Deobfuscating Name Scrambling as a Natural Language Generation Task

Pablo Ariel Duboue

Textualization Software Ltd.
Vancouver, British Columbia
CANADA

Abstract. We are interested in data-driven approaches to Natural Language Generation, but semantic representations for human text are difficult and expensive to construct. By considering a methods implementation as weak semantics for the English terms extracted from the method's name we can collect massive datasets, akin to have words and sensor data aligned at a scale never seen before. We applied our learned model to name scrambling, a common technique used to protect intellectual property and increase the effort necessary to reverse engineer Java binary code: replacing all the method and class names by a random identifier. Using 5.6M bytecode-compiled Java methods obtained from the Debian archive, we trained a Random Forest model to predict the first term in the method name. As features, we use primarily the opcodes of the bytecodes (that is, bytecodes without any parameters). Our results indicate that we can distinguish the 15 most popular terms from the others at 78% recall, helping a programmer performing reverse engineering to reduce half of the methods in a program they should further investigate.

1 Introduction

In this work we identify naming obfuscated methods as a well defined, complex task with practical applications that we believe researchers should consider among the problems of interest to AI. The task takes as input obfuscated compiled code, that is, binary code tampered with the objective of hindering human understanding, and seeks to imagine names associated with the original program (in our case, method names).

Besides its practical applications, it exercises the construction of small phrases that describe an object, a subtask of Natural Language Generation (NLG), with an input that sits in the middle between full semantic representations and full sensory input. For this proposed task, it is possible to use program analysis techniques to enrich the input with further semantics, helping to shed light to the AI question of whether we need better semantics or more unsupervised structures built on top of sensory data.

Real semantic representations are quite involved and difficult to come by and as such, recent work focuses in using more vague resources which can be profited using machine learning approaches. Interestingly, the world's complexity that

results in the need for very fine grained semantics to reason about text makes also programming computers difficult: computers require a very fine grained level of detail for the instructions given to them.

In that sense, we see bytecodes-as-semantics not as semantics in the traditional sense of the word (as a logic involving axioms, formulae and predicates) but as semantic breadcrumbs that can be used as sensor data and then processed in a statistical fashion. Such approaches have become more popular in recent years [1–4] and are showing increasing promise in the semantic acquisition problem in NLP.

In addition, deobfuscating name scrambling has plenty of available data. In the work presented here, we used compiled Java code within the Debian archive (consisting of 5 million compiled methods with their respective names). There are also many other options are available (the Maven archive, GitHub, etc). Machine learning approaches to NLG that try to mimick the success of machine translation usually encounter the problem of finding a large comparable knowledge-text corpus. Therefore, looking at the bytecodes as a weak semantic representations that allow for text generation or the reasoning needed for language processing is in itself a contribution.

The problem itself might be too difficult to solve. Indeed, humans performing reverse engineering by hand do not aspire to restore all original names, but just to gain enough understanding of the code to perform the adaptations or audits that motivated the reverse engineering in the first place. In this paper, we focus on identifying the first term in a CamelCased method name,[1] which in the Java naming conventions is equivalent to the verb of the small phrase represented by the name [5]. Moreover, we found the data to be extremely biased with the 15 most popular terms covering half the data. Our current results show we can distinguish 15 top level most popular cases vs. the rest with close to 80% recall. This is useful in practical terms, as the 15 top level cases most likely are of less interest when doing reverse engineering as they cover simpler method unlikely to be of interest to the practitioner and we are thus reducing the number of methods that need to be analyzed by hand by half.

For methods, we trained a Random Forest over features that represent a method by the total opcode counts and the first 28 opcodes. This constrained representation is robust enough in the presence of name scrambling obfuscation and straightforward enough to be computed from compiled code. We also experimented with a neural machine translation system with some preliminary success.

We deployed all our code and data available to the community as part of the Keywords4Bytecodes[2] 1stclass tool[3] and we are also packing our tool as a plugin for the popular BytecodeViewer tool. The rest of this paper is structured

[1] CamelCasing is a convention in many programming language in which multi-word names are combined by having the initial letter for each word capitalized then agglutinate by removing spaces.

[2] http://keywords4bytecodes.org

[3] https://github.com/Keywords4Bytecodes/1stclass

as follows: we discuss our data in the next section, the system and results, related work, discussion and conclusions.

Table 1. Training data sizes.

