Meaningful Variable Names for Decompiled Code: A Machine Translation Approach

Alan Jaffe
Carnegie Mellon University
apjaffe@andrew.cmu.edu

Jeremy Lacomis
Carnegie Mellon University
jlacomis@cmu.edu

Edward J. Schwartz
Carnegie Mellon University
Software Engineering Institute
eschwartz@cert.org

Claire Le Goues
Carnegie Mellon University
clegoues@cs.cmu.edu

Bogdan Vasilescu
Carnegie Mellon University
vasilescu@cmu.edu

ABSTRACT

When code is compiled, information is lost, including some of the structure of the original source code as well as local identifier names. Existing decompilers can reconstruct much of the original source code, but typically use meaningless placeholder variables for identifier names. Using variable names which are more natural in the given context can make the code much easier to interpret, despite the fact that variable names have no effect on the execution of the program. In theory, it is impossible to recover the original identifier names since that information has been lost. However, most code is *natural*: it is highly repetitive and predictable based on the context. In this paper we propose a technique that assigns variables meaningful names by taking advantage of this naturalness property. We consider decompiler output to be a noisy distortion of the original source code, where the original source code is transformed into the decompiler output. Using this noisy channel model, we apply standard statistical machine translation approaches to choose natural identifiers, combining a translation model trained on a parallel corpus with a language model trained on unmodified C code. We generate a large parallel corpus from 1.2 TB of C source code obtained from GitHub. Under the most conservative assumptions, our technique is still able to recover the original variable names up to 16.2% of the time, which represents a lower bound for performance.

ACM Reference Format:
Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. 2018. Meaningful Variable Names for Decompiled Code: A Machine Translation Approach. In *Proceedings of ICPC '18: 26th IEEE/ACM International Conference on Program Comprehension, May 27–28 2018, Gothenberg, Sweden (ICPC'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196321.3196330

1 INTRODUCTION

Developers expend a great deal of effort and consideration to select meaningful variable names, and for good reason. It has been shown that well-selected variable names make it *significantly* easier to understand code [1, 2]. Identifier names provide context to abstractions in high-level programming languages such as functions, loops, and classes, which allows developers to understand the function of these constructs easier. However, despite the human effort that goes into making source code readable, *e.g.*, by choosing meaningful identifier names, much of this readability is lost during compilation: as high-level abstractions are transformed to low-level sequences of instructions by a compiler, both the structure of the code and the carefully-chosen identifiers are lost.

The loss of variable names during compilation is typically not a concern when the original source is available, but this is not always the case. Commercial software vendors and malware authors alike often distribute their software in executable form without including the original source code. As a result, a class of analysts known as reverse engineers specialize in reading and understanding a program's behavior from its executable to analyze malware [3–5], discover software vulnerabilities [3, 6, 7], or patch bugs in legacy software [6, 7]. Historically, reverse engineers were often forced to "read" executable programs at the assembly code level. More recently, reverse engineers have been using *decompilers*, which attempt to reverse the compilation process by recovering information about the original program's variables, types, functions, and control flow structure, representing this information in a source code language such as C.

It is generally accepted that reverse engineers understand decompiler output more readily than they do assembly code [4, 6, 7]. Some modern decompilers are even explicitly designed to produce readable and understandable code [4]. However, significant readability challenges remain. First, decompilers produce code that is largely *not* idiomatic of what humans would produce. Decompilers often transform code originally written using one abstraction into a different, but semantically identical abstraction. The result is often not as natural to humans (*e.g.*, struct member references like `x.e` may be transformed to array accesses of the corresponding member offset `x[4]`). Second, current decompilers make no attempt to recover or suggest meaningful identifier names; instead, they assign generic variable names, like `v1` and `v2`. We illustrate these challenges with the example in Figure 1. Comparing the original source code for function `xmlErrMsgStr` (Figure 1a) with the HexRays[1] decompiler output (Figure 1b), note how the names for variables

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
ICPC'18, $15.00

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196330

[1] https://www.hex-rays.com

```
void xmlErrMsgStr(xmlCtxt ctxt, xmlErrs error,
   const char *msg, const xmlChar *val) {
  if ((ctxt != 0) && (ctxt->instate == -1))
    return;

  if (ctxt != 0)
    ctxt->errNo = error;

   _raiseError(ctxt, error, msg, val);
}
```

(a) Original source code.

```
int xmlErrMsgStr(uint32 *a1, int a2,
   const char *a3, int a4) {
  int v5;
  if ((!a1) || (v5 = a1[43], v5 != -1)) {
    if (a1)
      a1[21] = a2;
    v5 =
      _raiseError(a1, a2, a3, a4);
  }
  return v5;
}
```

(b) Decompiled code, with uninformative variable names.

```
int xmlErrMsgStr(uint32 *ctx, int error,
   const char *msg, int val) {
  int status;
  if ((!ctx ) || (status=ctx[43], status!=-1)) {
    if (ctx)
      ctx[21] = error;
    status =
      _raiseError(ctx, error, msg, val);
  }
  return status;
}
```

(c) Translated code with more natural variable names.

Figure 1: Illustrative example (simplified for presentation). Variables highlighted are lost during compilation, but recovered by our system. Even for v5, highlighted in red, an extra variable introduced during (de)compilation, our technique can suggest a more natural name (status).

ctxt, error, msg, and val, highlighted in yellow, are lost during compilation, becoming a1 through a4.

In this paper, we show that it is possible to recover variable names in decompiled source code *that naturally fit a particular context.* Although it may seem like recovering meaningful identifier names in decompiled source code is impossible since the original names are "lost" during the compilation process, recent work [8–12] has shown that because code is *natural* [13], *i.e.*, highly repetitive and predictable based on context, it is possible to assign natural names to identifiers in programs by learning names that developers have assigned to code used in similar contexts. Using a similar *naturalness* approach, for the example in Figure 1 our technique produces the names error, msg, val, and ctx (Figure 1c), reflecting either exactly (the first three) or approximately (the forth) the original programmer's intent.

More specifically, our work builds on a line of recent *naturalness*-based tools [9, 10, 12] to recover meaningful variable names in JavaScript code that has been intentionally mangled by obfuscators. Although decompiling a program is not a form of program obfuscation, the resulting code is similar: in both cases, the mangled program is stripped of its original variable names, but it is structurally and semantically similar to the original. The natural question we address in this paper, therefore, is whether similar techniques to recover meaningful variable names can be transferred to the domain of decompilation.

