Statistical Machine Translation is a Natural Fit for Automatic Identifier Renaming in Software Source Code

Jeremy Lacomis,[†] **Alan Jaffe,**[†] **Edward J. Schwartz,**[‡†] **Claire Le Goues,**[†] and **Bogdan Vasilescu**[†]
[†]Carnegie Mellon University, [‡]Software Engineering Institute, Pittsburgh, Pennsylvania
{jlacomis, apjaffe, clegoues, vasilescu}@cmu.edu, eschwartz@cert.org

Abstract

Advances in natural language processing have led to a variety of successful tools and techniques for solving problems such as understanding, generating, and translating natural languages. Given the success of these techniques, a natural question is whether they can also be applied to *programming* languages. However, the initial research has been mixed. Researchers attempting to translate between programming languages by employing statistical machine translation (SMT) found that a large percentage of the translated programs were not syntactically valid. On the other hand, SMT has been successfully employed to recover identifiers in obfuscated JavaScript code. In this paper, we discuss several differences between natural languages and programming languages that can thwart successful application of NLP techniques to program transformation. We also discuss several strategies to cope with these differences in practice, using our own experiences with using SMT to assign meaningful identifier names to variables in decompiled C programs as an example.

1 Introduction

Natural language processing (NLP) is, by now, a very mature field, responsible for solutions to problems such as translating, understanding, and generating natural language. These successes, coupled with increasing availability of large source code corpora from "Big Code" archives such as GITHUB and STACK OVERFLOW, have inspired a plethora of software engineering applications in recent years, *e.g.*, to program generation and transformation (Karaivanov, Raychev, and Vechev 2014). The key ingredient that made these applications possible is the "naturalness" property of source code (Devanbu 2015), *i.e.*, over a large corpus source code is very repetitive (Gabel and Su 2010) (even more so than natural language), therefore predictable using statistical models.

A tempting NLP approach for software source code applications is statistical machine translation (SMT): Can we "translate" programming languages the same way we translate, say, between English and Korean? Early attempts are promising, with notable examples in automatic porting of source code between different programming languages (Nguyen, Nguyen, and Nguyen 2013; 2014; Karaivanov, Raychev, and Vechev 2014) and, more recently, automatic renaming of minified identifiers[1] JavaScript code (Vasilescu, Casalnuovo, and Devanbu 2017).

In this paper we argue that automatic renaming of source code identifiers is a problem particularly well suited for SMT, and we show that it has broader applicability beyond obfuscated JavaScript code, *e.g.*, assigning meaningful variable names in decompiled C code. Our key insight is that both obfuscated JavaScript and decompiled C code can be seen as distortions (cf. the noisy channel model) of *same-language* original code: Unlike "translating" between, say, C# and Java, obfuscated JavaScript and decompiled C are still JavaScript and C, respectively.

2 SMT for Identifier Name Recovery

What makes SMT an appropriate solution to the problem of recovering natural identifier names in source code (*e.g.*, in obfuscated JavaScript (Vasilescu, Casalnuovo, and Devanbu 2017)), more so than for the more general problem of translating, say, between C# and Java? We identify several important properties:

- **The "naturalness" of software:** Due to the high degree of predictable repetition in source code, if given sufficient training data, SMT can successfully capture and exploit *contextual regularities* to provide nuanced, useful translations: In general, identifier names are chosen by programmers to be natural, unsurprising, and well-suited to local context (Devanbu 2015). Across a large corpus, names tend to repeat in similar contexts, and this can be captured by statistical language models.
- **Similar structure between source and target languages:** Programming languages require strict adherence to syntactic and semantic rules and programs cannot be interpreted or compiled when there is even a single error. This is much different from natural languages: humans can still makes sense of sentences that contain errors in syntax or grammar (*e.g.*, this sentence). In the JavaScript reverse minification example, this problem is avoided completely: The transformation between the obfuscated and renamed code is an α-renaming that does

[1]Minification is a common form of obfuscation in JavaScript code on the web, that involves replacing all identifiers with single-letter tokens.

not alter the program structure. In the more general case of translation between different programming languages, this becomes more challenging, as probabilistic models designed for natural languages are not meant to detect and preserve the strict rules and properties that are specific to programming languages.