Data	# classes	# methods	# instructions
Apache (dev-train)	67,217	574,620	11,027,500
Eclipse (dev-test)	93,865	721,153	13,352,704
Apache+Eclipse (train)	161,082	1,295,773	24,380,204
Rest (test)	519,541	4,318,079	89,353,021
Rest sample (obf)	111,562	1,032,290	23,974,818
TOTAL	680,623	5,613,852	113,733,225

2 Data

Collection. The corpus collection was greatly helped by the automatic tools available as part of the Debian project.[4] We performed a rsync process to obtain a functional Debian archive mirror,[5] the proceeded to extract all the binary packages, and searched for all the Java ARchive (.jar) files. That process yielded a total of 7,857 jar files occupying 3.3G on disk.

Doing a random split of all the available data into training and testing will pick up strong coding guidelines from the larger projects that dominate the training data. That will not speak of the generalization power of the system. To avoid this type of contamination of our results and false inferences, we use a development set with all the `org.apache` classes and a development test set of all the `org.eclipse` classes. For final test (and for the results we report) we use all the other classes, a much heterogeneous group without a strong focus or coding structure. This way we expect our results are as strong as possible. We could have done a cross validation across all the methods, but we felt doing that will inflate the results by letting these two projects that have very large code bases dominate. We also choose to training on Apache with Eclipse because it helps to have the highest level of quality in terms of naming conventions.

The three part data split (Apache vs. Eclipse vs. rest) sizes are in Table 1.

To extract opcodes, we use the libary org.ASM (we omit LABEL and LOCAL_VARIABLE as they are usually scrubbed during obfuscation).

[4] We use Debian (http://debian.org) in the event we want to make experiments that involve going back to the source code and have a full build system readily available. It also give us some idea of "notability" for the software and quality. We used the unstable repository as of Sun, 03 Jul 2016 15:39:37 UTC, we have kept all the .deb and .jar files and are making available together with our tool, to facilitate reproducibility of our results.

[5] Following the instructions and scripts at: https://www.debian.org/mirror/ftpmirror.

We only keep methods that have a "first term," that is, a verb to be extracted: we drop all the words after an underscore or after an upper case and ignore methods that start with '<' (internal constructor) or contain a '$' (internal methods).

Obfuscation. In Java bytecode obfuscation, there are three established processes [6] (page 29): (1) name scrambling, the focus of this paper; (2) string encryption, which obfuscates the string constants inside the .class file, a motivation for using only opcodes; and (3) control flow obfuscation, which presents a threat to our approach (see threats to validity in the last section). We focused on ProGuard (that does only name scrambling) due to its popularity and availability. The ProGuard tool is also the default scrambling process in the official Android release as part of the platform tooling. In our experiments, we checked both the original and ProGuard-obfuscated versiones produced the same sequence of opcodes (discussed next).

JVM Internals. As we are using JVM bytecodes rather than source code, we want to include a brief introduction on the JVM. The JVM is a stack machine, with most of its operations happening in the JVM stack. A Java compiler transforms Java source code into sequence of JVM bytecodes, which are then interpreted or just-in-time compiled by the JVM. There are 204 types of bytecodes, each prefixed by a unique opcode. The set of opcodes is quite limited (204) to simplify porting the JVM to different architectures. This vocabulary is much reduced than source code vocabulary.

One of the main benefits of using bytecodes is their reduced vocabulary. Alternatively, learning from Java source code directly will involve dealing with a much larger vocabulary and a tree input (part of the job of the compiler is to linearize the source code). However, there are a number of bytecodes that incorporate either the name of classes or methods, or string constants, making the vocabulary unbounded. For example:

ldc. This opcode pushes a constant onto the operand stack, which can be either a number or a string.

getfield. This opcode takes two parameters, an instance and a field name (actually a field index but in our disassembled output it is represented as a full fledged string) and returns the value of that field.

getstatic. This opcode takes two parameters, a classname and a field name and returns the value of that field.

invokedynamic. Invokes a dynamic method specified by an index in the constant pool (which gets transformed into an actual string by our disassembler).

by using only the opcodes we eliminate these but we might revisit this decision (see future work, Section 6.1).

Call Graph. If the semantics of a Java method are determined by its behavior, it will be short-sighted to consider only the code associated with the method itself: a large part of the behavior of a method is contained in the code associated with methods *called* by the original method.

