Recovering Clear, Natural Identifiers from Obfuscated JS Names

Bogdan Vasilescu
School of Computer Science
Carnegie Mellon University, USA
vasilescu@cmu.edu

Casey Casalnuovo
Computer Science Department
University of California, Davis, USA
ccasal@ucdavis.edu

Prem Devanbu
Computer Science Department
University of California, Davis, USA
ptdevanbu@ucdavis.edu

ABSTRACT

Well-chosen variable names are critical to source code readability, reusability, and maintainability. Unfortunately, in deployed JavaScript code (which is ubiquitous on the web) the identifier names are frequently minified and overloaded. This is done both for efficiency and also to protect potentially proprietary intellectual property. In this paper, we describe an approach based on statistical machine translation (SMT) that recovers some of the original names from the JavaScript programs minified by the very popular UglifyJS. This simple tool, Autonym, performs comparably to the best currently available de-obfuscator for JavaScript, JSNice, which uses sophisticated static analysis. In fact, Autonym is quite complementary to JSNice, performing well when it does not, and vice versa. We also introduce a new tool, JSNaughty, which blends Autonym and JSNice, and significantly outperforms both at identifier name recovery, while remaining just as easy to use as JSNice. JSNaughty is available online at http://jsnaughty.org.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

Deobfuscation, JavaScript, Statistical Machine Translation

ACM Reference format:
Bogdan Vasilescu, Casey Casalnuovo, and Prem Devanbu. 2017. Recovering Clear, Natural Identifiers from Obfuscated JS Names. In *Proceedings of ESEC/FSE'17, Paderborn, Germany, September 4–8, 2017,* 11 pages.
DOI: 10.1145/3106237.3106289

1 INTRODUCTION

While the primary goal of software programmers is to write programs that perform required tasks according to specifications, programs are also *written to be read.* This was famously noted by Don Knuth:

> *Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. [26].*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE'17, Paderborn, Germany
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-5105-8/17/09…$15.00
DOI: 10.1145/3106237.3106289

Studies show that programmers spend a large portion of their work time reading code and trying to understand it [43]. As such, when writing new code, programmers consciously spend effort to reduce the cognitive burden on those who would later read the code. A central case in point here is the choice of variable names in code: while names don't affect program correctness and function, a well-chosen name, well-fitting its context of use, can considerably improve the readers' comprehension [21, 29]. In some sense, the chosen name must seem "natural" (unsurprising) in the context, so that readers find the code familiar and easy to read. Coding standards, that prescribe how to choose variable names, also serve this purpose. By the same token, a poorly chosen variable name has the opposite effect; indeed, deliberately choosing cryptic or ill-suited names is recognized as a convenient way to obfuscate code. Furthermore, in settings where it is desirable to make code hard to read, there are tools available to corrupt the variable names.

In JavaScript (JS) in particular, there are two synergistic reasons for seeking to corrupt variable names. First, JS programs are downloaded on demand, often over mobile networks; once downloaded, they must be quickly loaded and interpreted. Developers often apply a "minification" technique (removing whitespace, shortening—or mangling—variable names, *etc.*) to shrink files, thus lowering bandwidth usage without affecting functionality. Second, JS programs are shipped as source, thus their logic and any proprietary, clever tricks that are used can easily be viewed within any browser that executes them. There is, hence, a naturally adversarial relationship between the original developers of an application, who want to preserve competitive advantage, and others, who wish to simply reuse the code.

For these two reasons, there is a substantial practical imperative to shorten JS code; since platform API calls and keywords must be retained, local variable names should be made as short, opaque, and confusing as possible. There are tools to automatically process a given JS program, with clear, well-named identifiers, and return an "uglified" program that replaces all the variable names with single-letter names. For example, UglifyJS[1] is an enormously popular such tool, and large amounts of JS code on the internet have been subjected to its distortions before being placed on websites. UglifyJS is particularly aggressive about reusing cryptic variable names in multiple scopes; this improves gzip compressibility (often used *after* minification) and diminishes readability, while leaving program semantics intact. There is, also, a natural need for automatic *natural name recovery*, so that the large amounts of uglified JS could be made more accessible for review, learning, maintenance, reuse, security analysis, *etc.*, especially when *source maps*[2] are not available. This is precisely what our system Autonym does.

[1] https://github.com/mishoo/UglifyJS
[2] https://en.wikipedia.org/wiki/Minification_(programming)#Source_mapping

In this paper, we adopt a simple, yet powerful and language-independent intuition in the design of AUTONYM: *software is highly repetitive*. Gabel & Su [20] observed that code in large corpora is rarely unique, and tends to reoccur. Hindle *et al.* [24] then first provided evidence for the "naturalness hypothesis", showing that repetition in code could be effectively captured in statistical language models, and used for software engineering tasks.

An illustrative task shown by Hindle *et al.* to work well in this regime was code suggestion: given a context κ, a statistical model (estimated from a large code corpus) was used to suggest the next token τ by maximizing (over τ) a probability from a model of the form $p(\tau \mid \kappa)$. Given the high degree of repetitiveness in large code corpora, these models can make highly accurate predictions, as shown by tools such as CACHECA [18].

We transfer this intuition concerning repetitiveness and naturalness of software to identifier naming: for clarity and readability, programmers will tend to choose the same name in the same context! Thus arises our central claim: to guess the clear, natural name ν of an obfuscated identifier ω, all we must do is to guess *the most likely name given the context* κ *in which in the obfuscated identifier* ω *appears.* In other words,

$$\nu = \text{clear_name}(\omega) = \underset{\alpha}{\text{argmax}}\, p(\alpha \mid \kappa) \quad (1)$$

Given a context κ, we choose that name α which maximizes the conditional probability as shown. Given that programmers are *un*creative in their choice of variable names, and if we have a big enough corpus, then we can estimate a decent model that gives reliable scores $p(\alpha \mid \kappa)$.