- **Acceptability of partial/approximate results:** All (translation) models are wrong, but some are useful. Translations between natural languages are often imperfect, yet most people would arguably find them useful, as they can at least point readers in the right direction. Similarly, the goal of SMT for identifier renaming should not be exact recovery of the original names in every single context. Indeed, recovery of names that fit naturally in similar contexts can be, arguably, just as informative. For example, research has shown that programmers are equally able to understand full-word identifiers such as `string` and abbreviated identifiers such as `str` (Scanniello et al. 2017). Additionally, in reverse engineering scenarios, even partial results can be useful: knowledge of some of the variable names can give programmers useful clues to decode the meaning of the remaining obfuscated variables. For example, in code that operates on geometry the code snippet `var1 = len * width;` gives strong clues that `var1` represents area.
- **Arbitrary amounts of training data:** Unlike translation between natural languages, arbitrary-sized training corpora can be constructed for the identifier name recovery problem: As long as one has access to the obfuscator (or compiler), arbitrary amounts of input data, *e.g.*, mined from open source archives, can be passed through. Observing the transformation between the input and the "noisy" output enables the creation of the needed SMT parallel training corpora. In the case of natural languages, parallel, sentence-aligned training data in different languages is relatively scarce. For the more general problem of translating between different programming languages, parallel data is relatively almost inexistent.
- **Context-enhancing transformation:** In most programming languages, any semantic information contained by identifiers is ignored by the compiler, and modifying identifier names does not change the meaning of the program. This is very different from natural languages, where each word carries meaning. Additionally, scoping rules in programming languages do not have a natural language counterpart. These scoping rules allow for the same identifier name to be used in many semantically different contexts. The ability to generate arbitrary amounts of training data has the extra benefit of enabling transformations of the input such that more of a name's context is embedded in the name itself; the only requirement is that this be done consistently, and both prior to training the model and later during live decoding. In general, the more context-specific a name, the better a translation can be learned. Which strategy is optimal in which settings is still unclear; we experimented with different hash-based approaches to embed context into the name itself in prior work on JavaScript deobfuscation (Vasilescu, Casalnuovo, and Devanbu 2017, Section 3.3).

With these properties in mind, we have begun investigating other program transformation problems that can be addressed using SMT tools. We believe that the problem of assigning natural variable names to decompiled code is similar to JavaScript deobfuscation, and that the challenges can be addressed with similar strategies.

3 Renaming Identifiers in Decompiled Code

Security practitioners are often faced with the task of reverse engineering executables without the corresponding source code in order to investigate malware, perform legacy software maintenance, or discover potential software vulnerabilities. Unfortunately, reading and understanding machine code is an extremely difficult task, and reverse engineers often prefer to *decompile* the executable, which transforms the executable semantics into a more abstract source code representation. Although decompilation generally makes code significantly easier to comprehend, there are still many features that cannot be reliably recovered because information is lost during the compilation process. For example, current state-of-the-art decompilers such as Hex-Rays (Hex-Rays 2017), Phoenix (Schwartz et al. 2013), and DREAM (Yakdan et al. 2015) cannot recover comments or identifier names unless debugging symbols are included with the binary, which is rare. Since researchers have demonstrated that variable names help programmers comprehend source code (Lawrie et al. 2006), it follows that recovering these names would also help reverse engineers.

Why SMT? We contend that this problem features the properties we identified in Section 2, and as a result, that we can apply SMT techniques to help solve it. First, we can observe the transformation ourselves (we have access to different compilers and decompilers), therefore we can generate arbitrary amounts of training data. Second, the language transformation we perform preserves program structure and thus syntactic and semantic validity is not a relevant concern. We also find that partial or approximate results are acceptable: if there is not an appropriate translation for a specific identifier in the decompiled code, our SMT approach can use the original identifier. In addition, research shows that there is little difference between the use of full-word identifiers and abbreviations (Scanniello et al. 2017), which suggests that our technique can still provide value without always recovering the exact original name. Finally, we should be able to employ artificial renaming techniques similar to those used for JavaScript deobfuscation (Vasilescu, Casalnuovo, and Devanbu 2017) to capture more of each name's context prior to training, therefore improving our results.