In Program Analysis [7], the term *call graph* refers to a graph where the nodes are methods directed edges connect them to indicate that a given method calls (in some moment during its execution) the other method.

There are a number of program analysis tools available to obtain a call graph. For example, the fingerprinting techniques (described in the next section) employed by Høst [8] make very limited use of the call graph information. However, in our current experiments, we did not make use of call graphs but we look into one particular behaviour described next.

Wrapper Methods. Early in our experiments, it became clear that a particularity of the Java programming language (lack of optional arguments) generated a large number of spurious "incomplete" methods that without special processing will pose an ill-defined learning problem: due to the lack of default values for arguments, it is common to express one method with different parameter lists which in turn complete the parameter list with default values and call the method with the same name and the full parameter list. We call these methods "wrapper methods". We believe when considered at the bytecode level, they should be indistinguishable irrespective of its name and thus assign them to the unique term WRAPPER that we also learn as all other terms (get, set, etc).

To identify wrapper methods, we sorted all methods with the same full name and keep its associated term for the one with the longest code as measured on number of instructions. The rest were marked as WRAPPER. While simple, this heuristic might be hurting us, as we will further discuss.

3 Related Work

There is ample of work related to ours, either in the name recovery / appraisal starting from source code or property discovery from binary code. While much of the work is indeed related and have informed our work, working from obfuscated bytecodes makes it a different problem than working from clean bytecodes (or source code).

The work most related to ours is the Java Programmer's Phrase Book, the construction of a phrasebook to catalog existing usage of English expressions in method names [5, 9]. In their work, they curated by hand a list of 17 predicates over Java bytecodes (e.g., "does the method contains a branch?", note that 3 of them are not robust under obfuscation, see Table 2) and compare the boolean fingerprint to abstracted versions of the method names themselves. By doing that they could say, for example, that method namaed `add-`*pronoun-etc* "tend to contain loops, often have parameters, uses local variables, etc." (see Figure 1). They use this resource for teaching Java conventions to new programmers [5, 9], detecting naming bugs [10] and improving conventions [11]. At the

end of their research they obtained almost 400 phrases with associated probability distribution over predicates. In our work, by picking the first token in CamelCase construction with no stemming or any further processing, we seek to learn the *verb-* part of such constructs without their relying on their predicates and operating directly from data.

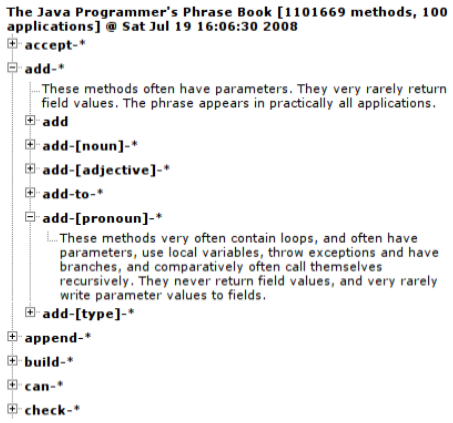

Fig. 1. Programmer's Phrasebook (excerpt). Adapted from http://phrasebook.nr.no.

A related project in mining properties of large code-bases is the Big Code project and the JSNice Tool[6] [12], which deobfuscates names in obfuscated JavaScript code. While our project and theirs address related problems (the closest one in JSNice side is the prediction of name variables), the differences are in dealing with interpreted source code vs. machine code. JSNice uses Conditional Random Fields for MAP inference over program properties. We use Random Forests over lower level features to accomplish similar goals. The MAP CRF approach used in JSNice could be applied to recover obfuscated names of functions in JavaScript, although that is not something it does at the moment. We believe local variable names belong to a smaller subset that function names and might be better suited for that approach. In our case the full names involve a space too big to predict at this time and we are focusing on the first term of the name. Because they are working with source code, the structure of the code can be recovered directly and provide more complex features. Their reported ablation experiments [12] indicate such structure would account for a 16.6% increase in precision. Also of note, their problem starts at 25% as a number of variable names cannot be obfuscated. We obfuscate and predict every method name, so our baseline is much lower.

[6] http://jsnice.org/

Also working from binary code, Chua et al.[13] work on argument recognition from binary data. Another work on name appraisal from source code using a neuro probabilistic approach is that of Allamanis et al. [14].

