v_j^{f(i)} = v_k^{g(i)}. \tag{1}$$

We say that v_j and v_k are *functionally equivalent*. Of course, a given static value v may have multiple dynamic values if that variable occurs within a loop. To define the equivalence of values in a loop, we use induction. Any variable that is changed within a loop has an associated ϕ instruction at its head. Such a ϕ instruction is equivalent to another if:

- *Base case*: the values of the ϕ instructions' operands from outside the loop are equivalent. (There is always such an operand because all loops have an entry point).
- *Inductive Step*: given equivalent ϕ instructions, the values of their operands inside the loop are equivalent.

Equivalence Alignments. An equivalence alignment is a relation between instructions in two functions. Let f and g be two functions, and let F and G be sets of all values that occur in each function, respectively. Let v_j be the instruction bound to value p_j. We define an equivalence alignment as a subset of the cartesian product $F \times G$:

$$\{(p_j, p_k) | v_j \in F, v_k \in G, v_j = v_k\} \tag{2}$$

That is, an equivalence alignment consists of the instructions whose results are equivalent. An equivalence alignment is a relation; it is not necessarily a function.

Note that this definition of an equivalence alignment, is, like program equivalence in general, undecidable. In practice, CODEALIGN uses a sound but incomplete definition of equivalence, which we discuss in Section 3.2. An equivalence alignment is related to a product program [Barthe et al. 2011], where elements related to each other are functionally equivalent.

3 CODEALIGN

CODEALIGN takes two functions f and g as input and outputs an equivalence alignment. Fig. 2 shows an example pair of functions f and g and their internal representations as they progress through the different stages of CODEALIGN. To begin, CODEALIGN converts each function to SSA form, and computes control dependencies for each instruction. Using this information, it creates lemmas that will be used to prove parts of f and g equivalent. In the final step, CODEALIGN iteratively proves that instructions are equivalent using induction (Fig. 2b). Codealign maintains pointers to the AST so that the alignment can be used to reason about the code itself, not a derived representation of it.

3.1 Pre-processing

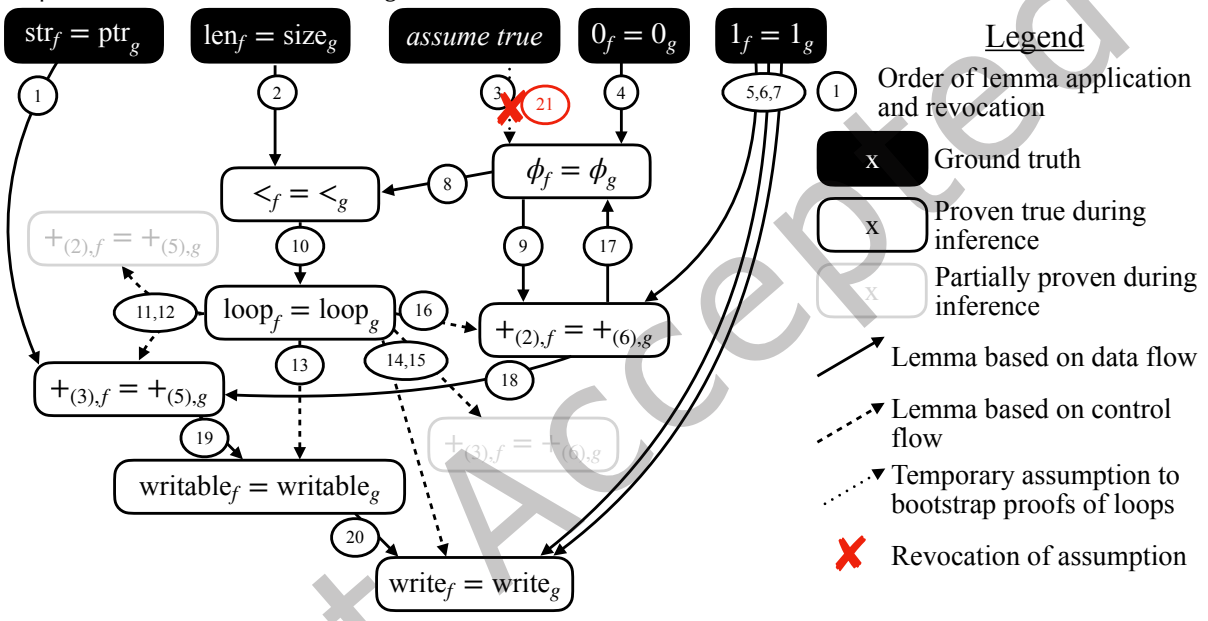
Because CODEALIGN operates on functions in isolation, and does not require header files or libraries, it first heuristically classifies unknown identifiers in functions based on their use. For example, undeclared identifiers in call expressions are assumed to represent functions. If the code treats an undeclared identifier as a value, CODEALIGN interprets it as a global variable. CODEALIGN then converts the IR to SSA form, performs standard

```c
void f(char *str, size_t len) {
  for (size_t i = 0; i < len; i++)
    write(1, writable(str + i), 1);
}
```

```c
void g(char *ptr, size_t size) {
  size_t i;
  i = 0;
  while (i < size) {
    write(1, writable(ptr + i), 1);
    i++;
  }
}
```

```
ssa_f(str, len) {
  %0 = ϕ(0, %4)
  %1 = %0 < len
  loop (%1) {
    %2 = str + %0
    %3 = writable(%2)
    write(1, %3 , 1);
    %4 = %0 + 1;
    }
}
```

(a) Two functions, f (left), g (middle) and f's SSA form (right). The SSA form of g is the same as f's except for parameter names. Despite their different syntax, each value in each function is equivalent to one value in the other. Because f and g contain multiple + instructions, we use the source line to differentiate them, e.g., $+_{(3),f}$ refers to the + on line 3 in f. We adopt LLVM's convention of illustrating SSA instruction values as %0, %1, etc.



(b) An inductive proof graph showing values from the functions in Fig. 2a equivalent. An instruction in f can be proven equivalent to an instruction in g if all of its' control- and dataflow dependencies can be proven equivalent. The graph is initialized with only ground truth nodes, and adds others as lemmas are applied in the order specified.

Fig. 2. An illustration of how CODEALIGN works. Fig. 2a shows two examples (derived from the data in Section 5) and their canonicalized SSA representations. Fig. 2b shows how, using pairs of SSA-representation edges as lemmas, various values in the examples can be proven equivalent.

copy propagation, and applies normalization and desugaring rules (e.g., converting `x++` to `x = x + 1`) designed to enable very similar code to be detected as equivalent. Fig. 2a shows an example.

CODEALIGN operates on an SSA-based data flow graph: nodes represent instructions, and edges represent a data-flow between them. Note that incoming edges, which represent the operands of a given instruction, are *ordered* because the order of operands is semantically important. The graph also treats SSA ϕ instructions as instructions that receive their own nodes. CODEALIGN also uses control dependence information from a control

dependence graph; nodes in the control dependence graph represent basic blocks and edges represent the control dependencies between basic blocks.

3.2 Lemma Generation

Pairs of equivalent values, one from f and the other from g, serve as propositions in CODEALIGN's logical system. Lemmas are implications that relate information about propositions to each other. CODEALIGN uses the SSA representations of the functions as well as control dependence information to generate lemmas to be used in the inductive phase.

Because finding all functionally equivalent values, as defined in Section 2, is undecidable, CODEALIGN uses a sound but incomplete notion of equivalence, which we term *dependency-based equivalence*. Dependency-based equivalence is based on the intuition that performing the same computation (that is, executing instructions with the same operator) on the same operands under the same control flow conditions will produce the same result. Let $v_j \in f$ and $v_k \in g$ be values bound to instructions. Informally, lemmas take the form

$$\text{all dependencies of } v_j \text{ and } v_k \text{ align } \wedge \text{ operator}(v_j) = \text{operator}(v_k) \implies v_j = v_k \tag{3}$$

Naively, there could be an equivalence lemma for each combination of instructions from f and g. If there are n instructions in f and m instructions in g, then there would be nm lemmas. However, if the operators for two instructions are different then it is impossible to satisfy the conditions of any lemma of the form given in Equation 3. Thus, CODEALIGN simply does not generate these lemmas, which speeds up lemma generation.

In CODEALIGN, we use an equivalent but slightly different form for representing lemmas that makes it easier to determine when the conditions of a lemma are satisfied. Instead of generating one lemma for each pair of instructions with n dependencies $\{d_1, \ldots d_i, \ldots d_n\}$, we generate n lemmas, one for each pair of dependencies. (In CODEALIGN, incoming edges are ordered, so there are not n^2; we discuss ordering below). The conclusion of each lemma is associated with a weight of $1/n$; the conclusion is only considered proven if the total proven weight is 1. These take the form

$$d_{i,j} = d_{i,k} \implies (v_j = v_k, \text{weight} = 1/n) \tag{4}$$

The weight formulation is useful for how we handle loop induction (Section 3.3). In Fig. 2, there are three lemmas concluding in $+_{(3),f} = +_{(5),g}$, where $+_{(3),f}$ refers to the `+` on line 3 of f and $+_{(5),g}$ refers to the `+` on line 5 of g.

(1) $\text{str}_f = \text{ptr}_g \implies +_{(3),f} = +_{(5),g}$
(2) $+_{(2),f} = +_{(6),g} \implies +_{(3),f} = +_{(5),g}$
(3) $\text{loop}_f = \text{loop}_g \implies +_{(3),f} = +_{(5),g}$

The first two are data flow lemmas, and the third is a control-flow lemma.

Note that if v_j and v_k have different numbers of data flow dependencies or different numbers of control dependencies, these instructions cannot be shown equivalent with CODEALIGN. We next make the notion of lemmas more precise, considering data- and control-flow dependencies in turn.