Our tool, JSNAUGHTY, realizes this intuition via two complementary mechanisms: *a*) the existing JSNICE tool [42], which conditions "natural" names on semantic properties of the minified code, using static analysis; and *b*) AUTONYM, which uses an off-the-shelf statistical machine translation tool, and is comprised of two parts: b_1) a *distortion* model that captures exactly how the minifier changes names, and b_2) a *language* model that captures how programmers choose to write code (including variable name choices). We use the standard natural language translation model from MOSES [27] to estimate these probabilities from a large, matched corpus of clear and minified names, as we describe in more detail below in Section 3.

This paper makes the following contributions:

- We present a novel approach to automatically recover identifier names from minified (JS) code: training an off-the-shelf statistical translation model (MOSES), originally designed for natural language translation, on a large matched corpus of clear and minified code. We show that even this simple, language-independent approach successfully recovers a non-trivial number of the names used in the original un-obfuscated code.
- We show that a simple bit of tuning on the training data, to account for JS syntax, and some post-processing, to manage scoping, improves performance substantially. With these two simple steps, the performance of this MOSES-based approach, which we call AUTONYM, matches that of JSNICE.
- Finally, we observed that JSNICE and AUTONYM are actually quite complementary, each often performing well when the other fails; this led us to offer to the community the JSNAUGHTY tool, which is an opportunistic blending of the two; JSNAUGHTY performs substantially better than either of its constituents.

To our knowledge, JSNAUGHTY is currently the *best-performing* available tool to recover the original names from minified JS names; in addition, our work illustrates how, with some slight tuning of the training data to account for language specifics, and some simple post-processing, an off-the-shelf statistical NLP tool can be quite useful in software engineering applications. Readers are welcome to try JSNAUGHTY online at http://jsnaughty.org.

2 BACKGROUND

AUTONYM uses a statistical machine translation (SMT) approach to recover the original names from the contrived ones that UGLIFYJS inserts. Our basic approach is independent of programming language, but performance can be improved through scoping analysis, the details of which we present below, in Section 3.

2.1 SMT 101

SMT is a data-driven approach to machine translation, based on statistical models estimated from (large) bi-lingual text corpora; see Ochs *et al.* [38] for a good introduction. SMT is widely used in services like Google Translate. In SMT, documents are translated according to a probability distribution $p(e|f)$ that a string e in the target language (say, English) is the translation of a string f in the source language (say, Finnish). As per the Bayes theorem, the probability distribution $p(e \mid f)$ can be reformulated as

$$p(e \mid f) = \frac{p(f \mid e)p(e)}{p(f)}, \quad (2)$$

and the best output string e_{best} given an input string f is

$$e_{\text{best}} = \underset{e}{\text{argmax}}\, p(e \mid f)$$

$$= \underset{e}{\text{argmax}}\, \frac{p(f \mid e)p(e)}{p(f)} \quad (3)$$

$$= \underset{e}{\text{argmax}}\, p(f \mid e)p(e) \qquad \text{(for a given } f\text{)} \quad (4)$$

Note above, that once a Finnish sentence f is given, the marginal term in the denominator of (3) is fixed, and we can just maximize the numerator as in (4).

This formulation of the translation problem is called the "noisy channel" model; intuitively, we think of Finnish as a "noisy distortion" of English, and attempt to recover the most likely English sentence that would have resulted in the Finnish sentence. As per (4), there are two parts to the SMT model: a *translation model*, which captures how English sentences can be "noisily distorted" into Finnish ones ($p(f \mid e)$) and a *language model*, ($p(e)$) which captures the likelihood of different types of English sentences. Hence, the problem of estimating $p(e \mid f)$ can be decomposed into two subproblems, estimating a translation model $p(f \mid e)$ and a language model $p(e)$. The connection to name "de-minifying" is evident: just as SMT is used to "de-noisify" and "de-distort" Finnish back to English, one could use SMT to remove the "noise" of minified variables and recover their original form.

The language model, often based on n-grams, is estimated from a text corpus in any single language using fairly straightforward maximum-likelihood methods [30]. The translation model can be estimated from parallel data using the expectation–maximization (EM) algorithm [14]. Such parallel data matches text in source and

target languages,[3] typically aligning matched text fragments at sentence level [39]. Consequently, each sentence in one language is matched to its corresponding translation in the other language. Then, a *phrase-based* translation model [31] is typically estimated. In principle, one could estimate a word-to-word translation model, which captures the probability of each word in the source language being related, by translation, to a specific word in the target language. However, by including phrases (sequences of words) in the translation table, one can both use local context for disambiguation and also capture local reordering, which together substantially improves translation quality [28]. There's an accuracy-performance tradeoff here: models trained with more data and longer phrase lengths (in the extreme, entire sentences or, in principle, even entire documents) are more accurate, but also more resource-intensive, slower, and more prone to overfit. For example, Moses [27], the SMT toolkit we use, by default uses phrases no longer than 7 words in its translation model, unless specified otherwise.

In natural language translation, matching phrases needn't have the same length (*e.g.*, the French "à la maison" translates to the English "home"), and comprising words needn't be in the same order (*e.g.*, the English "I want to go home" translates to the Dutch "Ik wil naar huis gaan", with the verb now at the end). Therefore, in practice [28] the translation model $p(f \mid e)$ is extended to include a "relative distortion" probability distribution that penalizes too much reordering, and a factor to calibrate the output length, usually biasing longer output. Finally, the translation model used by popular SMT systems such as Moses [27] (the basis for Autonym) also includes a lexical weighting component which validates the quality of a phrase translation pair by checking how well its words translate to each other. For example, the lexical weight of the word pair ("maison", "home") from the French–English matching phrases above can be computed as the relative frequency of "maison" among all possible translations of "home" into French.