Our work applies Natural Language Processing to Software Engineering, a topic that has recently gained renewed interest. Other work on the topic includes explaining object hierarchies [15], creating documentation from software specifications [16], coding examples for students [17] (pages 59–73), source code or source differences comment generation [18–20], ontology construction [21], source code summarization [22], and project categorization [23], and situated semantic parsing [24]. Many of these approaches operate over source code. The few systems that work on bytecodes do so by extracting API calls and cite obfuscation as a threat to their validity. In general, the relation between code and human language is receiving renewed attention in both AI and software engineering [25].

From an Artificial Intelligence perspective, real semantic representations are quite involved and difficult to come by. Recent work focuses in using more vague resources which can be profited using machine learning approaches [1–4].

Table 2. Predicates, adapted from [8]. (R) Robust refers to our opinion of whether the predicate is robust under obfuscation: Y(es), N(o), D(epends). (L) Learnable is our opinion of whether our approach might be capturing that predicate, we distinguish Y(es), N(o), P(ossible) and M(issing).

Predicate	R	L
Contains loop. There is a control flow path that causes the same basic block to be entered more than once.	Y	P
Contains branch. There is at least one jump or switch instruction in the bytecode.	Y	Y
Multiple return points. There is more than one return instruction in the bytecode.	Y	Y
Is recursive. The method calls itself recursively.	Y	M
Same name call. The method calls a different method with the same name.	N	N
Throws exception. The bytecode contains an ATHROW instruction.	Y	Y
Writes parameter value to field. A parameter value may be written to a field.	Y	Y
Returns field value. The value of a field may be used as the return value.	Y	Y
Returns parameter value. A parameter value may be used as the return value.	Y	M
Local assignment. Use of local variables.	Y	Y
Reads field. The bytecode contains a GETFIELD or GETSTATIC instruction.	Y	Y
Writes field. The bytecode contains a PUTFIELD or PUTSTATIC instruction.	Y	Y
Returns void. The method has no return value.	D	P
No parameters. The method has no parameters.	D	P
Is static. The method is static.	Y	P
Creates objects. The bytecode contains a NEW instruction.	Y	Y
Run-time type check. The bytecode contains a CHECKCAST or INSTANCEOF instruction.	Y	Y

4 System and Results

Our system uses Random Forests [26] with a feature set (Table 3) composed of three parts: total opcode counts (what is usually known as a Bag-of-Words, BoW features), the exact opcodes for the first 29 instructions (29 is the median length of a method in the Apache codebase) and the length of the method in opcodes. We use random forests because our positional features and our BoW features are highly correlated. Random forests are known to be able to handle such situations very well. Moreover, given the amount of training data, the fact that learning can be parallelized to several CPU core is a very valuable property. In addition, random forests are consistently among the top classifiers in general problems [27]. We use the implementation from the `fast-random-forest`[7] project, an external add-on to Weka. We train 250 trees for each forest.

Our feature representation for each bytecode sequence is a mixture of positional and bag of words. The feature vector contains 236 entries, of which the first 29 are the actual opcodes on the first 29 positions plus the counts for all opcodes on the whole bytecode sequence. The rationale here is that opcodes are very general and very information poor. The actual meaning lies in their sequencing. Because the methods with popular names tend to be short, by keeping the exact beginning of the bytecode sequence, we allow the machine learning component to pick up subsequences of opcodes that form a signature for particular method names.

Table 3. Features employed, divided into three subsets.

Pos	Feature	Type
1	Opcode at position 1	one of 205
...	...	
29	Opcode at position 29	one of 205
30	Total count for opcode 1	int
...	...	
234	Total count for opcode 205	int
235	Method length in bytecodes	int
236	Class	one of target tokens

4.1 Results

Which terms to predict is a difficult question. Ideally, we would like to predict all terms, but many terms are so rare that there is not enough evidence for the prediction to work. We settled on terms that appear at least 1,000 times in our training set (Apache codebase). There were about 50 such terms. Still, a 50-way classification is too complex of a problem. We solved this issue by training on

[7] https://code.google.com/archive/p/fast-random-forest/

Apache and validating on Eclipse. There, we found most of the keywords were indistinguishable from the background model. Only 14 were

- frequent enough to have training data available,
- have a minimal level of performance on Eclipse (which we set as having an F-measure greater than 10% so that the next condition holds),
- dommon enough to be of practical importance, that is, they account for roughly 50% of the training instances.