Inspired by JSNaughty [10], we cast the problem of recovering meaningful variable names in decompiled code as an instance of the "noisy channel" model used in natural language translation (*e.g.*, French to English). Consequently, we also base our solution on statistical machine translation (SMT). SMT is data-driven, using statistical models of language translation estimated from large, parallel (*i.e.*, sentence-aligned) corpora of text in the source and target languages. The key to the successful application of statistical approaches such as SMT to automated identifier renaming is the ability to generate arbitrary amounts of *parallel* training data. For JavaScript [9, 10, 12] this was possible, since one has access to the obfuscator (*e.g.*, UglifyJS[2]) and starting, say, from open-source code, could produce as many training examples as needed.

We leverage a similar insight here: arbitrary amounts of open-source (C) code can be compiled and decompiled, as needed. However, while the JavaScript minification that existing tools deal with is a simple α-renaming of the original code, rendering the construction of a line-by-line aligned training corpus trivial, our scenario is more challenging. Indeed, the process of compiling and decompiling does *not* produce an α-renaming of the original code, which makes constructing the parallel training corpus a challenge. For example, as illustrated in Figure 1b, decompilers may generate non-idiomatic code, as well as extra variables (v5) that do not have a correspondent in the original source.

Our contributions in this paper are twofold: (1) we show that it is possible to automatically generate an aligned parallel corpus between natural C code and decompiled C code, using simple alignment heuristics; (2) we train and evaluate an SMT model that can suggest natural variable names in decompiled C code, based on the open-source SMT toolkit Moses [14], commonly used in natural language translation. This demonstrates that SMT techniques can be used for information recovery in source code even when the difference between the original and the transformed source code is more complex than simple α-renaming of variable names.

The rest of this paper is structured as follows. In Section 2 we provide background on SMT and decompilation, focusing on the challenges to using SMT to rename variables in this new context. We outline our approach in Section 3. Section 4 describes the experiments we conducted to validate our approach, including the accuracy of our novel alignment technique; the accuracy of translation overall; and the impact of including additional information in the translation process on renaming results. Section 5 places our contribution in context with respect to related and prior work. We conclude in Section 6.

2 BACKGROUND

Our technique uses SMT, or statistical machine translation [15], to assign meaningful names to variables in decompiled C code. This section provides background on SMT (Section 2.1) and decompilation (Section 2.2) respectively, focusing on the particular challenges that apply in our domain.

[2] https://github.com/mishoo/UglifyJS

2.1 Statistical Machine Translation

SMT is a technique that translates between two languages by estimating statistical models from a large, aligned, bilingual corpus. SMT was originally developed to translate between natural languages, but it has since also been adapted to the transformation of programming languages. For example, attempts have been made to use SMT to translate between C# and Java [16, 17], generate pseudocode from source code [18], improve code completion tools [19], and reverse the obfuscation of JavaScript programs [10].

In SMT, to translate, *e.g.*, a French sentence f into an English sentence e, one learns a probability distribution $p(e|f)$ from an aligned parallel corpus, and tries to find the most likely translation by determining the sentence e that maximizes the value of $p(e|f)$. In a similar way, we can view a line of code e with natural variable names as a translation of a line of decompiled code with uninformative variables f, and use SMT to determine the e that maximizes $p(e|f)$.

SMT is based on the noisy channel model, where each phrase in the source language f is assumed to be a distortion of a phrase in the target language e (*e.g.*, compiling and decompiling the program). The model does not explicitly specify the reverse transformation from f to e, so one cannot directly calculate and maximize $p(e|f)$. Instead, using the Bayes theorem, one estimates:

$$\begin{aligned}\underset{e}{\operatorname{argmax}}\ p(e|f) &= \underset{e}{\operatorname{argmax}}\ \frac{p(f|e)p(e)}{p(f)}\\ &= \underset{e}{\operatorname{argmax}}\ p(f|e)p(e) \qquad \text{(for a specific } f\text{)}\end{aligned}$$

This formulation is common in SMT. The two parts of this equation are known as the *language model*, $p(e)$ and the *translation model*, $p(f|e)$. In our case, we estimate an n-gram language model, capturing the probability of n-grams occurring in natural C code (*i.e.*, before compilation and decompilation), from a corpus of such code. The translation model captures the probability that different "phrases" (sequences of tokens, not necessarily consecutive, within each line) in decompiled code are "translations" of the natural C code it was produced from, which can be estimated from a line-by-line-aligned parallel corpus. However, unlike prior work on JavaScript [10], generating the latter corpus is challenging, since compilation and decompilation can transform the structure of the code and even introduce new variables.

2.2 Decompilation and SMT Challenges

A *decompiler* is a program that takes a compiled program as input and outputs high-level source code that describes the compiled program [6]. There are decompilers for a wide variety of compiled and source languages, but in this paper we focus on *executable to C* decompilers [3, 4, 6, 7] due to the ubiquity and complexity of executable code. We employ Hex-Rays — a commercial x86 and x86-64 to C decompiler popular among reverse engineers — as an exemplar, but our techniques are not specific to Hex-Rays and should work with any decompiler.

Although C decompilers are generally able to recover some amount of information about functions, variables, types, and control flow structure [6], even state of the art decompilers struggle to produce idiomatic C code. For example, in Figure 1b, Hex-Rays fails to recover the `xmlCtxt` structure type and instead represents it as a pointer to `uint32`. As a result, Hex-Rays crudely translates accesses to the structure (*e.g.*, `ctxt->instate`) into array dereferences (*e.g.*, `a1[43]`) that a human programmer would be unlikely to write.

Unfortunately, as we show in Section 4.3, non-idiomatic decompilation complicates the use of SMT techniques for variable renaming. As discussed above, it is crucial for statistical approaches like SMT to have access to a large, *parallel*, *aligned* training corpus. Yet, C source code is not readily aligned with decompiled code (unlike JavaScript before and after minification), hence special alignment steps are needed. For example, a natural way to produce such a parallel corpus would start from the original (human written) C code, and then α-rename source variables to names similar to those used by decompilers (*e.g.*, `v1`, `v2`, etc.) to create the parallel corpus. However, because human programmers generally write idiomatic C code, this α-renamed corpus, while simple to construct, ultimately contains few examples of how to name variables in *non*-idiomatic contexts. This results in an SMT model that is unable to recover variable names when the decompiler produces non-idiomatic C code (which is quite often).

An alternative approach to construct a parallel corpus could be to leverage debugging symbols when available (*e.g.*, when code was compiled with `gcc -g`). However, this approach is potentially unrealistic. Indeed, debug symbols are rarely available in binaries. Moreover, since debug symbols include type information, this causes decompilers such as Hex-Rays to generate different code in the presence of complex types when using debug symbols than it does on ordinary, non-debug executables. This again creates a mismatch between the decompiled output of our target (non-debug) binaries and the source language in the corpus.