3.2.1 Data Flow Lemmas. Data flow lemmas are based on the operands of the instructions $v_j \in f$ and $v_k \in g$. In general, the operands to an instruction cannot be rearranged without changing the semantics of the instruction. For example, `strcmp(a, b)` ≠ `strcmp(b, a)`. Thus, CODEALIGN builds data flow lemmas by positionally proposing that the ith operands are equivalent.

Some instructions have no operands (primarily function calls with no arguments). These instructions receive a lemma of the form "true $\implies v_j = v_k$" which is counted, for the purposes of computing the weight, as a single data flow dependency.

3.2.2 Control Flow Lemmas. CODEALIGN also generates lemmas based on the control flow conditions under which instructions execute. In general, control dependencies are defined at the basic block level, rather than the

instruction level, though CODEALIGN's proof engine operates on the instruction level. As a result, if basic block A depends on basic block B, we say that all instructions in A depend on the instruction in block B which induced the control flow (which is necessarily a branch instruction, like `loop` or `if`). In Fig. 2, $\text{loop}_f = \text{loop}_g \implies +_{(3),f} = +_{(5),g}$ is a control-flow lemma. The `+` instruction on line 3 of f is in a basic block that is control dependent on the block ending with the `loop` instruction in f. Likewise, the `+` instruction on line 5 of g is in a basic block that is control dependent on the block ending with the `loop` instruction in g. Combining these forms the lemma. Control dependencies are *labeled* with the branch decision (true or false) at each branch statement. To ensure that CODEALIGN models the control flow behavior of each function, CODEALIGN only generates lemmas when they follow the same branch decision (e.g. are both true or both false).

A basic block may have more than one control dependency. It is not immediately clear how to build lemmas when one or more of the instructions have multiple control dependencies. If a pair of instructions each has n control dependencies, then there are n^2 possible combinations. Intuitively, it is not the case that each control dependency must be equivalent to each other control dependency. Rather, some subset must be equivalent.

In this work, we develop a novel method of *ordering* control dependencies such that two instructions can only align if their control dependencies align in that order. If the control dependencies of instructions $p_j \in f$ and $p_k \in g$ are placed in order, then the only way for p_j and p_k to be equivalent is if the ith control dependencies are each equivalent to each other. Ordering control dependencies significantly decreases the number of combinations of dependencies that have to be examined and allows one to know exactly which dependencies need to be compared to show to instructions equivalent.

To show the validity of our ordering strategy, we begin with several definitions. Let A, B, C be distinct basic blocks in f, and A', B', and C' be distinct basic blocks in g.

The definition of alignment states that a basic block A in function f aligns a basic block A' in function g if and only if its operators are equivalent and its dependencies align. Here, we are concerned with control dependencies; each control dependency of A must align with a control dependency of A'. Alignment of control dependencies is a necessary condition for alignment. Formally if D_A is a control dependency of A, and D'_A is a control dependency of A',

$$A \text{ aligns } A' \implies \left(\forall D_A \exists D'_A \text{ s.t. } D_A \text{ aligns } D'_A\right) \wedge \left(\forall D'_A \exists D_A \text{ s.t. } D_A \text{ aligns } D'_A\right) \tag{5}$$

Next, we define indirect (transitive) dependence. A is indirectly (transitively) dependent on C if A is dependent on C or if any dependency B of A is indirectly dependent on C. In other words, A is indirectly dependent on C if there exists a path in the control dependence graph from A to C. For brevity, we use "dep" to denote "depends on", and "idep" to denote "indirectly depends on". Second, we make use of a depth-first node order (denoted "dfo : BasicBlock $\to \mathbb{N}$"), also known as a reverse-post order. This dfo is defined such that the left branch is less than the right branch. The opposite convention can be obtained by reversing the order in which successors are visited when building the depth-first spanning tree for the dfo. Finally, for brevity, we use the phrase "A aligns A' " to mean "the instructions in A and A' " align[2].

We define $<$ for ordering control dependencies as:

$$C < B = \begin{cases} \text{dfo}(C) < \text{dfo}(B), & \text{if } C \text{ idep } B = B \text{ idep } C \\ false, & \text{if } C \text{ idep } B \wedge \neg(B \text{ idep } C) \\ true, & \text{if } B \text{ idep } C \wedge \neg(C \text{ idep } B) \end{cases} \tag{6}$$

We assert that, if A dep B, C and A' dep B', C':

$$B \text{ aligns } B' \wedge C \text{ aligns } C' \implies (C < B \wedge C' < B') \vee (B < C \wedge B' < C') \tag{7}$$

[2]If A is a dependency to another block, only the branch instructions needs to align for that block to align.

In other words, ordering control dependencies according to < as defined in Equation 6 is a necessary condition for aligning instructions in basic blocks A and A'. We first prove two useful facts which we will need to show Equation 7.

For the first fact, let X, Y and Z be basic blocks in f, and let W be a basic block in g. Further, let Z idep X, Y, and let X and Y be not necessarily distinct. Then:

$$X \text{ aligns } W \land Y \text{ aligns } W \implies X = Y \tag{8}$$

Proof of Equation 8 by contradiction and induction. Assume X aligns W, and Y aligns W, but $X \neq Y$ (that is, they are distinct). By the definition of alignment, each control dependency of X aligns with a control dependency of W. The same is true for Y and W. This means X and Y have the same number of control dependencies. Then we have:

Base Case: X and Y have no control dependencies. By definition, if X and Y have no control dependencies, then control always flows through X and Y. In other words, all paths through the CFG pass through X and Y, including all paths that pass through Z. Either X or Y must be executed first, since they are distinct. Say X is executed first. (The argument for Y is symmetrical). Because Z is indirectly dependent on X, by definition, there is a sequence of control dependencies $D_1 \dots D_k$ such that Z dep D_1, D_i dep D_{i+1}, and D_k dep X. Y is an indirect dependency of Z, but Y cannot be in any such $D_1 \dots D_k$ because Y has no control dependencies. So Y must be on another path from X to Z. But Y this means that Y is indirectly control dependent on X, and Y has no control dependencies. We have reached a contradiction, so it must be the case that when X and Y have no control dependencies, X = Y.

Inductive Step: X and Y have control dependencies. By inductive assumption, for all dependencies D_X of X there exists a dependency D_Y of Y such that $D_X = D_Y$, and vice versa. We call such a shared dependency D. Because D is a control dependency of X, there must be a CFG path from D to X that is postdominated by X. Y cannot be in that path, because X postdominates all bocks on that path; if Y were postdominated by X, it would not be an indirect control dependency of Z. Likewise, there is a path postdominated by Y from D to Y that does not contain X. This means that the paths D to X and D to Y must meet at D and are distinct. Thus, they are entered by different branch conditions (i.e. one is the "true" branch from D, the other is the "false" branch). But X aligns W and Y aligns W, which means that W is dependent on both the true and false branches of the block in g aligned with D. This is a contradiction, and so X = Y when X and Y have control dependencies. This completes the inductive step. □

Using this, we show:

$$B \text{ idep } C \land \neg(B' \text{ idep } C') \land C \text{ aligns } C' \implies \neg(B \text{ aligns } B') \tag{9}$$

Proof of Equation 9 by contradiction. Suppose B idep $C \land \neg(B'$ idep $C) \land C$ aligns $C' \land B$ aligns B'. Because B idep C, by definition, there exists some sequence of dependencies $D_1, ..., D_k$ between B and C such that B dep D_1, D_i dep D_{i+1}, and D_k dep C. Then, because B aligns B', each dependency of B must align with a dependency of B'. So there must exist some D_1' that aligns with D_1. Likewise, D_1 can only align with D_1' if all of their dependencies align, and so on. So there is a sequence of D_i' such that D_i aligns D_i' Again by the definition of alignment, there must be some dependency E' of D_k' that aligns with C. By Equation 8, E' must be C'. (In Equation 8's terms, $W = C$, $X = C'$, $Y = E'$, $Z = D_k'$). So there exists a sequence of control dependencies from B' to C', which means B' idep C'. However, $\neg(B'$ idep $C')$. We have reached a contradiction, so $\neg(B$ aligns $B')$. □

With this, we can now prove that ordering according to Equation 6 is a necessary condition for alignment.

Proof of Equation 7. We assume B aligns $B' \wedge C$ aligns C'. To show that the $(C < B \wedge C' < B') \vee (B < C \wedge B' < C')$, we must use Equation 6, the definition of <. Equation 6's conditions are expressed in terms of the dependence relationships among the dependencies of A and A'. We consider all combinations of dependence relationships between B and C, as well as all possible dependence relationships between B' and C'. By Equation 9, however, we need only consider cases where the dependence relationship between B and C matches the dependence relationship between B' and C'. (Likewise, the dependence relationship between C and B must match the dependence relationship between C' and B'). Otherwise, by Equation 9, we have $\neg(B \text{ aligns } B')$ or $\neg(C \text{ aligns } C')$, which is a contradiction.

Case 1: $\neg(C \textbf{ idep } B) \wedge B \textbf{ idep } C \textbf{ and } \neg(C' \textbf{ idep } B') \wedge B' \textbf{ idep } C'$

Plugging this set of conditions into Equation 6 means choosing the third case, which means C < B. The same is true for B' and C'. So $C < B \wedge C' < B'$.

Case 2: $C \textbf{ idep } B \wedge \neg(B \textbf{ idep } C) \textbf{ and } C' \textbf{ idep } B' \wedge \neg(B' \textbf{ idep } C')$

Plugging this set of conditions into Equation 6 means choosing the second case, which means $B < C \wedge B' < C'$.