We applied these criteria during training: whether these terms will perform well when run on the larger, unseen set was unknown to us until the end of our experiments. From a reverse engineering perspective, the practitioners have no names, so having some names improves their current state of affairs.

When adding the background class (OTHER, which encompasses all other possible terms) and WRAPPER (which identifies wrapper methods, for any term) that brings the total number of terms to 16. The OTHER class might be also called "interesting" as for the purpose of reverse engineering, the most popularly named methods would not attract much attention from a practitioner.

When trained on Apache+Eclipse and tested on the rest we obtain the results in Table 4. These results are statistically significant over a binary random baseline (second column on the table, note that this baseline is much higher than doing a 16-way baseline), with an average F1 of 0.49. We also performed a test on a sample that has been run through ProGuard and validated that our results are the same in the presence of obfuscation, a result expected given our strict feature set.

From the table, we can see that the OTHER class is at 78% recall, which we consider our strongest (most useful) result. It should help a practitioner performing reverse engineering to focus their attention on more interesting methods when, for example, stepping through a program on a debugger. The F1 measures for the individual terms are not as good but many terms have a precision above 80%, meaning that, when predicted, a practitioner could trust the classifier's output. A suitable UI that incorporates the confidence of the predictions can be particularly useful.

The Apache+Eclipse model is 185M compressed, 2.45Gb uncompressed. Training was done in a machine with 8 cores and 32Gb of RAM. Training over the Apache set took 50min using 23Gb of RAM of which 4min were spent on GC. We employed 250 trees per forest.

5 Discussion

Our current experiments can be considered as Reverse Engineering using pattern matching techniques or as an ML approach to static analysis. According to [28], to be considered static analysis it has to be doing some form of resolution of the target of jumps. All information is completely lost in the process of extraction of the bytecodes and as such the "large scale pattern matching" concept takes the lead. We find the tension between these two views quite thought-provoking.

Table 4. Results of predicting 14 terms (+OTHER, WRAPPER) trained over Apache+Eclipse and tested on non-Apache, non-Eclipse (4.3M methods from 520k classes).

Token	Count (%)	Baseline	F1	Prec.	Rec
OTHER	2,449,084 (56.7)	0.2835	0.74	0.70	0.78
WRAPPER	544,621 (12.6)	0.0630	0.37	0.50	0.29
add	66,778 (1.5)	0.0075	0.22	0.48	0.14
clone	7,968 (0.1)	0.0005	0.35	0.72	0.23
compare	9,517 (0.2)	0.0010	0.39	0.70	0.27
contains	16,811 (0.3)	0.0015	0.08	0.50	0.04
equals	17,749 (0.4)	0.0020	0.72	0.83	0.64
get	706,435 (16.3)	0.0815	0.54	0.47	0.64
hash	16,479 (0.3)	0.0015	0.49	0.81	0.35
is	117,638 (2.7)	0.0135	0.39	0.44	0.36
jj	12,431 (0.2)	0.0010	0.97	0.97	0.98
next	15,086 (0.3)	0.0015	0.18	0.56	0.10
set	249,094 (5.7)	0.0285	0.57	0.76	0.45
to	63,070 (1.4)	0.0070	0.35	0.61	0.24
value	14,247 (0.3)	0.0015	0.67	0.90	0.53
values	11,071 (0.2)	0.0010	0.80	0.94	0.70

The identification of wrapper methods is trivial when the names are in the clear but it becomes challenging under obfuscation. We believe our current proof-of-concept system biggest utility lies on its capability to highlight such methods plus other trivial ones such as `get-` and `set-`, potentially freeing time from the reverse engineering practitioner for more interesting methods.

Multiple questions arise when looking at work as presented here. The first one has to do with whether the type of semantic breadcrumbs obtained from the bytecodes has anything to do with semantics as understood by the field of NL semantics. This issue is clearly a philosophical question beyond this paper itself, but we believe the recent influx in "soft" treatement of semantics should encourage looking at the problem in a less traditional fashion [1–4].

A separate topic is the use of sequence of opcodes as proxy for real semantic representations (in the linguistic/epistemological sense) for the verbs identified. This view was one of the reasons to embark in this project and one of the basis of using only sequences of opcodes. Interestingly, semantically related words under software engineering semantics has been studied as part of SWordNet project [19]. We believe exploring the relation between confusing tokens in our results and SWordNet semantically related words ("rPairs") opens up exciting possibilities.