Overall, we cannot leverage Hex-Rays directly to automatically populate the original variable names into the decompiler output, thereby creating a parallel, aligned corpus for training our models. This motivates a method for *aligning* the variables in the decompiler output with those in the original C code in a decompiler-agnostic way. Such a method allows us to generate an aligned corpus that is suitable for our application of SMT because it more closely represents decompiled code. We describe how we construct such an alignment procedure in the next section.

3 APPROACH

Figure 2 provides a high-level overview of our approach to assign meaningful names to variables in decompiled C code. The user decompiles a binary using a decompiler (Hex-Rays in our case). The decompiled code is then optionally pre-processed with a hash-renaming optimization (Section 3.3) before being passed to an SMT tool. As basis for our tool chain we use the off-the-shelf SMT system Moses [14]. Moses automatically estimates the language and translation models given a sentence-aligned (line-aligned in our case) parallel corpus [20]. Moses then outputs a possible translation of each line, which we then post-process to extract and assign the suggested variable names (Section 3.2) in the renamed source code.

The quality of the aligned parallel corpus used by Moses to generate the language and translation models is central to the performance of our renaming system. A straightforward approach to generating the training corpus would simply rename the variables in the original, human-written C code to names that could

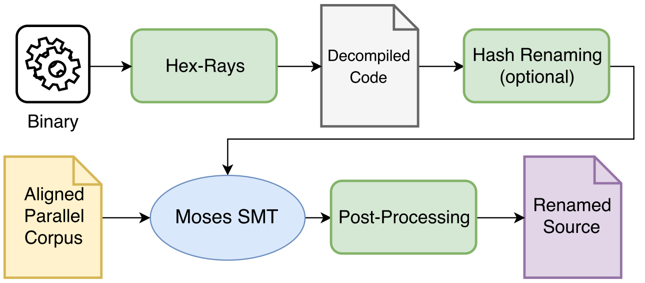

Figure 2: Overview of our technique for renaming variables in decompiled source code.

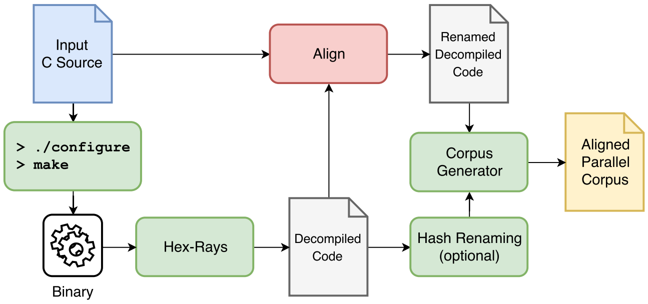

Figure 3: Overview of our approach for generating an aligned parallel corpus.

have been generated by a decompiler, thereby obtaining a parallel, line-by-line aligned corpus. However, as discussed in Section 2.2, because decompilation may substantively change the structure of code compared to its original source, the resulting translation model would perform poorly (since the contexts in which a name appears, the main ingredient for being able to recover names, would be different between the two corpora). We instead generate a corpus via a process of *alignment*.

3.1 Alignment

Training an SMT model requires a parallel corpus of *aligned* content in the two languages between which the model should translate. We produce this parallel corpus by relating the variables in the decompiler output to their correspondents in the original source. Note that **perfect alignment is not always possible**: decompilers often generate extra variables that did not exist in the original source, and often change the code structure with respect to the original. Instead, alignment represents our best guess for appropriate variable names in decompiled code given the original source code.

Figure 3 shows the workflow for generating an aligned corpus. First, we compile input C source code to executables using the configuration scripts and Makefiles supplied with each project.[3] We decompile these executables using Hex-Rays, which generates decompiled code. We then use our alignment techniques (discussed shortly) to map placeholder names in the decompiled code to names in the original source. Finally we combine this with the decompiled code, optionally hash-renaming the decompiled code (Section 3.3) to form the parallel corpus. Moses uses this corpus to estimate both the language model and the translation model (see Figure 2).

When designing our alignment algorithm, we experimented with different combinations of matching strategies and cost heuristics, finding three different combinations that performed best. We evaluated each of these three combinations (denoted A, B, and C) to choose the best-performing one for our system (Section 4.2).

Each of these alignment algorithms starts by separating the code into functions. Splitting code into smaller sections makes the process of alignment computationally tractable, but it limits recovery to local variables. In our experience the vast majority of variables are local, but future work should also investigate the renaming of global variables.

3.1.1 *Matching Algorithms.*

Each of the alignment methods uses a core algorithm that chooses the best matching of variables between two functions. These algorithms take as input two lists of variables and a heuristic for computing the cost of each specific pairing, and find the set of matches that minimizes the total cost. Method A treats variable matching as an instance of the *assignment problem*, where any variable in one list can be matched with a variable in the other list. We chose to use the Hungarian algorithm [21] for this approach.

Methods B and C both treat the problem of assigning variable names in the original source code as an instance of the *sequence alignment problem* [22, Section 3.2]. Given two ordered sequences of symbols and a metric for scoring an alignment between the two, sequence alignment algorithms find the minimum cost (or maximum value) alignment between them. Note that unlike the assignment problem, the ordering of each sequence must be preserved. For example, given the sequences *ABAB* and *AAB*, a cost function that assigns a pairing a cost of 0 if matched symbols are the same character and 2 if they are a different character, and a penalty of 1 for an unmatched symbol, the alignment

$$\begin{array}{cccc} A & B & A & B \\ A & A & & B \end{array}$$

has a cost of 3, while the alignment

$$\begin{array}{cccc} A & B & A & B \\ A & & A & B \end{array}$$

has the minimal cost of 1. The sequence alignment problem is common in biology when aligning multiple DNA or RNA sequences that are billions of symbols long and may have gaps or extra subsequences; as a result, many efficient algorithms have been developed to address it. Alignment methods B and C both use the Needleman-Wunsch algorithm [23].

Note that in all cases, the number of variables in the two functions can differ, so the algorithms need to be able to compute the cost of an unmatched variable, in addition to the cost of a particular assignment. After parameter tuning, we weigh the cost of an unmatched variable by 3 for methods A and B and 1 for method C.

3.1.2 *Signatures and Cost Functions.*

We use two heuristics as cost functions for alignments between variable names in the original source code to those in the decompiled code. These heuristics capture different properties of the variables used in source code.

[3] We use open-source C projects from GitHub; details in Section 4.1.

```
1   // ints x and y previously declared
2   x = 1;     // Usage Signature: =
3   for (;;){
4     y = x+1; // Usage Signature: {+
5   }
6   f(x,y);    // Function Signature: f#1
7   g(y,x);    // Function Signature: g#2
8   x = h(y);  // Function Signature: h#return
```

Figure 4: A small code snippet demonstrating usage and function signatures for the variable x.