2.2 SMT for Name Recovery

Statistical machine translation is surprisingly well-suited for the problem of recovering original names from minified ones. As discussed above, while translating between natural languages, SMT needs to be very much aware of context. Thus the word "bark" in the two phrases "the dog's bark" and "the tree bark" would have to be translated quite differently; a modern SMT tool is very capable of resolving such contextual ambiguities and making the right word choice based on context. Why? Fortunately, there is a high degree of repetitiveness, both within a single natural language, as well as in the translation processes between languages. Given sufficient training data, SMTs are very capable of capturing and efficiently exploiting contextual regularities to provide nuanced, correct translations.

As it turns out, there is a *great deal* of predictable repetition in software source code, in fact much more so than in natural language corpora [20, 24]. As mentioned earlier, programmers write code to be read by other humans; code is harder to read than natural language, so arguably, readability is an even stronger imperative to consider when writing code. As noted earlier, identifier names are

[3]E.g., the simultaneously translated multilingual European Parliament proceedings are a valued resource.

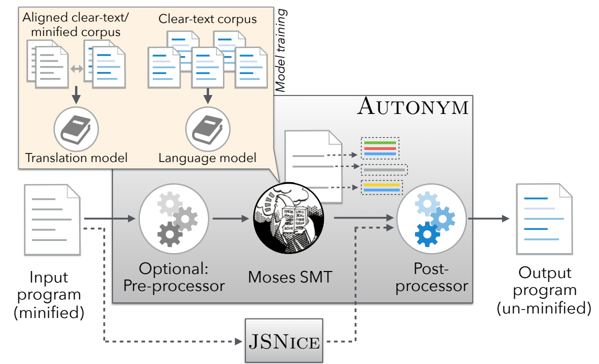

Figure 1: Overview of our approach.

chosen to be natural, unsurprising, and well-suited to local context, in much the same way that word choices in natural language reflect context. Thus, even though tools like UglifyJS reduce many different identifiers to cryptic single letters like `t` or `x`, they do so in very systematic ways, based on context; and thus, given sufficient data, SMT based tools are very well-suited to capture and exploit this through language and translation models.

A recent tool, JSNice [42], aims to recover minified variable names based on a similar conceptual stance. However, rather than using SMT directly on the source code, they rely on semantic analysis, and posit that a learned distribution of variable name choices, conditioned on semantic dependencies of that variable, would be sufficiently informative to recover suitable names. They use conditional random fields to model the name distribution. Their approach requires an analysis of the semantic dependencies in the code; our approach only requires variable scoping information, and recovers about as many variable names using an off-the-self SMT tool (Moses); we also find that their approach is quite complementary to ours in performance, leading to a synergistic combination that outperforms both, as we describe below.

3 APPROACH

We now motivate and present the technical details of our Autonym approach. We present the blending of Autonym and JSNice later, in Section 3.4.

To use SMT in any setting, we first have to train the models, and then use the trained models to perform the translation task. Our approach relies on using a JS minifier (UglifyJS) to generate large amounts of matched (clear—minified) pairs of training data, and then training an off-the-shelf SMT system (Moses [27]) on these pairs (Figure 1). Once trained, our system Autonym accepts JS code as input and tries to recover clear, *natural* identifier names as output, by "translating" (using Moses' built-in decoder [31]) the minified code into clear code, much the same way one would translate, say, French to English.

3.1 Translation Challenges

Unlike natural language texts, which are inherently polysemous and ambiguous and, therefore, more amenable to approximate translations, computer programs have strictly defined semantics that must be exactly preserved during translation. So how can a statistical

```
1  var geom2d = function() {
2    var sum = numeric.sum;
3    function Vector2d(x, y) {
4      this.x = x;
5      this.y = y;
6    }
7    mix(Vector2d, {
8      dotProduct: function dotProduct(vector) {
9        return sum([this.x*vector.x, this.y*vector.y]);
10     }
11   });
12   function mix(dest, src) {
13     for (var k in src) dest[k] = src[k];
14     return dest;
15   }
16   return{
17     Vector2d: Vector2d
18 };}();
```

(a) Original

```
1  var geom2d = function() {
2    var n = numeric.sum;
3    function t(t, n) {
4      this.x = t;
5      this.y = n;
6    }
7    r(t, {
8      dotProduct: function i(t) {
9        return n([this.x * t.x, this.y * t.y]);
10     }
11   });
12   function r(t, n) {
13     for (var r in n) t[r] = n[r];
14     return t;
15   }
16   return {
17     Vector2d: t
18 };}();
```

(b) After minification with UglifyJS

Figure 2: Example JavaScript program.

approach (with the inherent randomness), like the one Autonym uses, do this? The ***key insight*** comes from the structure of the problem we are trying to solve: layout and source code comments aside, minified JS programs are *alpha-renamings* of their clear-text counterparts; therefore, the probability of any language construct being a translation of anything other than itself is zero by construction, *i.e.*, the program's structure can be trivially preserved.[4] Even so, there are still two challenges with using SMT for "translating" source code, minified JS in particular, which we detail next. Our adaptations addressing these challenges are implemented in the pre- and post-processor components depicted in Figure 1.