Case 3: $\neg(C \textbf{ idep } B) \wedge \neg(B \textbf{ idep } C) \wedge \neg(C' \textbf{ idep } B') \wedge \neg(B' \textbf{ idep } C')$

First, we show that B and C have a most recent common indirect control dependency D. There exists a path from the entry block to A that goes through B. Likewise, there exists a path from the entry block to A that goes through C. These paths must both contain the entry block, so such a common ancestor in the CFG always exists. Because B is a control dependency of A, there exists a path from B to A postdominated by A (except B). C cannot be on this path, because C, as a control dependency, is not postdominated by A. Likewise, B cannot be on the path from C to A. Therefore, the paths from the common control flow ancestor through B and C to A differ by at least one node. This means at some node, the paths diverge. This node must be an indirect dependency of B and C, because it controls whether control can flow to B or C. We call this node D.

The above also true for A', B', and C'. We denote the common direct descendant of B' and C', D'.

B aligns B', so the dependencies of B must align the dependencies of B'. By Equation 8, all such dependencies are unique. Further, this must be true of the dependencies of the dependencies of B, and their dependencies, recursively. This includes D, which means D aligns D'. Because D aligns D', either B and B' are both reached via the true branch of D and D', or both via the false branch. C and C' must then each be reached following the other branch condition.

Thus, either $\text{dfo}(B) < \text{dfo}(C) \wedge \text{dfo}(B') < \text{dfo}(C')$ or $\text{dfo}(C) < \text{dfo}(B) \wedge \text{dfo}(C') < \text{dfo}(B')$. In turn, by Equation 6, $B < C \wedge B' < C'$ or $C < B \wedge C' < B'$.

Case 4: $C \textbf{ idep } B \wedge B \textbf{ idep } C \wedge C' \textbf{ idep } B' \wedge B'$ **B' idep C'**

In codealign's loop-proof system, dependencies corresponding to forward edges must be aligned before dependencies corresponding to back-edges. Further, the dfo for the forward edge is necessarily less than the one corresponding to the back edge. Either C must have been aligned with C' first or B must have been aligned with B' first. If C was aligned first, we have $\text{dfo}(C) < \text{dfo}(B) \wedge \text{dfo}(C') < \text{dfo}(B')$, so by Equation 6, $C < B \wedge C' < B'$. Conversely, if B was aligned with B' first, then we have $\text{dfo}(B) < \text{dfo}(C) \wedge \text{dfo}(B') < \text{dfo}(C')$, so by Equation 6, $B < C \wedge B' < C'$.

□

3.2.3 Function Pointers. The preceding discussion assumes a distinction between values and instructions, but this is not always the case when functions can be passed as values (such as with function pointers). Calls to function pointers don't have a statically defined operator. We consider function pointers to have equivalent operators if the values they call have themselves been proven equivalent; otherwise, they are treated like other instructions.

3.3 Loops

We calls cycles in the control flow graph loops. Like many program analysis tools, CODEALIGN assumes reducible flow graphs, which means all loops must be *natural loops*. Natural loops have a single basic block at which the loop is entered, called the head. All natural loops have a *back-edge*, which is an edge from B to A such that A dominates B. In Fig. 3, the CFG edge from blocks 5 to 1 is a back edge.

Loops may result in cycles in of dependencies. These cycles may contain only data flow dependencies, only control flow dependencies, or both control and data flow dependencies. For instance, when mutable variables are updated in the body of a loop, a cycle of dataflow dependencies is formed. In Fig. 2a, this occurs in both functions when the i variable is incremented. A loop with a break statement inside an if statement may result in a cycle of control dependencies: the loop branch is control dependent on the if branch, and the if branch is control dependent on the loop branch. Fig. 3 shows the CFG for such a loop. And in `do...while` loops, the head of the natural loop is control dependent on the block with the loop's branch statement, at the end of the loop. At the same time, the branch statement may be computed using values in the head of the natural loop. This creates a cycle of both control and dataflow dependencies.

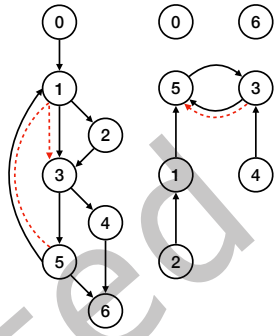


Fig. 3. A CFG (left) and CDG (right). There is a cycle of control dependencies between blocks 5 and 3. The red dashed line indicates a back-dependency, as does the CDG edge from 1 to 5.

Cycles of dependencies result in cycles of lemmas, which make it impossible to prove two loops equivalent unless special consideration is given to loops. In Fig. 2 to prove the + instructions from the desugared i++ expressions equivalent, the ϕ instructions must be shown equivalent. But the values of those same + instructions are operands to the ϕ instructions.

To resolve this, CODEALIGN identifies *back-dependencies*: dependencies that can only be reached from the dependent block by following the loop's back-edge when traversing the CFG. Fig. 3 highlights such a dependency. (Intuitively, this can be thought of as dependency edges where the dependency is "above" the dependent block in the code). Data flow back-dependencies necessarily end in a ϕ instruction at the loop's head. CODEALIGN examines each operand of each ϕ instruction at the head of a loop to determine if it is a back-dependency. Unlike data flow dependencies, control back-dependencies do not necessarily end at the loop's head. The highlighted dependency in Fig. 3 is an example.

CODEALIGN uses induction to bootstrap proofs of loops. As discussed in Section 2, the base case(s) are the dependencies from outside the loop. For the inductive step, CODEALIGN assumes that the ith loop iterations are equivalent, and attempts to prove the $i+1$st iterations equivalent. The necessary assumptions are precisely the back-dependencies previously identified. In Fig. 2, there is a cycle of dependencies between the ϕ instructions and the $+_{(2),f}/+_{(6),g}$ instructions.

Like any other dependency, each back-dependency is modeled as a lemma in CODEALIGN's logical system. For each back-dependency lemma of the form $A \implies B$ with weight $1/n$, CODEALIGN adds an Equation 4-form lemma of the form $\text{true} \implies B$ with weight $1/n$. For Fig. 2, that's $+_{(2),f} = +_{(6),g} \implies \phi_f = \phi_g, 1/2$ and $\text{true} \implies \phi_f = \phi_g, 1/2$, respectively. This means the total provable weight of proposition B is greater than 1, and that B can be proven without using any back-dependency lemmas. Adding these lemmas is the way CODEALIGN assumes the inductive hypothesis. In Fig. 2, the base case is completed by satisfying the condition of $0 = 0 \implies \phi_f = \phi_g, 1/2$, while inductive hypothesis is completed by satisfying $\text{true} \implies \phi_f = \phi_g, 1/2$. Then, to complete the inductive step, CODEALIGN attempts to satisfy the conditions of the back-dependency lemmas. If it can, then it has successfully inductively proven the loop; otherwise, it is unable to show the loops equivalent. In this case it can; using the facts that $\phi_f = \phi_g$ and $1 = 1$, $+_{(2),f} = +_{(6),g}$. This satisfies the condition of the back dependency lemma $+_{(2),f} = +_{(6),g} \implies \phi_f = \phi_g, 1/2$. If CODEALIGN is unable to show the loops equivalent, it

recursively revokes lemmas and propositions proven based on the assumptions. The mechanism by which this happens is explained in more detail in Section 3.4.

3.4 Inference

After all lemmas have been generated, CODEALIGN begins proving values equivalent by induction (at the instruction level, rather than the loop level as described in Section 3.3). The base cases consist of facts which are known or assumed to be equivalent a priori. Function parameters in the same positions are assumed to be equivalent, even if they have different names.[3] In doing so, CODEALIGN ignores the code authors' *intentions* of what should be passed to each parameter and models what *would* happen if the same parameters are passed. Similarly, global variables are assumed to be equivalent if they have the same name. Identical constant values are considered equivalent.

Lemmas are represented in the form shown in Equation 4. This allows them to be easily indexed in a hash table by their condition. A pair of values is proven equivalent when that proposition has total accumulated weight 1.0. When this happens, finding the lemmas whose conditions are subsequently satisfied is a simple dictionary lookup.

CODEALIGN builds its proof using a worklist algorithm as shown in Algorithm 1. The base cases are inserted into a queue, the worklist. Then, at each iteration, a proposition is popped from the worklist. If the equivalence satisfies the condition of any lemmas, the conclusions of those lemmas are added to the worklist. The process repeats until the worklist is empty. Whenever a proposition a lemma's conditions are satisfied, it is added to a proof-graph data structure. The nodes in the proof graph are propositions—pairs of equivalent values—and the edges are lemmas that relate one proposition to another. When the worklist is empty, CODEALIGN revokes any lemmas added to enable the induction of loops and decreases the weight associated with each corresponding conclusion. Then, if the back-dependency lemmas were not proven, the weight of the nodes assumed true will drop below 1.0. When this happens, CODEALIGN removes each lemma and proposition from the proof-graph that is reachable from the assumed node, thus removing any propositions proved based on the failed loop induction. Any remaining proposition that is still true in the graph becomes a part of the alignment. An example proof graph after attempting to align f and g is shown in Fig. 2b.

```
Data: lemmas: dict[Equivalence,
      list[Equivalence]]
Data: baseCases: list[Equivalence]
Data: loopAssumtions: dict[Equivalence,
      list[Equivalence]
worklist ← Queue(baseCases)
proofGraph ← ProofGraph(baseCases)
while worklist is not empty do
    condition ← worklist.pop()
    conclusions ← lemmas[condition]
    for conclusion in conclusions do
        weight ← proofGraph.addEdge( condition,
          conclusion)
        if weight = 1.0 then
            worklist.put(conclusion)
        end
    end
end
for condition in loopAssumptions.keys() do
    conclusions ← loopAssumptions[condition]
    for conclusion in conclusions do
        proofGraph.removeEdge( condition,
          conclusion)
    end
end
return proofGraph.validNodes()
```

Algorithm 1: CODEALIGN's inference algorithm. An Equivalence object represents a proposition that two values are equivalent. Lemmas are represented as a dictionary, with the lemma condition as the key and the conclusion the value. We group lemmas with the same conclusions in a list.