Usage Signature. This heuristic penalizes aligned variables that appear to be used in different ways. Each variable is assigned a *usage signature*, or a string of characters that represents the variable's use in unary and binary operations, loops, and assignment. Each time a variable is used, a signature for that usage is generated, consisting of a character representing the operation, and a sequence of characters representing the current nesting depth.

As an example, in Figure 4, the uses of x on lines 2 and 8 both have the signature "=" (where = indicates assignment), and the use of x on line 4 has the signature "{+" (where the character { represents one level of loop nesting, and + represents addition). These smaller signatures are composed to form the variable's usage signature (thus, x as used above has the signature "= {+ ="). We then compute the "distance" between the usage signature of a variable in the original code and that of a variable in the decompiled code.

Each of A, B, and C uses a different method to compute the distance between two usage signatures. To illustrate, we will demonstrate the distance between two strings $str_A = aabc$ and $str_B = abcd$. Method A computes the distance between two strings as the difference in the number of occurrences of each character in both strings (*i.e.*, the symmetric difference between the strings treated as unordered character sets). Since str_A has one more a than str_B, str_B has one more d than str_A, and both strings have the same number of bs and cs, the distance between str_A and str_B is 2.

Methods B and C compute the Levenshtein edit distance (*i.e.*, the number of edits required to transform one string into another string) between the two signatures. Since the second, third, and fourth characters must be changed in str_A to reach str_B (changing a to b, b to c, and c to d respectively), the distance between these two strings is 3. When method B computes the distance, it considers each of the smaller signatures as a single unit and computes the number of smaller sequences that need to be edited. Method C treats each character in the entire signature as a unit and computes the distance with respect to single-character edits.

Each of A, B, and C multiply the computed distance between usage signatures by a coefficient that was found to perform most effectively via a parameter sweep. The coefficients for usage signature distance in methods A, B, and C are 1, 1, and 0.1, respectively.

Function Signature. This heuristic prioritizes aligned variables that are used similarly in function arguments and return values. We generate a *function signature* for each variable, which records the function name and parameter position for that variable. The function signature also captures whether a variable is used to store a return value. We then compute the distance between the signature for the original source code and the decompiled code.

An example of the function signature can be seen in Figure 4. The use of x as the first parameter in the function f on line 6 has the signature "f#1", while its use as the second parameter in the function g on line 7 has the signature "g#2". On line 8, x is used to store the return value of h, and this generates the signature "h#return". Thus, the entire function signature of x is "f#1 g#2 h#return".

Methods A and B use the same distance metric for the function signature as they use for the usage signature. Method C treats each function signature as an unordered set of tokens,[4] and computes the symmetric distance between signatures (cf. method A). This outperformed Levenshtein distance. We hypothesize that this is because uses in function arguments or return values are salient, and the extra context provided by ordering is not needed.

As before, each of the methods multiplies the computed distance between function signatures by a coefficient. We found the best coefficients for A, B, and C after a parameter sweep to be 5, 2, and 1, respectively.

3.2 SMT for Renaming Variables

To generate a candidate list of renamings given a trained SMT model, the decompiled source code is fed into Moses line-by-line. Moses returns a list of possible translations for each line. Our process extracts candidate identifier names from the returned line and stores them as suggested new names for each source variable. SMT tools do not have a mechanism for ensuring that the translation of a single word is consistent between sentences, since natural languages do not have the same strict definition of scope as programming languages do. In other words, the same variable v1 appearing on multiple lines in the decompiled code may receive different "translations" from Moses, one for each line. Clearly, only one suggested renaming should be chosen to avoid breaking the semantic equivalence of the decompiled code compared to the original.

To ensure consistency, we adapt the following strategy. For each candidate renaming, similarly to JSNaughty [10], we rename all in-scope instances of the old variable in the current function. We then select the most-probable renaming *across all lines* according to the language model (Section 2.1). This process is repeated independently for all variables with at least one candidate renaming, assuming that all other variables remain unchanged.[5]

The "translated" version of a program with renamed variables should be an α-renaming of the variables, *i.e.*, the structure of the program should otherwise be preserved. Because different natural languages often have different word orderings, SMT tools like Moses do not necessarily preserve structure. As such, Moses could theoretically translate one line of source code into a structurally different line of code. However, because we want Moses to only perform α-renaming, we disabled configuration options that allow structural transformations, and enabled an option that forces Moses to preserve the number of tokens during translation. These settings, together with the way in which we construct the parallel training

[4] We specify *tokens* here instead of characters because when computing distances between signatures we treat function names and the return signature as a single token, instead of a sequence of characters.

[5] An alternative approach would be to rename each variable sequentially by greedily selecting the renaming that results in the most likely code sequence given the renamings that have already been selected. We suggest this for future work.

corpus (the source and target parts of the corpus are α-renamings of each other), should ensure the structure of programs is preserved.

3.3 Hash Renaming

Hex-Rays assigns names to variables using a prefix and an index (*e.g.*, `a1` and `v5` in Figure 1). These names are merely placeholders and do not convey any meaningful information by themselves. For example, there is no meaningful difference between `for (int v1 = 0; v1 < 10; v1++)` and its α-renamed counterpart `for (int v5 = 0; v5 < 10; v5++)`.

To increase the probability that these two lines (and others like these) are translated similarly, we can *canonicalize* variable names. For example, a simple canonicalization could name all variables identically (*e.g.*, `v`). However, this misses an important opportunity to encode additional context about each variable in its name. We can accomplish this by replacing each variable use in the decompiled source with a *hash* that captures context. We capture three different kinds of context when generating hashed names:

- **Type:** The variable is renamed to a hash of its type.
- **Argument Position:** Variables that are introduced as formal parameters in a function are renamed with a hash of their argument position. Argument order is generally preserved by the compilation process, and Hex-Rays attempts to recover the original order of arguments in the decompiled code. However, we do note that argument order may be changed in the presence of advanced compiler optimizations (*e.g.*, link time optimizations), though we did not employ these in our current work.
- **Most Informative Line:** The variable is renamed to a hash of the highest-entropy line (*i.e.*, highest information content) that it appears in, excluding lines with entropy above a certain threshold because they are unlikely to reappear in the corpus. The entropy of a line is computed from a language model that was trained when all variable names were renamed to a fixed string. This allows us to measure whether the line itself is "interesting", regardless of the variable names.

These optional hashing strategies are independent and can be combined arbitrarily. We evaluate the performance of all eight combinations in Section 4.3.