An interesting question is whether this constitutes a full-fledged NLG task. On its current form it is arguable. However, if we start looking into more complex phrases encoded as method names (e.g., *canRunInBackground*), we can start seeing how rich the problem is. The related task in NLG is lexical choice and dealing with subtle distinctions, such *getName* vs. *getProperty*, are key to the

task. An appealing characteristic of this problem from the NLG perspective, it is that reverse engineering practitioners are willing to tolerate noisy text, similar to the types of text that can be obtained from using machine learning approaches to NLG.

We performed a qualitative analysis and comparison between the Java Programmer's Phrase Book's predicates [5] and our approach (Table 2). We found that 14 out of 17 predicates should be usable after obfuscation and that our approach should capture also 14 (different 14) of these predicates and can be expanded to capture 16 of them, potentially improving on their work in the presence of obfuscation. A side to side comparison of the approaches will be addressed in future work. Note that one of their predicates ("Same name call") is destroyed by name scrambling but recovered by our wrapper classifier.

Reverse engineering and obfuscation is an arms race between analysts and developers [28]. From a theoretical perspective [29], the task of people seeking to protect their code and secrets is an impossible one (at least without resorting to special hardware). While they cannot stop the attacks from a theoretical standpoint, they can well hinder understanding of the code by humans peering into it. For example, [28] says:

> Identifier names are often critical to human understanding of a program but cannot be fully restored with the help of automated code analysis techniques.

A popular obfuscation technique is then to scrub any identifier left behind by the compiler and, if the runtime system requires an identifier, replace it with a non-sensical name, a practice known as name scrambling [6].

6 Conclusions

In this work, we identified the task of reversing name scrambling as a potential area for AI research, particularly after having been deemed "unsolvable" by traditional reverse engineering analysis [28][8]. We identified it as different from the aforementioned existing work on verification and description, as here the name is *missing* and the compiled code has been *obfuscated*. We showed some promise that the problem might be amenable to a machine learning solution based on opcodes only, as shown in our experiments that differentiate with a recall close to 80% the terms evenly into 15 common terms and situations versus a general case.

A more controversial topic particularly related to this approach is the use of compiled code rather than source code. Mixing source code and NLG has already proven fruitful for automatic comment generation [21] and other tasks [20]. We did not discount such approaches, but we found the low level behavior of compiled bytecodes appealing to statistical processing and machine learning.

[8] Our approach falls in the category of pattern matching for human understanding, which the authors deem unfeasible (page 12).

Moreover, many of the applications of practical interest to us revolve around helping a user build an understanding for a program for which source code is not available.

Our results using a very constrained feature set showed a clear signal from the data. We find these results very inviting to try different approaches and techniques and we hope other AI researchers will be interested in tackling this challenge. As such we are making our code and data freely available.

Our contributions include bringing the deobfuscation of named scrambled methods to the attention of the AI community, a first implementation using a Random Forest and a suitable data set. All the code and data is available for further experimentation by the AI community.

6.1 Future Work

For future work, we are currently performing RNN experiments using OpenNMT [30]. We believe special adaptations to OpenNMT should allow it to outperform the current Random Forest approach. Particularly, we want to incoporate some parameters from the bytecodes using sub-word encodings methodology from Senrrich et al. [31].

We are also interested in extending this work to the Android platform using the `dex2jar` tool and to explore dynamic analysis alternatives. Statistical techniques using sampling of execution points have already shown plenty of promise in Software Engineering [32]. This is complicated as it requires an environment where each individual method can be executed. Under this setting, we can predict the tokens based on the sequence of *executed* opcodes. The linguistic intuition is that methods are named by what they do rather than for what they are.

Acknowledgments. The author would like to acknowledge Annie Ying, Paul Pereira-Brunner for discussion and support. Earlier versions of this work were orally presented at REcon reverse engineering conference and the RALI-OLST seminar series. This work was greatly enhanced by reviewer comments both at the present venue and earlier submissions to other venues.