[3]We ignore types when matching function arguments. See Section 4.1.2.

3.5 Options

CODEALIGN has several settings that can modify its behavior. We find it useful to allow for partial proofs of loops. Even if the beginning of two loops are equivalent, any differences within the loop will result in the whole loop failing to align. As a result, CODEALIGN also offers an option called "partial loop" mode. In this mode, inductive loop assumption lemmas are generated, but back-dependency loop edges are not. Further, loop-assumption lemmas are not revoked (in Algorithm 1, the set of nested for loops before the return are ignored). Partial-loop mode is useful for diagnosing the cause of failed loop alignments: that is, determining which instructions failing to align caused all other instructions in the loop to fail to align. We demonstrate the use of this feature in Section 5.2 in Figure 5. Because the majority of the candidate and reference function are each one loop and because those loops are not equivalent, standard CODEALIGN will fail to align all instructions in the loop. This is correct, but unhelpful for reasoning about *why* the two loops are nonequivalent. With partial-loop mode enabled, it becomes easy to pinpoint that the difference is caused by the differing + instructions.

Additionally, control dependencies can be disabled, in which case CODEALIGN uses only data flow dependencies to perform its alignment. This makes CODEALIGN more flexible in what it can align. We don't disable control dependencies in any experiment in the paper.

Using either of these options makes CODEALIGN unsound, but can still be useful.

4 EVALUATION

In this section, we characterize the capabilities of CODEALIGN. In the next section, we show how CODEALIGN can be used to evaluate neural decompilers.

First, in Section 4.1 we substantiate our claim that CODEALIGN's dependency-based equivalence is sound, and describe the limitations of this claim when applied to real code. Next, we show how CODEALIGN performs on full-featured C, illustrating the applicability of dependency-based equivalence. While there do exist tools that solve related problems [Badihi et al. 2020; Lahiri et al. 2012; Person et al. 2008], we are unaware of any existing tool which produces equivalence alignments against which we can directly compare CODEALIGN. Instead, we appeal to symbolic execution, a widely-used technique in program analysis and equivalence work. In Section 4.2, we describe how we use symbolic execution to build approximate reference alignments and compare against them. While these are both unsound and incomplete, they serve as a useful reference to more intuitively characterize CODEALIGN's behavior. Finally, in Section 4.4, we measure CODEALIGN's runtime performance.

4.1 Correctness

CODEALIGN is sound, subject to some well-defined limitations. We claim that instructions that CODEALIGN aligns have functionally equivalent values, as defined in Section 2; however, CODEALIGN is necessarily incomplete and will not find all functionally equivalent values.

4.1.1 Soundness. By construction, the set of aligned nodes CODEALIGN produces are isomorphic in terms of their dataflow graphs and their control dependence graphs. This is because in order to align, all dependencies of a pair of instructions must align, so if an node is added to the alignment, all of the edges to it from nodes already in the graph are added along with it. Programs with isomorphic program dependence graphs and SSA forms are semantically equivalent [Yang et al. 1989]. Therefore, CODEALIGN is sound.

Computing graph isomorphisms is a difficult problem, and there are no known polynomial time algorithms. It is not known if finding a graph isomorphism is NP-complete [Grohe and Schweitzer 2020]. However, CODEALIGN

is able to do this efficiently by taking advantage of the characteristics of SSA form and control dependence graphs to significantly reduce the search space of combinations.[4]

4.1.2 Limitations. [Yang et al. 1989]'s theorem is defined only for programs meeting certain characteristics; CODEALIGN inherits these limitations.

Side Effect Ordering: CODEALIGN considers the equivalence of instruction side effects independently from the side effects of other instructions. For instance, if given `printf("A"); printf("B");` with `printf("B"); printf("A");`, CODEALIGN would align the `printf("A")`s and `printf("B")`s. While each of these instructions are indeed equivalent individually, the two programs write different output to the screen. CODEALIGN could be extended to differentiate instructions based on their side effects by adding edges between instructions that produce side effects if there exists a path on which both could be executed. CODEALIGN assumes that program state at the start of executing each function is the same. (Otherwise, even textually identical functions could produce different results.) What makes CODEALIGN unsound with respect to side effects is the possibility of executing instructions with side effects in different orders, changing some global state in a different way or producing different output. Side-effect dependencies would eliminate this risk, though some of the power of CODEALIGN is in its ability to flexibly align instructions out of the order they appear in the code text. In principle, any function call could have arbitrary side effects, drastically limiting flexibility. Modifications to mutable data structures are modeled as function calls with side effects in CODEALIGN; these at least would only need to have dependencies between operations on the same data structure. Finally, if the side effects are reflected in control or data flow or the instructions otherwise have different dependencies, as is often the case, CODEALIGN will remain sound.

Types: CODEALIGN does not presently consider types when building an alignment. In practice, different types are usually operated on by different instructions, so this does not have significant practical effect. In the worst case, overloaded operators (like +) can in principle lead to nonsensical lemmas containing propositions asserting values of two different types are equivalent. CODEALIGN could be extended to build on the frontend language's existing type inference system to infer types and then use them to rule out nonsensical lemma construction. In a recent work on type-aware neural decompilation [Dramko et al. 2025], we do exactly this, though implemented externally to CODEALIGN, rather than as part of the tool itself. (We reject alignments where aligned variables do not have the same types.)

Irreducibility: Finally, like many other program analysis techniques, CODEALIGN requires a reducible flow graph because it relies on natural loop analysis. However, even in C, one of the few programming languages that can form irreducible CFGs, they are extremely rare [Stanier and Watson 2012]. The difficulty with irreducibility comes from identifying the inductive hypothesis to assume true to bootstrap proofs on cyclic dependencies. For natural loops, this can always be done with back-dependencies, as discussed in Section 3.3. Back dependencies do not exist in irreducible control flow. The challenge, then, is to identify a minimal set of lemmas that could be assumed true such that if those lemmas are assumed and the loops are actually equivalent that those loops can be proven equivalent. With this, the rest of CODEALIGN's algorithm could handle irreducible control flow without modification.

4.2 Comparison with a Symbolic-Execution-Based Alignment

We use symbolic execution to build reference alignments against which CODEALIGN can be compared. Symbolic execution provides a reference against a widely-known technique to characterize codealign's behavior, showing the validity of dependency-based equivalence in practice.

The idea behind symbolic execution is to exhaustively test a program by considering all inputs simultaneously. To do this, program inputs are defined as symbolic variables instead of specific (concrete) values. The intermediate

[4]There exist efficient algorithms for determining if planar graphs are isomorphic, but data flow graphs are not necessarily planar. It is easy to write code with a $K_{3,3}$ data flow sub-graph.

values produced as the program executes are represented as mathematical expressions defined in terms of the symbolic input variables. When a symbolic variable is used to decide a branch, symbolic execution may explore both paths.

To build our symbolic-execution alignments, we execute each program we are comparing symbolically, logging the symbolic expressions that represent the values produced each time that instruction is executed (along any path). Using z3 [De Moura and Bjørner 2008], an SMT solver, we then check the equivalence between each pair of instructions between the two pieces of code. Because these symbolic expressions represent the computation of these values on all possible inputs, if the symbolic representations of the execution result are equivalent, the instructions are aligned (as defined in Equation 1).

In symbolic execution, the same instruction may be executed multiple times along different paths, including loop iterations. There may be different values associated with each instruction across different paths, so we also define equivalence for this scenario. In general, there will be ℓ executions of one instruction and k executions of another. Intuitively, each execution of one instruction should have a symbolic value equivalent to an execution of the other. As a practical matter, we terminate symbolic execution after collecting 10,000 instruction executions. If one loop is larger than another, it is possible that the instruction in the smaller loop is executed more times; that is, $\ell \neq k$. To handle this issue, we require that each execution of the instruction executed fewer times must be equivalent to an execution of the other instruction to consider two instructions equivalent. In practice, limits on execution and the need to aggregate values from multiple paths results in unsoundness and incompleteness.

4.2.1 Experiment Methodology. We modify the Klee symbolic execution engine [Cadar et al. 2008] to log the symbolic values of variables after the execution of instructions. Klee is built on top of LLVM, and functions as an LLVM IR interpreter that can represent values symbolically.

We only log symbolic values for the LLVM IR instructions that we can deterministically map to CODEALIGN instructions, and those that do not return memory addresses (e.g. `getelementptr`). Klee uses concrete memory addresses; even identical programs may have different concrete memory addresses on different runs.

We perform experiments on the POJ-104 dataset, a component of CodeXGlue [Lu et al. 2021]. The POJ-104 dataset consists of responses to programming competition questions. It is often used as a type-4 code clone detection benchmark because responses to the same question can, in some senses, be considered equivalent. We choose a clone detection dataset because it is somewhat more likely that responses to the same programming question contain equivalent values that it is that two randomly-selected open-source functions will; that is, the results are more likely to be interesting. However, in this experiment, we are not trying to determine which examples are attempted solutions to the same programming question. We are illustrating how CODEALIGN detects equivalent values inside pairs of functions.

As with many programs from programming challenges, input is read through standard input (typically through the `scanf` or `gets` functions). In the interest of simplifying symbolic constraints, we rename each `main` function, and pass to that function the symbolic input that would have been read from standard input. Then, we remove the call, the associated variables, and associated initialization code from the renamed main function.

Before sampling examples from the dataset, we perform several filtering steps to remove functions that Klee cannot handle or which we cannot preprocess into a form that Klee can handle without substantial alteration of the example. These include functions that use unhandled input/output functions (like C++ input methods `cin` and `cout`, or `scanf` arguments we cannot resolve to a variable); contain floating point variables that Klee automatically sets to 0.0, lack a traditional `main` method, or do not parse. After filtering, 17,106 examples remain of the original 52,000.