4 EVALUATION

Our goal is to automatically assign *meaningful* and *readable* names to variables in decompiled C code to assist reverse engineers. This section describes experiments that validate our SMT-based approach's success at this task. In Section 4.1, we describe our experimental setup, including dataset and metrics. Alignment is a critical element for both generating our parallel corpus and validating our technique; we proposed a number of alignment procedures to that end (Section 3.1). In Section 4.2, we evaluate the precision and recall of each alignment procedure. We use the best-performing alignment procedure for all subsequent experiments. We measure how often we can recover the original variable names or an approximation of them in Section 4.3. We conclude by exploring the utility of incorporating additional information into our analysis in Section 4.4, and conversely study the effect of training on a smaller corpus in Section 4.4.3.

4.1 Experimental setup

4.1.1 Dataset. We generated our training corpus from a large number of C files sourced from GitHub. We used the GHTorrent [24] service to identify 402,925 projects written in C. We randomly selected 20,225 of these projects, consisting of 1.2 TB of code and 8.4 billion lines of C, and downloaded them. For each project, we automatically executed available `configure` scripts and then ran `make`. We added a wrapper around `gcc` to ensure that all binaries were compiled with optimizations disabled (`-O0`). In total, we automatically compiled 174,383 binaries (note that many GitHub projects build multiple binaries).

We split the compiled binaries into different sets for training and testing. We randomly assigned each binary to a training, test, tuning, or validation set with probability 94%, 3%, 0.5% and 2.5% respectively. We used the training and tuning sets to generate the parallel corpora that Moses uses to estimate its statistical models. We used the test set to evaluate the system. The validation set was reserved for various manual testing, alignment heuristics parameter tuning (see above), and experimentation.

4.1.2 Alignment metrics. We consider alignment to be successful when it correctly maps a decompiled variable to its corresponding name in the original code. To establish ground truth for alignment and therefore be able to evaluate different alignment strategies, we first compile the original source code *with debug symbols*. We then use the Hex-Rays decompiler to generate decompiled source code from these binaries, which maintains the original variable names because of the debug symbols. Next we strip the variable names from this source code and replace them with dummy names (`v1`, `v2`, etc.), consistent with how Hex-Rays would have named them in the absence of debug symbols. We then attempt to align the variable names between the original source code and the decompiled source code containing these dummy names. Finally, we compare each dummy variable name with the original variable name that we replaced it with.

This evaluation strategy is reasonable for the alignment procedure, but not for the actual renaming. This is because the types of contextual differences between code decompiled by Hex-Rays with and without debugging information, such as type names, are not components of the heuristics we use for alignment (which we wanted to keep decompiler-independent). As a result, our alignment system should perform equally well on code decompiled with and without debugging symbols. We use this approximation, which can be automated, in lieu of a human manual evaluation, which is prohibitive on a dataset of this size.

4.1.3 Renaming metrics. Unlike when we evaluate our alignment techniques, we cannot evaluate our variable renaming accuracy by simply decompiling the program with debug symbols. This is because the SMT toolchain *does* use additional information provided by debugging symbols: we show in Section 4.4 that variable renaming performs better on programs with debug symbols, presumably because the decompiled output contains better typing information. Since debug symbols are unlikely to be present in real binaries, this approach would be unrealistic. Instead, we simply assume that the original name identified by our best alignment method is the correct one, which makes no assumptions about presence of debug symbols in the binaries. Recall that even though alignment is not completely accurate, it does represent our best guess for the correct variable names, and many variables often have no corresponding name in the original source.

Again, our goal is to suggest *meaningful* renamings for variable names in decompiled code, that fit well the context in which the variables are used. Meaningfulness, or naturalness, is inherently subjective, and likely depends on a multitude of factors, including the exact reverse engineering application. To keep the evaluation tractable, we use proxies for naturalness which can be automated.

We assume that recovering the exact variable name in the original source code provides substantial benefit to a reverse engineer. We consider a renamed variable to be an *exact match* if it is identical to the corresponding variable name in the original source code. Additionally, several studies show that humans work just as well with abbreviated identifiers as they do with full-word identifiers [25, 26]. We therefore also assume that abbreviated identifiers (*e.g.*, `ctx` in place of `context`) provide a similar level of utility as exact matches. With this in mind, we additionally count *approximate matches*, identified by the following rules:

- One variable name is a prefix of the other and at least half as long. For example, `str` and `string` match this rule, but `s` and `string` do not.
- Both variables consist of a sequence of letters followed by a sequence of numbers, and the non-numeric part of the names match and constitute at least half of the length of the longer name. For example, `str1` and `str2` match this rule, but `v10` and `v11` do not.
- Special cases that were manually added by inspecting the results on the validation set (not used during testing), such as `format` and `fmt`.

Collectively, exact and approximate matches provide a conveniently automated, but conservative estimate of the utility our renaming approach provides. However, it is not necessary for a variable name to even resemble the original name for it to be meaningful or useful. For example, `count` and `quantity` are both reasonable names for a variable holding the number of items in an inventory, but our metric would not identify these as approximate matches. It is even theoretically possible that our system could suggest a more descriptive name than the original programmer provided, if such a name was used more often in a similar context across a large corpus. We do not currently count such synonymous names in our results, but we do outline some concrete examples where we encountered them during our experiments. A human study would be required to measure the utility of such matches, but we leave such a study to future work.

We also note that our approach can likely be considered as a "do no harm" approach: non-placeholder names should, in theory, always be preferable to placeholder names, unless there are situations when the more natural names can cause additional confusion, which we expect is rare. Human studies are necessary to disentangle these effects, which goes beyond the scope of our current work.

Table 1: Precision and recall of the three configurations of alignment parameters described in Section 3.1.

Configuration	Precision	Recall	F Measure
A	86.1%	70.0%	.77
B	93.2%	69.4%	.80
C	91.3%	72.8%	.81

Table 2: Confusion matrix for our chosen alignment technique. There were 501,711 variables total, of which 333,153 had original names.

		Aligned	
		Positive	Negative
Actual	Positive	242,471	80,565
	Negative	23,097	155,638

4.2 Alignment

Table 1 shows the precision, recall, and F-measure for each of the three alignment configurations described in Section 3.1. Based on these results, we selected alignment method C for subsequent corpus generation and evaluation. We chose this configuration because C has a slightly higher F measure than B, and we found in initial experiments that a model trained using the corpus generated with this method recovered more variables.