Inconsistency. The context captured by multi-word *phrases*, as opposed to single words is paramount to the effectiveness of SMT (recall the example pairing "bark" with "tree" or "dog"). However, while each occurrence of "bark" can be independently translated in a natural language document, depending on what is most appropriate for its context, the same cannot be said about source code identifier names. Consider the example JS program in Figure 2. Indeed, renaming different occurrences of the same identifier differently (*e.g.*, the two occurrences of `n` on lines 2 and 9 in the minified program in Figure 2b, both corresponding to `sum` in the clear-text version in Figure 2a) would alter the program's semantics. There is ***no inherent mechanism in phrase-based SMT to ensure consistency of translation for the same identifier name***. This is especially true since phrases captured by the *translation model* component would likely not span the entire program (recall the accuracy-performance tradeoff discussed in Section 2); indeed, by default, Moses translates each line independently. We describe our solution to this challenge in Section 3.2.

Ambiguity. While source code is unambiguous, and admits no ambiguity in reserved language keywords (`for` is always `for`), across a large corpus identifiers could still be named differently despite having similar context. So far, this is not much different from natural language, where words can have multiple meanings. However, JS minification carries an additional complication: it is possible to "overload" the same minified name by reusing it within several different scopes. An extreme version of this occurs when a line like "`function Vector2d(x, y) {`" is minified as "`function t(t, n) {`" (line 3 in Figure 2). This is entirely legal; `t` is simultaneously a function and a parameter to the function. Indeed, this type of name overloading amplifies obfuscation, and makes the program even harder to read.

There are two consequences. First, this presents a challenge for name recovery. It's very unlikely that real programmers overload the same name in this confusing manner. If we were to recover the same name for both occurrences of `t` (a straightforward solution also to the inconsistency challenge above), while certainly correct, it would not result in *natural* names and reusable code. Therefore, ***post-translation renaming should be attempted only within scope***; this overloading is a peculiarity of JavaScript, and is the only step that makes our approach language-dependent.

Second, high-performance statistical NLP methods, SMT included, all rely on large training corpora. In our case, we train Moses' translation model on a large corpus of *aligned* clean—minified JS source files (Figure 1). Therefore, in addition to the inherent ambiguity resulting from, say, different parameter names being minified to `t` across our corpus, this peculiarity of minifying JS means that different function names are also minified to `t` when such name overloading occurs. As a result, one has to choose the most likely clear-text name (that would have resulted in the `t` minification) among many possible alternatives. ***The more a minified name is reused, the less useful its context becomes in the training corpus***. We describe our optimization addressing this challenge in more detail in Section 3.3.

3.2 Resolving Inconsistencies

The Moses SMT tool (see Figure 1) reads a minified program, line by line; for each minified line l_m, Moses uses a conditional distribution over "clear lines" $p(l_c \mid l_m)$ to produce a candidate set of possible translations. The number of suggested translations per line can be set as a parameter. Furthermore, even without configuring Moses to provide multiple, instead of single, suggestions per line, different suggestions for the same identifier occurring on different input

[4] More sophisticated (structure altering) types of JavaScript compression, such as those introduced by Google's Closure compiler https://developers.google.com/speed/articles/compressing-javascript, are beyond the scope of this work.

lines may still ensue, since lines are translated independently. For example, Moses might suggest that the line `r = t(r)` in minified code be rendered as `setA = conv(setA)`, and also suggest that the next line, *e.g,* `s = t(s)` be rendered as `setB = redup(setB)`, thus providing two choices to unminify the identifier `t`, either `conv` or `redup`. Therefore, for each minified identifier, we must choose precisely one unminified "translation" from the available suggestions, to ensure name consistency.

Algorithm 1 describes the process by which we recover (in the Postprocessor component seen in Figure 1) clear names from minified ones, addressing the inconsistency challenge described above. There are two subroutines. First, the procedure ComputeCandidateNames takes a tokenized input file, and uses Moses to generate a translation for each line (line 2). Then, it collects all possible renamings suggested by Moses for any given minified name (lines 3–6). However, as discussed above (recall the "`function t(t, n) {`" example), we need to address the name-overloading ambiguity challenge: renaming both instances of `t` to the same name may result in unnatural renamings, since one is a function name and the other is a parameter. In order to preserve program semantics, we need only ensure that distinct variable names are retained *when they exist in the same scope.* Therefore, in the aforementioned example, function `t` and parameter `t` (line 3 in Figure 2) will have different candidate sets, and thus can be renamed differently, because they are not in the same scope.

Second, the procedure RankCandidates takes a set of candidate renamings for a minified name, and ranks them based on how well they fit the context where they are to be used, *i.e.*, how natural they are. For each candidate, we use the same language model used by Moses to score translations (recall the two SMT components, translation model and language model, from Section 2) to compute a "naturalness" score for each corresponding minified line, after temporarily renaming the minified name to that candidate (lines 8–10). We exhaustively try all candidates and rank them by their average naturalness score (line 11), computed across the different lines implicated. We also experimented with other ranking schemes, detailed below.

The main procedure PickNames (lines 12–19) iterates over all minified names and chooses the highest ranked candidate for each. Since a candidate could have been suggested for different minified names in the same scope, the traversal order matters. For example, assume two same-scope minified names `n` and `t`, with candidates `{src}` and `{src, dest}`, respectively (*e.g.*, line 12 in Figure 2). Since a candidate cannot be used multiple times in the same scope, choosing `src` for `t` before considering `n` would invalidate `src` as a candidate for `n` in a next step. But which names should one give priority to? We base our traversal on two criteria: (1) we assume that more frequent names in the input, *i.e.*, those appearing on more lines, are more important for program comprehension and should be considered first; (2) we traverse names with smaller candidate sets first. Consequently, we sort all minified names by line frequency, descending (line 14), then traverse all minified names with a single candidate translation (lines 15–16) before minified names with larger candidate pools (lines 17–19). In the example above, we would first rename `n` to `src`, since `t` has more candidates

Algorithm 1 Choose consistent names

1: **procedure** ComputeCandidateNames(*input*)
2: Translate *input* using Moses. Assert translated lines have as many tokens as originals.
3: **for all** *line* ∈ Moses output **do**
4: Parse line.
5: **for all** minified *name* ∈ matching input line **do**
6: Record Moses suggestion as *candidate* renaming. Record line number.