We sample 1100 different pairs of functions from the filtered dataset from three categories: (1) 500 pairs of different solutions to the same problem, (2) 500 pairs of identical functions (the same solution to the same problem), and (3) 100 pairs of solutions to different problems, which are expected to have very few, if any, equivalent

instructions. Although the 500 pairs of textually identical functions are trivially equivalent, both CODEALIGN and symbolic execution ignore textual similarity.

We then compile each example to LLVM IR and symbolically execute them, logging the symbolic value produced by executing each instruction in the rewritten `main` functions, until they log 10,000 constraints or reach a timeout of one hour. We then build the alignment by comparing the constraints for each instruction in one function with each instruction in the other, with a timeout of 2 hours. Finally, we map LLVM IR instructions to CODEALIGN instructions and compare the alignments.

4.2.2 Results. Of the experiments, 737/1100 succeed. The errors, by cause, are: (1) 144 have constraints for fewer than five instructions, (2) 69 have more than 200 MB of symbolic constraints, (3) 46 contain floating point instructions despite not containing any `float` variables, (4) 46 contained symbolic variables that are not a part of the functions' parameters (so we could not map them between functions in the pair), (5) 30 failed to compile, (6) 23 timed out after 2 hours, (7) 4 had issues mapping LLVM IR instructions to CODEALIGN instructions, (8) CODEALIGN crashed on 1 example.

	Precision
Self-Alignment	99.9%
Same Problem	95.9%
Different Problem	100.0%

Table 1. agreement between CODEALIGN and symbolic execution.

The results are shown in Table 1. The majority of pairs of instructions between the functions are, as one may expect, nonequivalent. Therefore, we report precision scores. Because we are comparing CODEALIGN against a well-understood approach, we define the symbolic execution alignment as the ground truth for the purposes of calculating precision, though it is unsound and incomplete.

We evaluate over all possible combinations of values. Precision scores are very high meaning that instruction pairs CODEALIGN determined were equivalent had equivalent symbolic values. There are very few exceptions: only 12 of the 737 functions contained any false positives. We manually analyzed all twelve cases and identified three reasons for the differences. The most common reason, affecting 6/12 of the cases, were the use of different integer types (e.g. `int` vs `long`). While these types did indeed hold equivalent values, Klee constraints are modeled with bitvectors, and so different integer types are automatically nonequivalent. The second most common reason, affecting 5/12 cases, is when a mutable data structure is mutated and accessed twice; CODEALIGN considers the accesses the same when they are actually different (See Section 4.1.2). The final case has to do with uninitialized memory. CODEALIGN treats uninitialized memory as a constant and allows it to align with other uninitialized memory; otherwise, a function with uninitialized memory could not align with itself. In the final false-positive case, CODEALIGN aligned two different segments of uninitialized memory.

We do not report recall scores because our symbolic-execution-based alignment performs an overwhelming number of spurious alignments. The culprit is primarily concrete values. In symbolic execution, only the values passed to the function and other values computed based on the input are symbolic; local variables that do not interact with symbolic variables retain concrete values. It is easy for these concrete values to spuriously correlate with other concrete values. For instance, `printf` returns the number of characters that were printed, which may happen to be equal to an unrelated value, such as a loop's iteration counter. The necessarily loose definitions for symbolic multi-execution equivalence defined in Section 4.2 allows these to be marked as equivalent.

4.3 Comparison with Unit-Test-Based Equivalence

In the prior section, we constructed a reference alignment based on symbolic execution to evaluate CODEALIGN's instruction-level equivalence alignments. However, alignments can also be used in a coarser-grained way: for checking function-level equivalence. We say a prediction is *perfectly aligned* with the reference if every value in the prediction is aligned with one in the reference, and vice versa. Being perfectly aligned is a strong indicator of equivalence. (See Section 4.1.2 for limitations.) We compare CODEALIGN with the most widely-used technique

to establish functional equivalence: unit tests. Unit tests are unsound—there may be untested code paths that feature differentiating behavior—but are often reasonable. Here, we measure the correlation between equivalence by perfect alignment and equivalence by unit tests.

In this experiment, we use ExeBench [Armengol-Estapé et al. 2022], a dataset of C functions. Of particular interest is the test set, which contains a test harness and suite of unit tests for each example. We focus on the "real" subpartition rather than the "synth" (synthetic) subpartition. We filter out trivial examples with empty function bodies.

Performing an equivalence check requires comparing two functions. For each function in ExeBench, we compare the output of a neural decompiler—the type of tool CODEALIGN is designed to evaluate—with the original definition provided in the dataset. In particular, we used the CodeT5+-based neural decompiler we developed in Section 5. (We developed two other neural decompilers in Section 5 and all provide almost identical results here.) We compare the prediction and original with both codealign and the ExeBench-provided test suites and measure the correlation between them.

We use the Phi coefficient, which is suitable for paired binary data, and which ranges from -1 to 1. The Phi coefficient is mathematically equivalent to the Pearson correlation coefficient. We find that the correlation between perfect alignment and test-equivalence is 0.52. This is a strong correlation, but indicates some disagreement. We manually analyzed a sample of 50 (prediction, original) pairs on which codealign and exebench disagree. The primary reason, true in 36/50 cases, is due to semantically equivalent code that is syntactically different. For instance, `1<<5` and `0x20` are actually semantically equivalent, but showing this requires a deep understanding of the semantics of C types. We also find four examples of cases where the disagreement is caused by test unsoundness: the functions are not equivalent, but the tests do not explore all code paths and the functions pass all tests. There are five cases where different types hold equivalent values, and one instance where side-effect dependencies impact CODEALIGN. The remaining four cases result from quirks in the testing setup, e.g. a function has the same name as a standard library function.

4.4 Runtime Performance

CODEALIGN is very fast. We quantify CODEALIGN's runtime performance with a benchmarking experiment on C functions in GNU Coreutils version 9.5. We extract all functions from all files the src subdirectory, excluding functions which contain features CODEALIGN does not currently support, such as `#ifndef` macros and `goto` statements. In total, we collected 1176 functions. We then align each function with itself and measure the time it takes to build each equivalence alignment. Aligning a function with itself is the worst case in terms of runtime. (When aligning two completely different functions, there are few lemmas and little induction, leading to early termination.) We used an 18-core Intel Xeon Gold 6240 CPU with 256GB RAM, though the experiment code was single-threaded and processed each function serially.

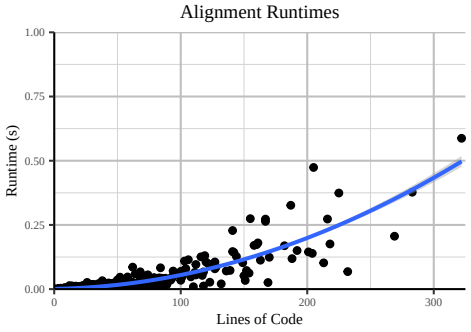


Fig. 4. Runtime performance of CODEALIGN on functions from GNU Coreutils with a quadratic regression trend line.

Fig. 4 displays CODEALIGN's runtime performance. Despite being written in pure python, CODEALIGN can build an alignment in under a second, even on large functions spanning hundreds of lines. This dramatically outperforms, e.g., [Gupta et al. 2018]'s technique for equivalence-checking compiler optimizations, though given

```c
int write_response(int fd, char *response, int len) {
  int retval, byteswritten = 0;
  while (byteswritten < len) {
    retval = write(fd, response + byteswritten, len -
         byteswritten);
    if (retval <= 0) {
      return 0;
    }
    byteswritten += retval;
  }
  return 1;
}
```

(a) **Original** (f): writes a message to a file descriptor, retrying with the remainder of the message if the call to `write` fails to write all of the message.

```c
signed long long
write_response(int a1, long long a2, int a3) {
  int v4, v5, i;
  v4 = a3;
  for (i = 0; i < v4; i += v5) {
    v5 = write(a1, (const void *)(a2 + i),
                    v4 - i);
    if (v5 <= 0)
      return 0LL;
  }
  return 1LL;
}
```

(b) **Decompiled**: Fig. 5a, compiled and determinstically decompiled. This code is much harder to read: variable names are generic (like `a2`), variable types are inaccurate (`char *` is replaced with `long long`), and extra variables are introduced (`v4`).

```c
int write_response(int fd, char *buf, int len) {
    int i;
    for (i = 0; i < len; i += len) {
        if ((i = write(fd, buf + i, len - i)) <= 0)
            return 0;
    }
    return 1;
}
```

(c) **Prediction** (g): A fine-tuned CodeT5's prediction when provided Fig. 5b as input. The prediction is subtlety incorrect.

$$
\begin{aligned}
<_{(4),f} &= <_{(3),g} & \texttt{fd}_f &= \texttt{fd}_g \\
+_{(5),f} &= +_{(4),g} & \texttt{response}_f &= \texttt{buf}_g \\
-_{(6),f} &= -_{(4),g} & \texttt{len}_f &= \texttt{len}_g \\
\texttt{write}_{(5),f} &= \texttt{write}_{(4),g} & \texttt{retval}_f &= \texttt{i}_g \\
<=_{(7),f} &= <=_{(4),g} &&
\end{aligned}
$$

Unaligned: $+_{(8),f}$ and $+_{(3),g}$

(d) *Left*: The partial-loop (Section 3.5) alignment produced between Fig. 5a (f) and Fig. 5c (g), excluding control-flow instructions. *Right*: The variable name mapping derived from the alignment.