Table 2 shows more detailed results of configuration C's performance as a confusion matrix. When using debug symbols, our ground truth, we were able to detect 333,153 variables out of 501,711 that were aligned between the original and decompiled source code. Of these variables, our best alignment procedure (C) correctly identified the corresponding variable 242,471 times, for a recall of 72.8%. Of the remaining 90,682 variables, the alignment procedure failed to report any alignment for 80,565 (24.1%) of them, and reported an incorrect alignment for 10,117 (3.0%). In addition, the alignment procedure incorrectly aligned 12,920 variables that were introduced by the decompiler, and thus have no corresponding variable in the original source code. The incorrect alignments were combined in the false positive cell of Table 2. This corresponds to a precision of 91.3%.

4.3 Baseline results

Table 3 reports how often our techniques can recover variable names that are either exact matches or the combination of exact and approximate matches (as defined in Section 4.1). The *No Alignment* and *Alignment* columns represent our baseline results. *No Alignment* refers to the results produced when we generated the "foreign" language by α-renaming variables in the original source code to match the generic variable names produced by Hex-Rays (*i.e.*, `v1` and `a1`, cf. Section 2.2). In *Alignment*, we instead generate our parallel corpus using our alignment technique as described in Section 3.1. The *Hashed Context* column describes the type of context hashed as canonical variable names (Section 3.3). The *Exact* column reports the percentage of variable names suggested by the technique that are identical to the original variable names, while

Table 3: Percentage of exact and combined (exact + approximate) matches for our renaming technique.

	No Alignment		Alignment		Local		Additional Context	
Hashed Context	Exact	Combined	Exact	Combined	Exact	Combined	Exact	Combined
None	5.3%	8.3%	12.1%	15.4%	20.7%	24.0%	26.1%	33.7%
Type	3.2%	5.8%	11.9%	15.4%	20.7%	25.3%	28.6%	37.1%
Entropy	4.3%	6.7%	11.6%	15.1%	21.4%	25.2%	27.6%	34.6%
Arg. Pos.	5.3%	8.1%	12.5%	16.1%	20.9%	25.0%	26.9%	34.9%
Type + Entropy	2.9%	5.0%	11.9%	15.4%	20.8%	24.5%	28.7%	36.0%
Type + Arg. Pos.	4.2%	7.1%	12.7%	16.2%	21.6%	25.5%	28.0%	36.1%
Arg. Pos. + Entropy	4.7%	7.4%	12.7%	15.1%	21.6%	25.0%	27.9%	34.6%
Type + Arg. Pos. + Entropy	3.8%	6.1%	11.8%	15.3%	23.1%	26.5%	28.3%	35.2%

```
my_rc base2_string(base2_handle base2_h,
  char* buffer, size_t buffer_size) {/*...*/}
```

(a) Original source code.

```
my_rc base2_string(base2_handle a1,
  char* a2, size_t a3) {/*...*/}
```

(b) Decompiled source code.

```
my_rc base2_string(base2_handle base2_h,
  char* buf, size_t len) {/*...*/}
```

(c) Renamed decompiled code.

Figure 5: Header of a function renamed using our technique. In this instance `base2_h` is recovered exactly, `buffer` is recovered approximately as `buf`, and `buffer_size` is not successfully recovered.

the *Combined* column reports both exact and approximate matches that meet the criteria described above in Section 4.1.

Examples of exact and approximate renamings, in addition to a failed renaming can be seen in Figure 5. In this example, `base2_h` is recovered exactly by our technique, while the variable `buffer` is approximately recovered as `buf`. The system fails to successfully recover `buffer_size`, suggesting the name `len` instead. Note, however, that `len` could be considered a reasonable alternative name for `buffer_size` in some cases, and is almost certainly a better choice than `a3`, the name assigned by Hex-Rays.

As can be seen in the table, using *Alignment* to generate a parallel corpus produces an SMT model that can recover significantly more variable names than the naïve alternative in all cases. Without applying contextual hashing, we exactly recover 12.1% of the original names and a combined 15.4% of the exact and approximate names when we use *Alignment*, while we are only able to exactly and approximately recover 5.3% and 8.3% of the original names, respectively, in the *No Alignment* experiment. As we explain in Section 2.2, we attribute the poor performance of the *No Alignment* configuration to the non-idiomatic constructs that decompilers generate, and this largely motivates our alignment-based approach. Interestingly, applying contextual hashing increases the performance of the system under *Alignment*, but *decreases* the performance of the system with *No Alignment*. In the best case, we are able to recover 12.7% of the original names exactly and 16.2% approximately using *Alignment*, while we are only able to recover 5.3% and 8.1% with *No Alignment*, respectively.

While the recovery of 16.2% of the names may seem low, recall that: 1) current decompilers do not attempt to assign any meaningful names to variables;[6] 2) these numbers are computed under the most conservative of assumptions; and 3) exact or approximate recovery is but a proxy for meaningful or naturalness of names. We believe that providing reverse engineers with even a few meaningful names greatly aids code comprehension and reduces some of the mental effort involved in the complex task of reverse engineering. Furthermore we expect that some suggested variable names may be meaningful and useful even if they do not meet the relatively strict criteria that we require for an exact or approximate match.

An example of this is shown in Figure 6. Our system suggested `name` in place of the original name `mapname`, which we do not count as an approximate match even though it provides useful context. Using the information provided by the identifier `name`, a reverse engineer could conclude on line 3 that `aasworld` is a C struct that holds the value of `name` at offset 88, while the format string on line 5 (`"maps/%s.aas"`) provides the rest of the context needed to know that `name` holds the name of a map. In contrast, `a1`, the identifier assigned by the Hex-Rays decompiler, does not provide the same useful information.

In addition, we have no automated way of evaluating the names assigned by the system for decompiler-generated variables that are not present in the original source code. For example, our system suggests the name `status` in Figure 1c, which we believe is an improvement over the name the decompiler assigned, `v5`, but this is not reflected in our numerical results.

4.4 Additional Information

In this section, we explore other ways to improve our technique's performance, by using additional contextual information when suggesting variable names.

4.4.1 Locality. A common use case of decompilers is in the maintenance of legacy software [6]. For example, a company may have lost the source code for the latest version of a program, but

[6] Some decompilers do have rules for assigning reasonable names to very common identifiers, such as the use of `i` as a loop iterator, but to our knowledge this is the most advanced approach currently used.

```
int AAS_LoadFiles(const char* mapname) {
  \\...
  strcpy(aasworld.mapname, mapname);
  \\...
  Com_sprintf(aasfile, 64, "maps/%s.aas", mapname);
  \\...
}
```

(a) Original source code.

```
int AAS_LoadFiles(const char* a1) {
  \\...
  strcpy(&aasworld[88], a1);
  \\...
  Com_sprintf(aasfile, 64, "maps/%s.aas", a1);
  \\...
}
```

(b) Decompiled source code.

```
int AAS_LoadFiles(const char* name) {
  \\...
  strcpy(&aasworld[88], name);
  \\...
  Com_sprintf(&c, 64, "maps/%s.aas", name);
  \\...
}
```

(c) Renamed decompiled code.