7: **procedure** RankCandidates(*candidates*)
8: **for all** *candidate* ∈ *candidates* **do**
9: Select all affected minified lines and temporarily rename minified *name* to *candidate*.
10: Use language model to compute log probability for each line, after renaming.
11: Sort candidates by average log probability across all affected lines, descending.

12: **procedure** PickNames(*input*)
13: ComputeCandidateNames(*input*)
14: Sort minified names by how many lines they appear on in *input*, descending.
15: **for all** *name* ∈ *input* with single *candidate* **do**
16: Rename to *candidate* if not already chosen elsewhere in same scope. Keep minified name otherwise.
17: **for all** remaining *name* ∈ *input* **do**
18: RankCandidates(*candidates*)
19: Rename to top *candidate* if not used elsewhere in same scope. Keep minified name otherwise.

to choose from. If, during this process, a minified name remains without any feasible candidate, we leave it unchanged.[5]

Alternative Ranking Schemes. There are other ways to select a renaming from a set of candidates, besides the one we presented above (language model). We describe two additional variants of the RankCandidates procedure that we experimented with.

Frequency-based. A straightforward approach is to choose the renaming option that is suggested for the greatest number of lines on which the minified identifier appears (ties can be broken, *e.g.*, by name length). This is best explained through an example. Recall the two potential choices to unminify the identifier `t` above, either `conv` or `redup`, collected across different minified input lines where `t` appears. In the frequency-based ranking, if `t` is rendered as `conv` 5 times and `redup` only twice, we would prefer `conv`. This technique is simpler (and faster) than using the language model, so it might be preferred in practice if it performs well.

Feature-based. A more general approach is to extract features from both the context and the suggestions themselves, and learn a classifier to rank candidates using a weighted combination of these features. Many features could potentially be useful: How long is the name in characters? Is it only a single character?[6] Does the name use camel case, underscores, or the dollar sign? We can also take advantage of the language model to extract additional

[5] Or we rename to an artificially suffixed minified name, when the minified name itself had been the top candidate for something else in the same scope earlier (very rarely).
[6] Single character names are uncommon outside of iterators, and should likely be avoided in many other contexts.

features: What's the average log probability of all the implicated lines, after replacing the minified name with the candidate, *i.e.*, how well does the candidate fit the context, on average?[7] What's the maximum gain in probability across all implicated lines, relative to the minified file, *i.e.*, does the candidate fit *any* minified line particularly well? In addition to these *suggestion features*, there are features that apply to the context where we are making the suggestion: How many lines does the minified identifier appear on? What's the maximum number of times the name appears on any single line? Does the name appear on lines with literals or de-minifiable names (*i.e.*, globals)? Does the name appear on a line with a loop statement? We will call these *context features*.

To determine which (if any) of these features might be useful and what the weighted combination should be, we built logistic regression models in R, on a set-aside tuning dataset of clear text JS files that had been minified by UglifyJS and de-minified by Autonym. The criterion for evaluating a regression model was the number of original names correctly recovered. We tried to answer the question: *when the desired renaming is among the translation candidates, what is a weighted combination of suggestion and context features that maximizes the probability of ranking it as the top candidate?* Therefore, in our tuning sample, we only considered suggestion lists that contained the original de-minified name. Furthermore, we selected exactly one incorrect renaming at random from each such list of (potentially many) candidates, as imbalanced data can lead to poor models. We also assessed multicollinearity using the variance inflation factor and excluded predictors accordingly. Finally, we pruned the tuning data to remove suggestions with outlying behavior in their features, which would act as high leverage points in the regression models. Equation 5 lists the features we selected for our logistic model that performed best in the prediction task on the tuning data.

$$\begin{aligned} score = \underline{\alpha}_1 \cdot log(\#linesSuggestedFor) \\ + \underline{\alpha}_2 \cdot (length > 1) + \underline{\alpha}_3 \cdot (averageLogProbDrop) \\ + \underline{\alpha}_4 \cdot (AverageRawProb) + \alpha_5 \cdot (usesCamelCase) \\ + \alpha_6 \cdot (useUnderscores) + \alpha_7 \cdot (contains\$) \end{aligned} \quad (5)$$

We found that while including the context features as controls resulted in better model fit, excluding them led to better predictive performance. Similarly, experimentation showed that while some features were useful as controls, including their coefficients in the ranking score calculation diminished accuracy in recovering the original names in the tuning set. The coefficients included in our final ranking function are underlined: the log of the frequency of the suggestion, measured in lines; the average log probability across implicated lines; the average change in log probability between the minified and renamed implicated lines; and whether the candidate name is a single character or not.

3.3 Optimization: Reducing Ambiguity

We described in Section 3.2 how scoping information can help to resolve ambiguities caused by overloaded minified names, *e.g.*, when a line "`function Vector2d(x, y) {`" is minified as "`function t(t, n) {`" (line 3 in Figure 2). To amplify this association between context and names, we artificially (but consistently) rename minified names in each scope *before feeding the code to* Moses (in the Preprocessor component in Figure 1), so that such name overloading is eliminated already *prior to decoding*. Moses can then more easily provide distinct translations. Of course, this means that Moses must be trained on data that consistently renames variables according to scope in the same way.