Fig. 5. A function (Fig. 5a) which writes a message to a file descriptor with a retry loop. Fig. 5b shows the same function having been compiled, then decompiled. Machine learning models can be used to render decompiler output more readable, but they may produce semantically nonequivalent code. Fig. 5c shows an example. The alignment generated by CODEALIGN in Fig. 5d can be used to both detect the hallucination, and evaluate the quality of the variable names in the decompiled function. Fig. 5d, shoes the use of partial-loop (Section 3.5) alignment to analyze why the loop does not align.

the different hardware and task the comparison is not apples-to-apples.

5 UTILITY: EVALUATING A NEURAL DECOMPILER

Neural decompilation—generating source code from an executable binary or a deterministically derived representation of one—has become an area with rapidly increasing interest [Armengol-Estapé et al. 2024; Cao et al. 2022; Fu et al. 2019; Hu et al. 2024; Jiang et al. 2023; Katz et al. 2018; Liang et al. 2021; Tan et al. 2024; Wu et al. 2022]. Deterministic decompilers fail to recover many of the abstractions, like names and types, that make code readable (Fig. 5a vs Fig. 5b), because those abstractions are discarded during compilation. Neural decompilers can help rewrite code to be more readable, but can also hallucinate, as Fig. 5c shows.

In this section, we show how to use CODEALIGN to quantify the correctness of model predictions. In neural decompilation, the ideal decompilation is identical to the original source. Further, we use the alignment generated by CODEALIGN to map variables in the model's prediction with the ground truth in the original code. This allows

us to evaluate the quality of the generated variable names. Generating these abstractions is a key reason to use neural decompilation in the first place; evaluating their quality is thus an important part of any neural decompilation evaluation.

5.1 Methodology

To demonstrate how CODEALIGN can be used to evaluate neural decompilers, we design a controlled experiment between three different neural decompilers. Because language models are increasingly popular for neural decompilation (in addition to many other tasks), we fine-tune three language models: two sizes of CodeT5 [Wang et al. 2021] (60 and 220 million parameters) and one size of CodeT5+ [Wang et al. 2023] (220 million parameters; CodeT5+ lacks a 60m size). We finetune all three on the same training data to perform a neural decompilation task.

For training data, we used a cross-optimization labeled dataset we previously developed for training neural decompilers [Dramko et al. 2025]. The dataset consists of input/output pairs; the models learn to predict the output given the input. In a neural decompilation task, the input must be a representation of the executable binary that can be recovered entirely deterministically. Here, we use the the industry-standard Hex-Rays (deterministic) decompiler[5] to produce the input. The output is the original source code written by developers. The training set features 106,238 functions each at four optimization levels, for a total of 424,952 functions. We train each neural decompiler on this aggregate training set to expose the neural decompilers to a variety of optimization levels. We evaluate on 1699 functions in the test set at each optimization level; this number is exclusive of 49 examples we filtered out which use compiler extensions that CODEALIGN does not support. Finally, we generate predictions for each models on each test set example, and evaluate those predictions with CODEALIGN and several existing evaluation techniques.

5.2 Results

The results are shown in Table 2. Table 2a shows a CODEALIGN-based evaluation of the two models. Recall from Section 4.3 that a prediction is *perfectly aligned* with the reference if every value in the prediction is aligned with one in the reference, and vice versa; being perfectly aligned is a strong indicator of correctness. CODEALIGN shows that the models are perfectly aligned a small minority of the time, around 14-17%. We also measure the average percentage of instructions that align between the predicted and original code, which ranges from around 21-28%. We see that the models' perform worse at higher levels of optimization, as may be intuitively expected since optimizations make the neural decompilation task more difficult. We also see that the larger 220-million parameter models perform better than the 60-million parameter model; this is also expected, because larger models generally perform better, with steeply diminishing performance returns as model size increases. This is in line with the findings of the CodeT5 authors [Wang et al. 2021], who show that performance as model size increases at most a few percentage points when scaling the model up from 60 to 220 million parameters across various tasks. We also see that the newer CodeT5+ performs better than the older CodeT5 at higher levels of optimization; at lower levels, they're about the same.

Neural decompilers often make subtle mistakes that can be difficult to detect. CODEALIGN can help detect them. Fig. 5c shows an incorrect prediction based on Fig. 5b. If the call to `write` succeeds in writing the whole message at once, then the prediction works in the same way as the original in Fig 5a. However, if the call to `write` writes fewer bytes than expected, the prediction will silently fail to write the rest of the message while returning `1` to indicate success. Accordingly, Fig. 5a and Fig. 5c do not align. When using partial-loop mode (Section 3.5), the alignment shown in Fig. 5d reflects the fact that the instructions executed on the first pass through loop are correct, but that the `+` instruction that calculates the offset in the buffer for the retry is not correct.

[5] https://hex-rays.com/decompiler/

(a) Alignment Results

		Functional Correctness			*Variable Name Quality*	
	Base Model	Perfectly Aligned	% Aligned	Invalid	Accuracy	VarCLR
O0	CodeT5 60m	16.1	25.8	6.8	13.9	41.3
	CodeT5 220m	17.3	28.2	3.9	15.8	42.5
	CodeT5+ 220m	17.0	28.2	3.1	15.8	42.8
O1	CodeT5 60m	14.6	23.4	6.7	13.2	39.6
	CodeT5 220m	15.5	25.5	2.9	14.5	40.4
	CodeT5+ 220m	15.9	25.9	2.0	14.5	40.8
O2	CodeT5 60m	14.0	22.2	6.9	12.9	38.6
	CodeT5 220m	14.5	24.5	3.0	14.0	39.5
	CodeT5+ 220m	15.6	25.4	2.8	14.4	40.4
O3	CodeT5 60m	13.9	21.7	8.4	12.8	38.6
	CodeT5 220m	14.5	24.2	3.7	13.9	39.3
	CodeT5+ 220m	15.6	25.1	2.6	14.1	40.3

(b) Existing similarity metrics

	Base Model	CodeBLEU [Ren et al. 2020]	CodeBERTScore [Zhou et al. 2023]	CrystalBLEU [Eghbali and Pradel 2022]	Corpus BLEU [Papineni et al. 2002]
O0	Codet5 60m	55.2	84.1	27.1	39.3
	Codet5 220m	57.3	84.3	29.2	42.1
	Codet5+ 220m	57.6	84.2	28.8	42.5
O1	Codet5 60m	52.9	83.6	26.1	37.5
	Codet5 220m	55.0	83.9	27.5	40.1
	Codet5+ 220m	55.1	83.8	27.8	41.0
O2	Codet5 60m	51.7	83.2	24.7	35.9
	Codet5 220m	54.5	83.6	26.6	39.0
	Codet5+ 220m	55.0	83.6	26.7	39.5
O3	Codet5 60m	51.5	83.0	24.7	35.7
	Codet5 220m	54.3	83.5	26.2	38.7
	Codet5+ 220m	54.9	83.5	26.3	39.1

Table 2. CODEALIGN and several code similarity metrics used to evaluate two neural decompilers. Results exclude CODEALIGN failures, which occur in 2-3% of cases. In Table 2a, % Aligned refers to the average percentage of instructions that aligned in each predicted/original function pair. Inlining is disabled at all levels of optimization. VarCLR was introduced by [Chen et al. 2022b].

Alignments generated by CODEALIGN can be further used to evaluate the quality of variable names produced. If two aligned instructions assign their results to variables in the code, we record those pairs of variables. For instance, Figure 5, the variable `i` from Figure 5a and `retval` from Figure 5c align because they store the results of the aligned `write` instructions. Not all instructions are associated with a variable; for instance, in Figure 5a,

the result of the expression `retval <= 0` is directly used by the `if` statement; it isn't stored in a variable. With the variables aligned, we can score them using various metrics. Here, we use exact-match accuracy and the variable name similarity metric VarCLR [Chen et al. 2022b]. The scores in Table 2a also include parameter names, matched positionally. The variable alignment for Fig. 5c is shown in Fig. 5d. Two of the variable names are exactly correct, but `buf` is more generic than `response` and `i` is a poor name for `retval`. Only CODEALIGN can enable this type of variable name evaluation.

Table 2b shows the results of several popular similarity metrics used to evaluate the same predictions by the same models. The metrics agree that CodeT5+ (220m) usually performed better than CodeT5-220m, and that the larger models perform better than the smaller ones. However, it's not clear by how much, and it's not clear how well each model performed in isolation in a way directly interpretable to humans. These metrics are relative, rather than absolute, and offer substantially less detail. Therefore, using these metrics allows weaker claims to be made about the performance of the models.

For instance Fig. 5a and Fig. 5c have a CodeBERTScore of 0.860. It is difficult to know how good of a score this is, or to even understand why the score is what it is. It is not the necessarily the case that an example given a higher score is better. For example, these functions:

```c
int main(void) {
    init();
    return auth() != 0;
}
```

```c
int main(int argc, char *argv[]) {
    init();
    return auth() != 0;
}
```

are equivalent in C, but CodeBERTScore gives them a slightly lower score: 0.848. CODEALIGN aligns them perfectly.

6 CASE STUDY

In this section, we demonstrate how CODEALIGN applies to a real neural decompilation target—malware. CODEALIGN is able to identify neural decompilers' mis-generations relative to the original source. The real original source code for most malware is generally unavailable, carefully kept secret by the malicious actors who write the malware—hence the need for decompilers in the first place. The malware sample in our case study is from the well-known and widespread Mirai-family botnet malware [Shapiro 2023], the source code of which was released publicly and anonymously by the authors in an unsuccessful bid to throw off law enforcement and maintain plausible deniability in the event of their arrest. Since its release, numerous Mirai variants have appeared as other hackers have modified the source code for their own purposes.