Challenges. Despite the similarities to renaming obfuscated JavaScript code, there are some challenges specific to decompiled code. One major complication is identifying which variables in the decompiler output correspond to those in the original source code, and vice versa, which is essential for constructing the parallel training corpus. This problem is exacerbated by the fact that there is not always a one-to-one relationship, *e.g.*, decompilers often generate extra variables that do not correspond to any variable in the original source code. It is also possible for decompiled code to be syntactically very different from the original source code, yet still be semantically equivalent. For example, `for` loops can be written as `while` loops. This means that we cannot simply look for the original context of a variable to determine its position in the decompiled code.

To improve corpus generation and facilitate automatic evaluation, we have developed several algorithms that attempt to match original variable names with variable names in the decompiled code. These techniques use several different heuristics to align variable names. One heuristic is to compare the uses of a variable in the original code (*e.g.*, the variable is read inside a doubly nested loop) to the uses of the variables in the decompiled code and compute the closest matching usage signature. Another heuristic is to compare how the variables are used in function calls, including their use as return values and their argument position (which is generally preserved by the decompiler). We evaluated this alignment procedure by compiling executables with debugging information so that the decompiler can recover the original variable names. We then manually stripped these names and compared the output of the alignment tool to the original decompiler output. Our best heuristic is able to correctly align 68% of the original variable names.

Although this alignment might seem unnecessary when we can generate debugging information that the decompiler can use to recover variable names, we found this to be an ineffective way to generate a corpus: The decompiler uses additional debugging information, *e.g.*, types, and generates different code when this is available. We found that generating a corpus in this manner hurt our results, likely because of the different surrounding context than is generated when it is not available. We are currently experimenting with using this debugging information to improve our alignment technique, and generate better training corpora.

4 First Results

To evaluate our renaming technique, we collected 1.2 terabytes of C source code from over 20,000 GITHUB projects (identified using GHTorrent (Gousios 2013)), which we split into a testing and training set. We generated a training corpus by compiling using `gcc` with the `-O0` compilation flag to disable optimizations, then decompiling the code using the Hex-Rays decompiler (Hex-Rays 2017), applying context-enhancing hash renamings to variable names, and then using our alignment heuristics to match hashed variable names with the original variable names. We then used this corpus to train MOSES, an open-source SMT toolkit (Koehn et al. 2007). MOSES generates a ranked list of possible translations for each line of source code. Following JS-NAUGHTY (Vasilescu, Casalnuovo, and Devanbu 2017), we renamed each variable in each line with the top suggested name from MOSES, computed its log-probability using a language model, and selected the highest-probability renaming for every variable, following C-specific scoping rules.

Hash Renaming	**Recovered Names (%)**	
	Exact	**Approx.**
None	22.1	24.1
Type	21.8	25.2
Type + Position	24.0	27.2
Type + Entropy	22.1	25.4
Type + Position + Entropy	23.5	26.6

Table 1: Initial results for our name recovery technique.

To evaluate each experiment, the renamed variables were compared to the names the alignment algorithm assigned. Although the alignment algorithm is not 100% accurate, we justify this comparison by noting that the different structure of the decompiled code means that there is not necessarily a "correct" assignment of the original variable names to the variables in the decompiled code.

We also considered an alternative evaluation technique in which we recovered the proper variable renamings by decompiling executables that contained debugging information, since the decompiler is able to use the original names from the debugging symbols. However, our decompiler, Hex-Rays (2017), leveraged type information from the debug symbols to generated code with a more readable structure. This difference in code structure unfortunately means that there is no straightforward α-renaming between the decompiled versions of the executables with and without debug symbols, which is why our alignment procedure is necessary.

We stress that recovering exact original variable names is unnecessary to improve the comprehensability of decompiled source code: abbreviated identifiers can be just as effective (Scanniello et al. 2017). Thus, we also consider it to be a success when our techniques recover an approximation of the original name. We consider a match to be an approximation of the original when any of the following holds:

- One variable name is a prefix of the other, and at least half as long (*e.g.*, `str` and `string`).
- Both variables are a string of letters followed by a string of numbers, and the non-numeric portions are identical and at least half of the length of the longer identifier (*e.g.*, `sum1` and `sum10`).
- Special cases that we manually identified (*e.g.*, `format` and `fmt`).

Although this method is likely to miss some abbreviations (*e.g.*, `s` is not considered an abbreviation of `str`), it is a conservative approach that we believe yields few false positives. We are investigating better techniques and human studies to more accurately define approximate matches.