Table 1: Examples of context tokens that make up our hash renamings for some minified names in Figure 2. Note that function `r` (line 12) does not get renamed because its hash has been used once by function `t` in the same scope (line 3).

Minified	Original	Location	Hash renaming
`n`	`sum`	line 2	SHA1("`var#=numeric.sum;`")
`t`	`Vector2d`	line 3	SHA1("`function#(,){`")
`t`	`x`	line 3	SHA1("`function(#,){`")
`r`	`mix`	line 12	`r`
`r`	`k`	line 13	SHA1("`for(var#in)[]=[];`")

One can readily imagine many such possible context-aware renaming strategies. One strategy we experimented with is to compute a SHA1 hash of *all the tokens on the definition line*[8] *of a name, that remain unchanged during minification*, *i.e.*, all language constructs and global names. Table 1 illustrates the context tokens that feed the hashes, for some of the minified names in Figure 2. Note that all local names have been stripped away and the minified name itself has been replaced with the # placeholder. We also make sure not to overload hashes in the same scope (see the example of function `r`). In practice, this renaming scheme increases the vocabulary size (unique names) in our corpus by approximately 250%, *i.e.*, it is quite effective at disambiguating!

3.4 Blending Autonym with JSNice

In order to blend Autonym and JSNice, we simply updated the ComputeCandidateNames procedure in Algorithm 1 to record, for each minified name in the input, the JSNice renaming as an additional suggestion to the ones offered by Moses. This means that the rest of our algorithm for choosing consistent names remains unchanged: the renaming candidates, which now include also the JSNice renaming, are ranked using the different schemes we presented above in Section 3.2; the top ranked candidate is chosen as the final renaming. We also tried building logistic models to explicitly discriminate between when to choose the name selected by JSNice vs Autonym using the *suggestion* and *context* features described earlier, but found the models extremely poor. Thus, JSnaughty uses the simple blending scheme described here.

4 EVALUATION

To evaluate Autonym, we train and tune a Moses model on a joint corpus of minified and clear-text JavaScript code from GitHub; then measure the *accuracy of retrieving the original, un-minified names* on a set-aside testing corpus.

[7] This is exactly the "naturalness" score ranking criterion in Algorithm 1 above.

[8] We use information stored in the minifier's internal representation of the abstract syntax tree to infer which line is the definition line for a name. Note that the definition line is not always the line of first occurrence; *e.g.*, function `r` (originally `mix`, line 12) is used on line 7, before it is defined.

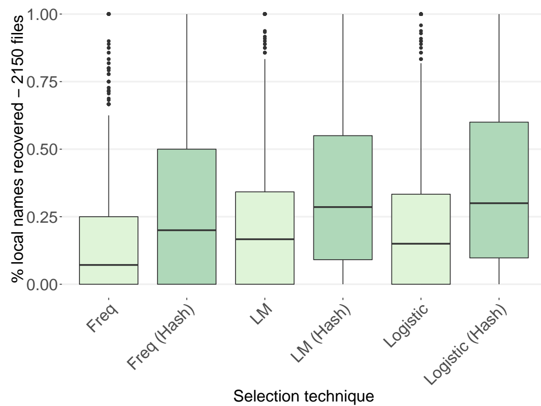

Figure 3: File level accuracy of our three name selection techniques, with and without the optimization.

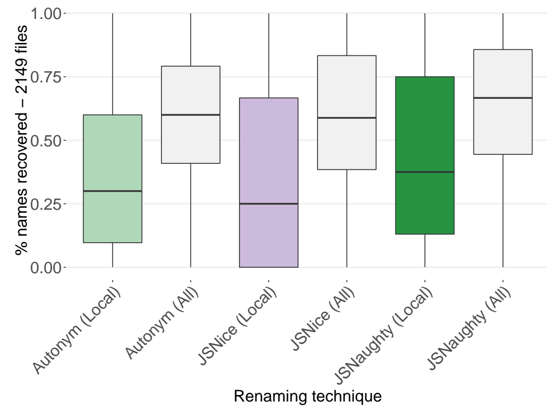

Figure 4: File level accuracy for our AUTONYM tool, JSNICE, and our blended tool, JSNAUGHTY, which substantially dominates the state-of-the-art JSNICE.

This evaluation strategy, also adopted by JSNICE [42], was chosen primarily due to its practicality, *i.e.*, it can be fully automated. Note, however, that failure to recover the original identifier names is not necessarily a failure of the method, as the suggested names could actually be "better" than the originals. An extensive human evaluation of the quality of the recovered names, for different attributes such as readability, and while controling for the provenance of the training data (*e.g.*, different open-source projects have different naming conventions and style guides, which could influence performance) is beyond the scope of this work.

4.1 Experimental Setup

Corpus. We first identified the oldest 100,000 non-fork JavaScript repositories using GHTORRENT [22] and cloned them from GITHUB, yielding 5.3 million files with a `.js` extension. Next, we removed duplicate files (*i.e.*, having the same SHA1 hash), leaving 1.4 million unique files; from these we randomly sampled to generate the following non-overlapping sets: 300,000 files for translation model training, 1,000 files for MOSES hyper-parameter tuning, and 10,000 files for logistic regression estimation. We further used heuristics and the minifier UGLIFYJS to determine which files are already minified, and we excluded them from our samples: first, we excluded files that remained unchanged after minification; second, since not all minified files remain unchanged, *e.g.*, files minified by a different minifier could still change, we used additional heuristics based on the distribution of identifier name lengths. Few other files that failed to parse during scope analysis (we reuse the scope analysis information that UGLIFYJS collects during minification, rather than writing our own scope analyzer) were discarded as well.