Figure 6 shows an example of a function from this malware. Figure 6a shows the original code written by the malware authors. The function removes leading and trailing whitespace from a string. In the context of the malware, it processes commands from the botnet's command-and-control server, potentially to perform a malicious action like a Distributed Denial-of-Service (DDoS) attack. This function is called from the main control loop of the bot, part of the pipeline for receiving, processing, and running commands.

A security analyst attempting to reverse engineer this malware and assess the threat may attempt to determine how those commands are processed. However, the deterministically decompiled form of this function (Figure 6b) is much harder to understand than the original source code, because it's missing many of the details that make code readable in the first place. To avoid painstaking manual analysis, the analyst may choose use a neural decompiler like those we trained in Section 5. Figure 6c and Figure 6d show the predictions of our 220m CodeT5 and CodeT5+-based neural decompilers, respectively.

The neural decompilers make some notable substantial improvements to the code. Names for several important identifiers are added, including the function names, variable names, and the name of the `enum` value used, which offer important insights as to what the function does. The neural decompilers also correct for the misleading return behavior present in the decompiled code: the decompiled code returns a value, in particular, a pointer to the string's null-byte, whereas the original code does not. (The null-byte-pointer happened to be in the `rax` register,

```
void trim(char *str) {
  int i;
  int begin = 0;
  int end = strlen(str) - 1;
  while (((*__ctype_b_loc ())[(int) ((str[begin]))] & (unsigned short) _ISspace)) begin++;
  while ((end >= begin) && ((*__ctype_b_loc ())[(int) ((str[end]))] & (unsigned short) _ISspace)) end--;
  for (i = begin; i <= end; i++) str[i - begin] = str[i];
  str[i - begin] = '\0';
}
```

(a) A function which removes leading and trailing whitespace from a string. In the context of the malware, the string is a command for infected computers on the botnet, and is called from the malware's main control loop. The original code is shown after macros have been expanded.

```
const char *__fastcall func22(const char *a1) {
  const char *result;
  int v2;
  int v3;
  int i;

  v3 = 0;
  v2 = strlen(a1) - 1;
  while ( ((*__ctype_b_loc())[a1[v3]] & 0x2000) != 0 ) ++v3;
  while ( v2 >= v3 && ((*__ctype_b_loc())[a1[v2]] & 0x2000) != 0 ) --v2;
  for ( i = v3; i <= v2; ++i ) a1[i - v3] = a1[i];
  result = &a1[i - v3];
  *result = 0;
  return result;
}
```

(b) Figure 6a, having been compiled, then decompiled. The decompiled code is missing meaningful identifier names, including the function's name, variable names, and the name of the **enum** value **_ISspace**. These identifiers offer key clues as to the purpose of the function. Further, the decompiled function has misleading return behavior, a byproduct of the conservative assumptions that decompilers often make.

```
void process_line(char line[]) {
  int i = 0;
  int j = strlen(line) - 1;
  while (((*__ctype_b_loc ())[(int) ((line[i]))] & (unsigned short) _ISspace)) { ++i; }
  while (j >= i && ((*__ctype_b_loc ())[(int) ((line[j]))] & (unsigned short) _ISspace)) { --j; }
  for (i = i; i <= j; ++i) { line[i-i] = line[i]; }
  line[i-i] = '\0';
}
```

(c) CodeT5's (220m) prediction. At first glance, the prediction looks like it could be correct, but it is not. CodeT5 used the same variable name (**i**) to represent the start of the non-space segment of the string, which should be held constant throughout the **for** loop, and the loop counter, which is updated on each iteration. CODEALIGN detects this issue; the **i - i** instructions and all instructions downstream in the dataflow fail to align.

```
void str_rtrim(char *s) {
  int i; int j = 0; int l = strlen(s)-1;
  while(((*__ctype_b_loc ())[(int) ((s[j]))] & (unsigned short) _ISspace)) { j++; }
  while(l>=j && ((*__ctype_b_loc ())[(int) ((s[l]))] & (unsigned short) _ISspace)) { l--; }
  for(i=j;i<=l;i++) { s[i-j]=s[i]; }
  s[i-j]='\0';
}
```

(d) CodeT5+'s (220m) prediction. The prediction correctly represents the semantics of the original (Figure 6a); CODEALIGN determines that this prediction and Figure 6a perfectly align.

Fig. 6. A function from an instance of Mirai-family malware [Shapiro 2023], its decompiled form, and two different neural decompilations: one correct, and one incorrect. CODEALIGN is able to identify the subtle incorrectness in Figure 6c and show that Figure 6d is correct.

which is often used to hold the return value in a function call. However, it can also be used as a general-purpose register, as it was here; the deterministic decompiler conservatively assumes that that value should be returned.)

However, neural decompilers can also make mistakes, undermining the validity of the code transformation. Indeed, while not immediately obvious, the CodeT5-based neural decompiler makes the same mistake twice, on lines 6 and 7 of Figure 6c. The `for`-loop on line 6 copies the non-space characters in the string to the beginning of the string. However, the variable `i` is used for two purposes: recording the start of the non-space segment of the string and controlling iteration through the string. The variable can't simultaneously be used in both conflicting roles. CODEALIGN is able to detect this mistake when comparing it with the original code. The `i - i` instructions on lines 6 and 7 in Figure 6c fail to align with the `i - begin` instructions on lines 7 and 8 in Figure 6a, respectively. Downstream instructions dependent on those subtraction instructions fail to align. CODEALIGN not only detects the mistakes but also where the errors occur (the points at which alignment fails).

In contrast, the CodeT5+-based neural decompiler does successfully predict the original code. CODEALIGN determines that Figure 6a and Figure 6d perfectly align.

In theory, CODEALIGN can be used to compare neurally decompiled code with the corresponding decompiled code as an alternative reference to the original code (e.g. Figure 6b vs Figure 6c), though it is generally unsuitable for this. Doing so is desirable because the original source code is not usually available for decompilation targets—and if it were, decompilation would not be necessary. However, decompiled code offers substantial systematic syntactic differences from original source code [Dramko et al. 2024] that make such a comparison nontrivial. For instance, in Figure 6b, the name of the `enum` value `_ISspace` is replaced by the value itself, `0x2000`. CODEALIGN has no way of knowing that `_ISspace` and `0x2000` actually refer to the same value.

However, CODEALIGN still offers substantial utility. CODEALIGN can be used to evaluate neural decompilers' abilities to produce code in situations where the original source code is available, as we have done in this case study and in Section 5. The resulting knowledge of how well neural decompilers perform in these scenarios can be used to calibrate expectations about how well they will perform in practice—and allow reverse engineers to determine how much each model can be trusted. All existing evaluation techniques [Eghbali and Pradel 2022; Papineni et al. 2002; Ren et al. 2020; Zhou et al. 2023] are also only suitable for comparing the original and predicted code.

7 DISCUSSION

CODEALIGN can be used to evaluate code generated as part of other machine learning tasks. It works best when the generated code and reference code compute the result using the same algorithm. Other machine learning tasks that have this property include transpilation and automated refactoring. CODEALIGN will struggle in contexts where substantially different but semantically equivalent algorithms are acceptable. Of course, this is in general undecidable.

CODEALIGN is also useful for tasks besides evaluating ML-generated code, like code clone detection (the parts of functions that align are clones). Or, consider *patch generation*, a critical part of automated program repair that creates code fragments to fix a bug in a piece of software. An important subproblem in template-based patch generation is determining which variables should be instantiated inside a patch [Afzal et al. 2019; Liu et al. 2019]. An equivalence alignment can map the variables in the buggy region to those in a candidate patch in a similar way to how we mapped variable names to each other in Fig. 5d. Third, CODEALIGN may also be useful for plagiarism detection. Plagiarized code is by definition identical or very similar to the original. CODEALIGN is robust to low-effort attempts to disguise plagiarism, such as variable renaming and (inconsequential) statement reordering, and the the equivalence alignment can be used as evidence.

8 RELATED WORK

The most directly analagous piece of work to CODEALIGN is that of [Yang et al. 1989]. They also operate on program dependence graphs and SSA-form code to determine if two functions are equivalent. However, their technique is purely theoretical and handles an abstract, academic, feature-restricted language with only scalar variables and constants, and only if and while statements for control flow. The last limitation means that instructions must have at most one control dependency. Further, ϕ nodes must be definitively associated with either an 'if' or 'while' statement, limiting the complexity of the control flow to which their approach can scale. In contrast, CODEALIGN has an implementation, can handle real C (CODEALIGN doesn't support `goto`s but theoretically could less irreducability) and an arbitrary number of control dependencies.

Another related line of work is *semantic differencing*. These tools output a *functional* difference between two functions, such as a counterexample generated by an SMT solver. Differential symbolic execution [Person et al. 2008] identifies textual or AST-based differences in code, symbolically executes them, and queries the SMT solver to check their equivalence. ArrDiff [Badihi et al. 2020] builds on this work by extracting information from unchanged code blocks that may nonetheless be useful for showing the diffs equivalent. Symbolic-execution-based approaches are useful, but are unsound with respect to loops, are expensive to run, and must usually compile the input programs, which renders them useless for tasks like evaluating neural decompilers. In contrast, SymDiff [Lahiri et al. 2012] translates input functions into Boogie [Barnett et al. 2006], a verification language, and creates logical formulas summarizing the effects of the functions, then uses an SMT solver to check their equivalence. This approach cannot fully handle loops; they must be unrolled to a specified depth or translated to tail-recursive functions, the latter of which are checked for equivalence separately.

Decompiler testing finds bugs in decompilers. While a different task, some of the involved techniques are applicable to evaluating neural decompilers. D-Helix [Zou et al. 2024] finds bugs by comparing the input binary and the output decompiled code by attempting to re-compile the decompiled code, symbolically execute it and the corresponding function in the binary, and show the equivalence of the results with an SMT solver. Re-compiling decompiled code is generally difficult; D-Helix attempts to iteratively fix compilation errors with heuristics, which works up to 72.4% of the time. For evaluating the semantic correctness of neural decompilers, it would be necessary to instead compile the predicted code, which involves its own unique challenges, especially generating appropriate definitions for unknown types. D-Helix inherits the advantages and disadvantages of symbolic execution.