References

1. Chen, D.L., Mooney, R.J.: Learning to Interpret Natural Language Navigation Instructions from Observations. In: AAAI. (2011)
2. Garoufi, K., Koller, A.: Automated Planning for Situated Natural Language Generation. In: ACL. (2010) 1573—-1582
3. Vogel, A., Jurafsky, D.: Learning to Follow Navigational Directions. In: ACL. (2010) 806—814
4. Branavan, S.R.K., Chen, H., Zettlemoyer, L.S., Barzilay, R.: Reinforcement Learning for Mapping Instructions to Actions. In: ACL/AFNLP. (2009) 82—90
5. Høst, E.W., Østvold, B.M.: The programmer's lexicon, volume i: The verbs. In: SCAM 2007, IEEE (2007) 193–202

6. Cipresso, T., Stamp, M.: Software reverse engineering. In: Handbook of Information and Communication Security. Springer (2010)
7. Grove, D., DeFouw, G., Dean, J., Chambers, C.: Call graph construction in object-oriented languages. In: Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA '97, New York, NY, USA, ACM (1997) 108–124
8. Høst, E.W.: Meaningful Method Names. PhD thesis, Oslo (2010)
9. Høst, E.W., Østvold, B.M.: The java programmer's phrase book. In: Software Language Engineering. Springer (2009) 322–341
10. Høst, E.W., Østvold, B.M.: Debugging method names. In: ECOOP 2009–Object-Oriented Programming. Springer (2009) 294–317
11. Høst, E.W., Østvold, B.M.: Canonical method names for java. In: Software Language Engineering. Springer (2011) 226–245
12. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from big code. In: ACM SIGPLAN Notices. Volume 50., ACM (2015) 111–124
13. Chua, Z.L., Shen, S., Saxena, P., Liang, Z.: Neural nets can learn function type signatures from binaries. In: 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, USENIX Association (2017) 99–116
14. Allamanis, M., Barr, E.T., Bird, C., Sutton, C.: Suggesting accurate method and class names. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM (2015) 38–49
15. Lavoie, B., Rambow, O., Reiter, E., et al.: The modelexplainer. In: 8th IWNLG, Herstmonceux, Sussex (1996)
16. Paris, C., Vander Linden, K., Lu, S.: Automatic document creation from software specifications. In: Proceedings of ADCS '98, the Third Australian Document Computing Symposium. (1998) 26–31
17. Mittal, V.O.: Generating Natural Language Descriptions with Integrated Text and Examples. PhD thesis, USC/ISI, CA (September 1993)
18. Wong, E., Yang, J., Tan, L.: Autocomment: Mining question and answer sites for automatic comment generation. In: ASE. (2013)
19. Yang, J., Tan, L.: Swordnet: Inferring semantically related words from software context. Empirical Software Engineering **19**(6) (2014) 1856–1886
20. Loyola, P., Marrese-Taylor, E., Matsuo, Y.: A neural architecture for generating natural language descriptions from source code changes. In: Proc. of ACL. (2017)
21. Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K.: Towards automatically generating summary comments for java methods. In: ASE, ACM (2010) 43–52
22. Ying, A.T.T.: Code Fragment Summarization. PhD thesis, McGill University (2016)
23. Escobar-Avila, J., Linares-Vásquez, M., Haiduc, S.: Unsupervised software categorization using bytecode. In: ICPC, IEEE (2015)
24. Richardson, K., Kuhn, J.: Learning semantic correspondences in technical documentation. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics (2017) 1612–1622
25. Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P.: On the naturalness of software. Communications of the ACM **59**(5) (2016) 122–131
26. Breiman, L.: Random forests. Machine learning **45**(1) (2001)
27. Fernández-Delgado, M., Cernadas, E., Barro, S., Amorim, D.: Do we need hundreds of classifiers to solve real world classification problems? Machine Learning Research **15** (2014) 3133–3181

28. Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E.: Protecting software through obfuscation: Can it keep pace with progress in code analysis? ACM Computing Surveys (CSUR) **49**(1) (2016) 4
29. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im) possibility of obfuscating programs. In: Annual International Cryptology Conference, Springer (2001) 1–18
30. Klein, G., Kim, Y., Deng, Y., Senellart, J., Rush, A.: Opennmt: Open-source toolkit for neural machine translation. In: Proc. of ACL. (2017)
31. Sennrich, R., Haddow, B., Birch, A.: Neural machine translation of rare words with subword units. In: Proc. of ACL. (2016)
32. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: ACM SIGPLAN Notices. Volume 38., ACM (2003) 141–154