In the end, the different filters shrunk our samples to: 227,233 files in the MOSES training set, 659 in the MOSES tuning set, and 5,825 in the regression estimation set. In addition, we randomly sampled 500,000 files to train the language model (no overlap with the tuning and test sets). Finally, we randomly sampled 2,150 parseable and non-minified files to form a held-out testing set. Tuning and test files had 100 lines or less (the median file pre-sampling had 58 lines, *i.e.*, our sample is representative). The training and test sample sizes are comparable to those used by JSNICE [42].

Training. Recall the two components to a MOSES-based translation system that need to be trained: the *translation model*, estimated from a parallel, sentence-aligned corpus in the source and target languages, and the *language model*, estimated from a standard corpus in the target language.

Building our parallel JS corpus was straightforward: we passed all files in the training sample through the minifier UGLIFYJS[9] and scanned the output to ensure the program structure (and alignment) was preserved. We then trained two standard translation models using the built-in MOSES `train-model.perl` script, one for the default case without the ambiguity reduction optimization in Section 3.3, and one with the context-aware hash renaming implemented. The only adjustment to the default settings was that we allowed phrases of length up to 20 to be included in the phrase table (default is 7). For the language model, we estimated a KenLM [23] five-gram model with modified Kneser-Ney smoothing [9], after pruning singletons above order three.

Tuning. After training, MOSES assigns default weights to the different components that make up its translation model (recall the discussion in Section 2.1). Three[10] of these components are relevant to our discussion here: the *phrase table* component, which contributes a probability of two "phrases" (sequences of consecutive tokens) being translations of each other (*e.g.*, how likely is it that "`function t(t, n) {`" is a minified version of "`function Vector2d(x, y) {`"?); the *language model* component, which contributes a probability of the output being being "natural" JS; and the *lexical weighing* component, which checks how well tokens translate to each other (*e.g.*, how likely is it that `t` is a minification of `Vector2d`?). We used the built-in MOSES `mert-moses.pl` script with default settings on the hyper-parameter tuning set to tune the different weights given to each component, separately for each of the renaming techniques we tested.

Evaluation Criterion: Precision. For each JS file in our test set, we minify it ourselves, then use AUTONYM (after training and tuning) to recover the original names. We additionally run JSNICE

[9] We used version 2.7.3, with the `-m` (mangler) parameter.
[10] Trivially, token reordering does not occur in our setting.

on the same minified files, for comparison. We used the publicly available implementation of JSNice from the `unuglify-js` NPM package,[11] which is a client of the Nice2Predict[12] framework.

Our performance metric is the percentage of original, pre-minification *local* identifier names that each technique recovers, per file; *global* names can be trivially recovered, since only local names are renamed during minification. Our test files contain between 1 and 75 local (minifiable) names (mean 10.6, median 8); or between 1 and 87 total names, both local and global (mean 15.5, median 13).

4.2 Choosing Consistent Names

The first set of results we present pertains to the strategy used to choose a consistent renaming from among a set of candidates. We implemented all three strategies presented in Section 3.2: our default language model based approach ("LM"), the straightforward frequency-based approach ("Freq"), and the more general feature-based logistic regression approach ("Logistic"). Figure 3 presents the distributions of precision scores on the test set for the three approaches, in light green.

We observe that all three distributions are right skewed, *i.e.*, all three methods have limited precision. LM performs best; it retrieves a median 17% of the local names in our test files, and it statistically significantly outperforms Freq (paired Wilcoxon signed rank test, $p < 0.0001$), with a medium effect size[13] ($r = 0.32$). LM and Logistic are statistically indistinguishable (paired Wilcoxon signed rank test, $p = 0.99$), *i.e.*, the more general feature-based approach does not provide additional gains over our default language model based approach; this is not particularly surprising, as we have tuned the logistic regression coefficients only on data pre-processed with the context-aware hash renaming strategy discussed above in Section 3.3.

4.3 Capturing Context Using Hash-based Pre-renamings

The second set of results we present, visualized in dark green in Figure 3, pertains to the effectiveness of the context-aware hash-based renaming we apply as a pre-processing step. We observe that the model trained on a hash-based renamed corpus consistently outperforms the standard (not preprocessed) one, with a between 12–15 percentage points increase in the median number of local names recovered. All differences between precision with and without hashing are statistically significant (paired Wilcoxon signed rank test, $p < 0.0001$), with medium effect sizes ($r = 0.4$ for Freq; $r = 0.38$ for LM; and $r = 0.4$ for Logistic).

The best performing model, "Logistic (Hash)" uses the more general feature-based logistic regression approach to rank candidate names, as well as artificially renaming local names to a hash of their context tokens (Section 3.3) prior to translation. Naturally, the training corpus must be preprocessed similarly. The interquartile range of precision on *local* names is 10–60%, with a median of 30%. We select this model for the subsequent comparison to JSNice.

[11] https://www.npmjs.com/package/unuglify-js
[12] http://www.nice2predict.org
[13] We report the r measure proposed by Field [16] as an alternative to Cohen's d for non-normal distributions. We interpret r using the rules of thumb suggested by Cohen, with suggested thresholds of 0.1, 0.3, and 0.5 for small, medium and large magnitudes respectively.

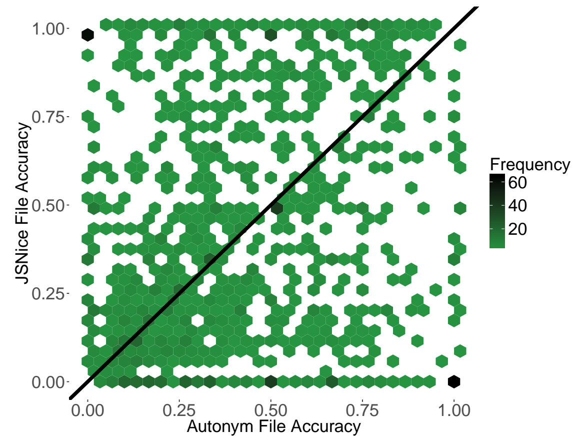

Figure 5: Hexbin scatterplot comparing file level accuracy for Autonym and JSNice. Excludes files where both techniques fail or succeed completely.