Figure 6: Example of a renaming generated by our technique that does not match the original name, but still provides useful context. Note how the function parameter in the renamed decompiled version was assigned the identifier `name`, while the original code used the identifier `mapname`. This is a more informative name than `a1`, which was assigned by Hex-Rays.

may still possess source code for other software developed by the same engineers (such as an older version of the same system). By adding this older code to the training corpus, it should be possible to improve the performance of our approach by exploiting the *localness* of source code [27, 28]. While all human-written source code is repetitive (*i.e.*, natural), it is even more so when compared to other source code in the same project, module, or function.

To measure the effect of localness on our renaming technique, we generated new testing and training sets on a per-function rather than a per-binary basis. This configuration increases the likelihood that different functions within a binary will all be assigned to either the training and testing sets, which simulates additional context that might be contained in source code written by the same programmers, for instance.

The *Local* column in Table 3 describes the results. In short, locality has a positive impact on our ability to recover variable names: we are able to recover 23.1% of variable names exactly, and a combined 26.5% exactly and approximately, which is an increase of 10.4% and 10.3% respectively. We hypothesize that the increase in performance is primarily due to the capture of project-specific identifiers in the language model.

4.4.2 Context. Some decompilers struggle to recover user-defined types such as `xmlCtxt` in Figure 1a. However, a variable's type is often linked to the purpose and name of the variable. For example, a variable named `count` is much more likely to be of type `int` than type `string`. When given access to more accurate type information, the system should be able to suggest more natural variable names.

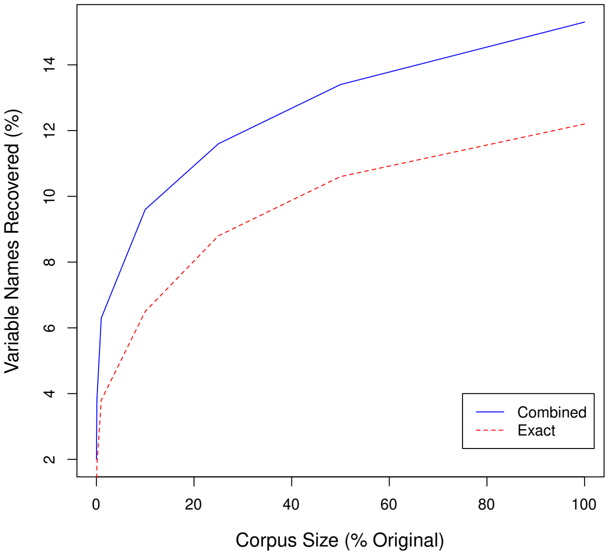

Figure 7: The impact of corpus size on the recovery rate of our technique.

To test this hypothesis, we compiled binaries with debug symbols (using `gcc -g`), which Hex-Rays uses both to name variables and assign their types. We then stripped the variable names from the decompiled code, and applied our SMT technique to recover those names. Training was performed using a corpus generated using the alignment technique, as in the previous experiments. This allows for direct comparison between the techniques, isolating the impact of more accurate types.

The *Additional Context* column of Table 3 shows the results. The additional context of accurate types significantly improves our ability to recover variable names. We are able to recover 28.6% of variable names exactly and a combined 37.1% exactly and approximately. Our technique does not require any additional training or information to take advantage of the additional context provided by better type information. This means that the technique presented in this paper is likely to benefit from future improvements that researchers develop in type recovery.

We do note, however, that these numbers are also likely an upper-bound on the performance of this technique, and that additional algorithmic improvements would be required to recover more variable names. Other promising avenues for improvement include corpus generation with better alignment heuristics, the addition of boosting techniques to improve classification [29], or moving to more recent algorithms used in NLP such as neural networks [30].

4.4.3 Amount of Training Data. The corpus size used in the preceding experiments is quite large; the collection and compilation of over a terabyte of code is not always practical. We therefore performed another experiment to evaluate the impact of corpus size on our results. To perform this experiment, we generated a new training set the same size as used in the original evaluation and then randomly subsampled this training set to create new, smaller collections of training data. For this evaluation, we use the same alignment method as in the baseline evaluation, and the *Type + Arg. Pos.* contextual hash configuration, since it performed the best in our original evaluation.

The results of this evaluation are shown in Figure 7. In this graph, the number of exact matches are represented by the red dotted line, and the number of combined exact and approximate matches are represented by the solid blue line. Note that the number of variable names recovered is not linear and increases rapidly at small corpus sizes. This suggests that our technique could be useful even with a much smaller corpus size. With a corpus size an order of magnitude smaller than the full corpus, we were still able to recover 6.5% of the original variable names exactly and 9.6% of the variable names exactly or approximately, as compared to 12.7% and 16.2% respectively when using the full corpus.

5 RELATED WORK

Our work is closely related to the fields of decompilation and reverse engineering. We also adapt and expand on work done on the application of natural language processing techniques to software engineering problems.

5.1 Reverse Engineering and Decompilation

Decompilation of executables is a large field, with applications to malware analysis [3–5], security auditing [3, 6, 7], and maintainance of legacy software [6]. Decompilation research stretches back several decades [6]. Many modern decompilers are based on the pioneering idea that decompilers should be engineered using similar design as *compilers*, with explicit front and back-ends that are connected by an intermediate language [31]. This shift in design allowed decompilers to be organized as a series of modular transformations in which each transformation recovers a different type of abstraction.

This design has allowed subsequent decompiler research to focus on improving techniques for one type of abstraction recovery, rather than the engineering of a decompiler as a whole. For example, Phoenix [7] and DREAM [3] both proposed new methods for recovering control flow structure (*e.g.*, transforming `goto` statements to `while` loops). Other decompiler researchers have proposed new methods for recovering information about types and variable names [32–34]. Although in this research type recovery and variable recovery go hand in hand, variable recovery is usually limited in scope to identifying storage locations and the context in which they are used in executable code. In particular, this existing work on variable recovery does *not* attempt to recover meaningful variable names for variables. We hope that this paper will motivate researchers to include meaningful names as a component of variable recovery in the future.

We are not aware of any other work that attempts to recover variable names in decompiled code. The most closely related work to ours is the recovery of identifiers in obfuscated JavaScript by JSNice [9], Context2Name [12], and JSNaughty [10]; our technique is directly inspired by the latter.