Code clone detection [Rattan et al. 2013; Roy et al. 2009; Zakeri-Nasrabadi et al. 2023] tries to find functionally identical pieces of code so they can be refactored into a single entity for easier maintainability. CODEALIGN can be used for code clone detection as discussed in Section 7. Code clone techniques vary significantly based on the type of clones they attempt to target; CODEALIGN bears the strongest resemblance to program-dependence-graph (PDG)-based techniques [Saha et al. 2013]. Isomorphic PDGs are code clones. Unfortunately, finding maximal isomorphic subgraphs is NP-complete, so these techniques find approximations rather than true isomorphisms. This renders these techniques unable to provide an equivalence alignment as CODEALIGN does and are unsound.

Code similarity metrics [Eghbali and Pradel 2022; Papineni et al. 2002; Ren et al. 2020; Tran et al. 2019; Zhou et al. 2023] are used to evaluate code generated by machine learning models. These are intended to measure how well a prediction matches the reference. While the goal is to detect equivalent programs, these methods are necessarily heuristic and justified based on agreement with subjective human scores. Unlike CODEALIGN, these measures offer a unitless similarity score which is meant to be compared with other values from the same metric.

Another related area is the formal, sound equivalence checking used to validate the correctness of optimizations produced by compilers. A common abstraction used in this area is the *simulation relation*. A simulation relation is defined at program points, rather than for pairs of values, and contains symbolic relationships between live variables at those points. The definition of a simulation relation does not provide a method for constructing the

relation, and unlike an equivalence alignment, makes no suggestion as to what kinds relationships should be found at each program point. Many translation validations techniques use compiler instrumentation to populate the simulation relation [Necula 2000; Sewell et al. 2013; Stepp et al. 2011]; this is not possible in neural decompilers because of their nature as machine learning models. Likewise, approaches that use execution traces [Churchill et al. 2019] are not applicable to evaluating neural decompilers because decompiled code cannot, in general, be compiled. The most directly applicable approaches are static, black-box translation validatiors, which makes no assumptions about the nature of the optimization performed. In particular, work based on Joint Transfer Function Graphs (JTFG), are the most analagous [Dahiya and Bansal 2017; Gupta et al. 2020, 2018], though in their current form, they require compilation (not execution). These are simulation relations that bundle control flow with nonbranching code and attempt to match branches in the optimized and unoptimized with each other, usually with a heuristic guess-and-check strategy. In contrast, CODEALIGN does not need heuristics to build an alignment; in particular, control flow alignment is efficient due to ordering. JTFG approaches have substantial overhead, and take on the order of tens to hundreds of seconds per example even for very short functions of fewer than two dozen lines of code [Gupta et al. 2018]. CODEALIGN is much faster.

9 CONCLUSION

In this work, we introduce the idea of an equivalence alignment, a relation between equivalent instructions in two functions. We present a tool, CODEALIGN, which builds equivalence alignments. Using CODEALIGN, we demonstrate how it can be used to evaluate code from a neural decompiler by identifying how often the model's predictions are equivalent to that of the reference. We further show how an equivalence alignment can be used to determine which variable names in the decompiled code map to those in the reference, allowing us to evaluate variable name quality as well. We discuss how the equivalence alignment abstraction has applications to a variety of different tasks, including program repair, code clone detection, and plagiarism detection. CODEALIGN can be found at https://github.com/squaresLab/codealign, and a replication package for the experiments at https://github.com/squaresLab/codealigneval.

ACKNOWLEDGMENTS

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE2140739. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work was funded in part by Microsoft.

REFERENCES

Afsoon Afzal, Manish Motwani, Kathryn T Stolee, Yuriy Brun, and Claire Le Goues. 2019. SOSRepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2162–2181.

Bowen Alpern, Mark N Wegman, and F Kenneth Zadeck. 1988. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1–11.

Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael F. P. O'Boyle. 2022. ExeBench: an ML-scale dataset of executable C functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 50–59. https://doi.org/10.1145/3520312.3534867

Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael FP O'Boyle. 2024. SLaDe: A Portable Small Language Model Decompiler for Optimized Assembly. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 67–80.

Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on*

the foundations of software engineering. 13–24.

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2006. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*. Springer, 364–387.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational verification using product programs. In *International Symposium on Formal Methods*. Springer, 200–214.

Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

Ying Cao, Ruigang Liang, Kai Chen, and Peiwei Hu. 2022. Boosting neural networks to decompile optimized binaries. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 508–518.

Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022a. Augmenting Decompiler Output with Learned Variable Names and Types. In *USENIX Security Symposium*. 4327–4343.

Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. 2022b. Varclr: Variable semantic representation pre-training via contrastive learning. In *Proceedings of the 44th International Conference on Software Engineering*. 2327–2339.

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1027–1040. https://doi.org/10.1145/3314221.3314596

Manjeet Dahiya and Sorav Bansal. 2017. Black-box equivalence checking across compiler optimizations. In *Programming Languages and Systems: 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings 15*. Springer, 127–147.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Luke Dramko, Claire Le Goues, and Edward J Schwartz. 2025. Idioms: Neural Decompilation With Joint Code and Type Prediction. *arXiv preprint arXiv:2502.04536* (2025).

Luke Dramko, Jeremy Lacomis, Edward J Schwartz, Bogdan Vasilescu, and Claire Le Goues. 2024. A Taxonomy of C Decompiler Fidelity Issues. (2024).

Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

Michael James Van Emmerik. 2007. *Single Static Assignment for Decompilation*. Ph.D. Dissertation. The University of Queensland School of Information Technology and Electrical Engineering. http://vanemmerikfamily.com/mike/master.pdf

Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An End-to-End Neural Program Decompiler. In *Conference on Neural Information Processing Systems*.

Martin Grohe and Pascal Schweitzer. 2020. The graph isomorphism problem. *Commun. ACM* 63, 11 (oct 2020), 128–134. https://doi.org/10.1145/3372123

Shubhani Gupta, Abhishek Rose, and Sorav Bansal. 2020. Counterexample-guided correlation algorithm for translation validation. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.

Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. 2018. Effective use of SMT solvers for program equivalence checking through invariant-sketching and query-decomposition. In *Theory and Applications of Satisfiability Testing–SAT 2018: 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings 21*. Springer, 365–382.

Peiwei Hu, Ruigang Liang, and Kai Chen. 2024. DeGPT: Optimizing Decompiler Output with LLM. (2024).

Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2023. Nova+: Generative Language Models for Binaries. *arXiv preprint arXiv:2311.13721* (2023).

Deborah S. Katz, Jason Ruchti, and Eric Schulte. 2018. Using recurrent neural networks for decompilation. In *IEEE International Conference on Software Analysis, Evolution and Reengineering*. 346–356. https://doi.org/10.1109/SANER.2018.8330222

Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A neural approach to decompiled identifier naming. In *IEEE/ACM International Conference on Automated Software Engineering*. 628–639. https://doi.org/10.1109/ASE.2019.00064

Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24*. Springer, 712–717.

Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. 2021. Neutron: an attention-based neural decompiler. *Cybersecurity* 4 (2021), 1–13.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 31–42.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun

Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

George C Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 83–94.

Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, Adam Doupé, Chitta Baral, and Ruoyu Wang. 2024. "Len or index or count, anything but v1": Predicting Variable Names in Decompilation Output with Transfer Learning. In *IEEE Symposium on Security and Privacy*. https://github.com/sefcom/VarBERT

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Păsăreanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 226–237.

Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199. https://doi.org/10.1016/j.infsof.2013.01.008

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 12–27.

Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495. https://doi.org/10.1016/j.scico.2009.02.007

Ripon K Saha, Chanchal K Roy, Kevin A Schneider, and Dewayne E Perry. 2013. Understanding the evolution of type-3 clones: an exploratory study. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 139–148.

Thomas Arthur Leck Sewell, Magnus O Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 471–482.

Scott J Shapiro. 2023. The strange story of the teens behind the Mirai Botnet. *IEEE Spectrum* 23 (2023).

James Stanier and Des Watson. 2012. A study of irreducibility in C programs. *Software: Practice and Experience* 42, 1 (2012), 117–130.

Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-based translation validator for LLVM. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 737–742.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling Binary Code with Large Language Models. *arXiv preprint arXiv:2403.05286* (2024).

Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. 2019. Does BLEU score work for code migration?. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 165–176.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 1069–1088.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

Ruoyu Wu, Taegyu Kim, Dave Jing Tian, Antonio Bianchi, and Dongyan Xu. 2022. {DnD}: A {Cross-Architecture} deep neural network decompiler. In *31st USENIX Security Symposium (USENIX Security 22)*. 2135–2152.

Jiaqi Xiong, Guoqiang Chen, Kejiang Chen, Han Gao, Shaoyin Cheng, and Weiming Zhang. 2023. HexT5: Unified Pre-Training for Stripped Binary Code Information Inference. In *IEEE/ACM International Conference on Automated Software Engineering*. 774–786. https://doi.org/10.1109/ASE56229.2023.00099

Wuu Yang, Susan Horwitz, and Thomas Reps. 1989. *Detecting program components with equivalent behaviors.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. 2023. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software* (2023), 111796.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 13921–13937.

Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave Jing Tian. 2024. D-Helix: A Generic Decompiler Testing Framework Using Symbolic Differentiation. In *33rd USENIX Security Symposium (USENIX Security 24)*. 397–414.

Received 17 February 2025; revised 26 August 2025; accepted 4 October 2025