Table 1 shows the results of our initial experiments. The "Hash Renaming" column identifies the type of information we include in the hash renaming of each variable, cf. (Vasilescu, Casalnuovo, and Devanbu 2017, Section 3.3). The "Recovered Names" columns identify the amount of variable names assigned by our SMT toolchain that match the variable names aligned with the original source code using our alignment algorithms. The "Exact" column refers to identical matches, while the "Approx." column includes both identical and approximate matches. Without any hash renaming strategy our toolchain is able to recover 22.1% of variable names exactly, and 24.1% of variable names approximately. When we use both variable types and argument positions to hash the variable names, our recovery rates increase to 24.0% and 27.2% respectively.

These results show that SMT is a viable strategy for improving the understandability of decompiled source code. Although an identifier recovery rate of 27.2% seems low, recall that current state-of-the-art identifier naming techniques only assign meaningful names in very specific cases (*e.g.*, using `i` as an iterator in `for` loops), and no attempt is made at a general naming solution. As a convenient way to evaluate our models at scale, we count how many of the original names present in the source code prior to compilation and decompilation our models can recover exactly. However, it is likely that different meaningful names are assigned to identifiers. A more nuanced, human-based evaluation is left for future work.

5 Conclusions and Future Work

Our exploration into the use of SMT techniques for automatic identifier renaming shows that despite the challenges in their application to the general translation of programming languages, they can still be successfully used to solve more constrained program transformation problems. The properties of the specific problem of renaming identifiers allow us to restrict the transformations that can be performed by SMT techniques in such a way that the output will always compile, and will always be semantically equivalent to the input program. We do not believe that these properties are unique to identifier renaming, and that there are many other program transformations that should be addressable using SMT and other NLP techniques.

We are considering the use of more advanced machine learning techniques. Neural network methods are now being adapted to NLP problems, including machine translation (Goldberg 2017). We also do not yet use advanced classifier techniques such as boosting (Schapire 2003), which would likely improve our results.

We also believe that the addition of more programming language-specific information should allow us to perform more powerful transformations using SMT techniques. Program analysis techniques could be combined with SMT to add the ability to reason about semantics, improving the quality of generated code or allowing for semantic transformations. There has also been research into improving the grammatical correctness of the output of machine translation techniques through the addition of a context-free grammar for the target language (Wu and Wong 1998) that may be a good fit for programming languages.

References

Devanbu, P. 2015. New initiative: The naturalness of software. In *International Conference on Software Engineering*, 543–546.

Gabel, M., and Su, Z. 2010. A study of the uniqueness of source code. In *International Conference on the Foundations of Software Engineering*, 147–156.

Goldberg, Y. 2017. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies* 10(1):1–309.

Gousios, G. 2013. The GHTorrent dataset and tool suite. In *Working Conference on Mining Software Repositories*, 233–236.

Hex-Rays. 2017. Hex-rays 2.4.

Karaivanov, S.; Raychev, V.; and Vechev, M. 2014. Phrase-based statistical translation of programming languages. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 173–184.

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

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

Nguyen, A. T.; Nguyen, T. T.; and Nguyen, T. N. 2013. Lexical statistical machine translation for language migration. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 651–654.

Nguyen, A. T.; Nguyen, T. T.; and Nguyen, T. N. 2014. Migrating code with statistical machine translation. In *International Conference on Software Engineering*, 544–547.

Scanniello, G.; Risi, M.; Tramontana, P.; and Romano, S. 2017. Fixing faults in c and java source code. *Transactions on Software Engineering and Methodology* 26(2):1–43.

Schapire, R. E. 2003. The boosting approach to machine learning: An overview. In *Nonlinear Estimation and Classification*. Springer New York. 149–171.

Schwartz, E. J.; Lee, J.; Woo, M.; and Brumley, D. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Conference on Security*, SEC '13, 353–368.

Vasilescu, B.; Casalnuovo, C.; and Devanbu, P. 2017. Recovering clear, natural identifiers from obfuscated JavaScript names. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 683–693.

Wu, D., and Wong, H. 1998. Machine translation with a stochastic grammatical channel. In *Meeting of the Association for Computational Linguistics and the International Conference on Computational Linguistics*, COLING-ACL '98. Association for Computational Linguistics.

Yakdan, K.; Eschweiler, S.; Gerhards-Padilla, E.; and Smith, M. 2015. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*.