5.2 Naturalness of Software

The application of natural language processing techniques to software is possible because code is *natural*. It is well known that short code sequences are rarely unique [35], and Hindle *et al.* [13] demonstrated that statistical language models can be *more* effective at capturing regularities in software source code than in natural language because of this effect. Allamanis *et al.* [8] also leveraged this property to learn coding conventions, and suggest natural identifier names and formatting in a development environment. This naturalness property has enabled us and other researchers to generate probabilistic models of source code and apply them to software engineering problems [36–38].

5.3 Readability

The problem of software readability is also well-studied, and researchers have developed models of software readability that measure the difficulty of reading and comprehending source code [25, 39, 40]. These models incorporate identifier names as a component, and more research has shown that careful choice of identifier names aids in the comprehension of software [1, 2]. Other research has shown that although identifier names can largely be arbitrary, programmers carefully choose identifier names to convey meaning to readers of their code [41]. Readability has inspired research into techniques for the automated suggestion of method, class, [42] and unit test [43, 44] names.

6 CONCLUSION

Understanding executable programs without the use of source code is a significant challenge for reverse engineers. Although modern decompilers can effectively recover variables, types, and high-level code structure, they do not recover meaningful variable names, which are an important component of software readability. Our results show that meaningful variable recovery *is possible* by leveraging the fact that code is *natural*. Furthermore, our techniques for recovering variable names can be applied to the output of any suitable executable decompiler to improve readability and reduce the cognitive burden required to comprehend the code.

REFERENCES

[1] E. M. Gellenbeck and C. R. Cook, "An investigation of procedure and variable names as beacons during program comprehension," Oregon State University, Tech. Rep., 1991.

[2] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? A study of identifiers," in *International Conference on Program Comprehension*, ser. ICPC '06, 2006, pp. 3–12.

[3] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations," in *Network and Distributed System Security Symposium*, ser. NDSS '15, 2015.

[4] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping Johnny to analyze malware: A usability-optimized decompiler and malware analysis user study," in *IEEE Symposium on Security and Privacy*, ser. SP '16, 2016, pp. 158–177.

[5] L. Durfina, J. Kroustek, and P. Zemek, "PsybOt malware: A step-by-step decompilation case study," in *Working Conference on Reverse Engineering*, ser. WCRE '13, 2013, pp. 449–456.

[6] M. J. van Emmerik, "Static single assignment for decompilation," Ph.D. dissertation, University of Queensland, 2007.

[7] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *USENIX Security Symposium*, ser. USENIXSEC '13, 2013, pp. 353–368.

[8] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Symposium on the Foundations of Software Engineering*, ser. FSE '14,

2014, pp. 281–293.

[9] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Symposium on Principles of Programming Languages*, ser. POPL '15, 2015, pp. 111–124.

[10] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated JavaScript names," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '17, 2017, pp. 683–693.

[11] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of Android applications," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016, pp. 343–355.

[12] R. Bavishi, M. Pradel, and K. Sen, "Context2Name: A deep learning-based approach to infer natural variable names from usage contexts," TU Darmstadt, Department of Computer Science, Tech. Rep., November 2017.

[13] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. International Conference on Software Engineering (ICSE).* IEEE, 2012, pp. 837–847.

[14] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, and E. Herbst, "Moses: Open source toolkit for statistical machine translation," in *Association for Computational Linguistics Interactive Poster and Demonstration Sessions*, ser. ACL '07, 2007, pp. 177–180.

[15] P. Koehn, F. J. Och, and D. Marcu, "Statistical phrase-based translation," in *Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*, ser. NAACL HLT '03, 2003, pp. 48–54.

[16] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! '14, 2014, pp. 173–184.

[17] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '13, 2013, pp. 651–654.

[18] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *International Conference on Automated Software Engineering*, ser. ASE '15, 2015, pp. 574–584.

[19] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *ACM SIGPLAN Notices*, vol. 49, no. 6, 2014, pp. 419–428.

[20] D. Marcu and W. Wong, "A phrase-based, joint probability model for statistical machine translation," in *Conference on Empirical Methods in Natural Language Processing*, ser. EMNLP '02, 2002, pp. 133–139.

[21] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[22] C. Setubal and J. Meidanis, *Introduction to Computational Molecular Biology*, 1997.

[23] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[24] G. Gousios, "The GHTorrent dataset and tool suite," in *Working Conference on Mining Software Repositories*, ser. MSR '13, 2013, pp. 233–236.

[25] R. P. L. Buse and W. Weimer, "Learning a metric for code readability," *Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, jul 2010.

[26] G. Scanniello, M. Risi, P. Tramontana, and S. Romano, "Fixing faults in C and Java source code," *Transactions on Software Engineering and Methodology*, vol. 26, no. 2, pp. 1–43, 2017.

[27] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Symposium on the Foundations of Software Engineering*, ser. FSE '14, 2014, pp. 269–280.

[28] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '17, 2017, pp. 763–773.

[29] R. E. Schapire, "The boosting approach to machine learning: An overview," in *Nonlinear Estimation and Classification.* Springer New York, 2003, pp. 149–171.

[30] Y. Goldberg, "Neural network methods for natural language processing," *Synthesis Lectures on Human Language Technologies*, vol. 10, no. 1, pp. 1–309, 2017.

[31] C. Cifuentes, "Reverse compilation techniques," Ph.D. dissertation, Queensland University of Technology, 1994.

[32] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS '11, 2011.

[33] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, 2013, pp. 51–60.

[34] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, 2016, pp. 27–41.

[35] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Symposium on the Foundations of Software Engineering*, ser. FSE '10, 2010, pp. 147–156.

[36] M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *International Conference on Machine Learning*, ser. ICML '15, 2015, pp. 2123–2132.

[37] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 731–747, 2016.

[38] P. Bielik, V. Raychev, and M. Vechev, "PHOG: Probabilistic model for code," in *International Conference on Machine Learning*, ser. ICML '16, 2016, pp. 2933–2942.

[39] J. Dorn, "A general software readability model," Master's thesis, 2012.

[40] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Working Conference on Mining Software Repositories*, ser. MSR '11, 2011, pp. 73–82.

[41] B. Liblit, A. Begel, and E. Sweeser, "Cognitive perspectives on the role of naming in computer programs," in *Annual Psychology of Programming Workshop*, ser. PPIG '06, 2006.

[42] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '15, 2015, pp. 38–49.

[43] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *International Conference on Automated Software Engineering*, ser. ASE '16, 2016, pp. 625–636.

[44] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names Or: would you name your children thing1 and thing2?" in *International Symposium on Software Testing and Analysis*, ser. ISSTA '17, 2017.