4.4 Comparison to JSNice

The results from the comparison between Autonym (the best performing "Logistic (Hash)" variant) and JSNice are depicted in Figure 4. As discussed, we perform the comparison of precision only on *local* names, which get minified by UgligyJS and need recovery. We also present (light gray background boxplots) the precision results on all names, local and global (the latter don't need any recovery), to be consistent with the original presentation of JSNice [42].

We observe that the static analysis based JSNice and our simpler, SMT (Moses) based Autonym perform comparably. The precision values for JSNice are more dispersed: the interquartile range is 0–67%, with a median of 25%; in contrast, Autonym has higher median precision, 30%, but the distribution is more concentrated (IQR 10–60%). Formal testing confirms our visual observation: there is no significant difference in medians between the two distributions of precision values (paired Wilcoxon signed rank test, $p = 0.76$).

4.5 Blend between Autonym and JSNice

The previous comparison suggests that Autonym and JSNice, despite having similar performance, behave quite differently. Figure 5 displays a hexbin scatterplot comparing the file level precision of Autonym and JSNice at recovering originals from minified local names. Points below the main diagonal correspond to files where Autonym outperforms JSNice; points above the main diagonal correspond to files where JSNice outperforms Autonym. Analyzing the scatterplot enables us to confirm the earlier intuition: the two techniques seem quite complementary, with one often succeeding when the other fails, and vice versa. This led us to propose JSNaughty, a straightforward blend between Autonym and JSNice, which adds the renaming proposed by JSNice to the candidate pool proposed by Moses; JSNaughty uses the logistic regression based approach to rank candidates (now one extra) described above.

To assess how well does the blend performs we turn our attention back to Figure 4, where the last pair of boxplots depicts the distribution of file-level precision values for JSNaughty on local and, for consistency with the original JSNice presentation [42], all

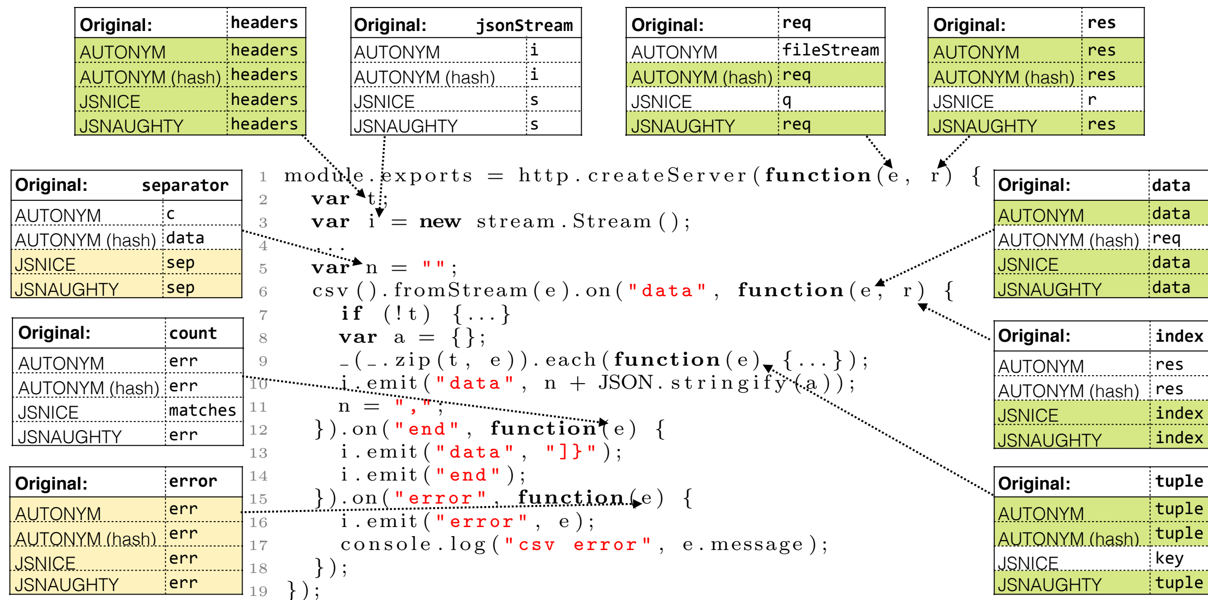

Figure 6: Example minified test file, with names suggested by Autonym with and without the hash based optimization, JSNice, and JSNaughty shown alongside. Exact matches are highlighted green; approximate matches yellow.

names. We observe that JSNaughty clearly dominates both of its constituent approaches: it recovers a median 37% (mean 44%) of the local names in a file, with an interquartile range of 13–75%. Considering all names in a file, both local and global, JSNaughty recovers a median 67% of names, with an interquartile range of 44–86%. The performance improvement is non-trivial (paired Wilcoxon signed rank test, $p < 0.0001$): JSNaughty performs significantly better than both JSNice ($r = 0.23$) and Autonym separately ($r = 0.18$), with a small effect size. Although the effect size is small, given the experimental finding that JSNaughty does a substantially better job at name recovery than the best available alternative, it would make little sense for practitioners to not use our approach.

4.6 Illustration

We illustrate the complementary nature of Autonym and JSNice, which inspired JSNaughty, with the example in Figure 6, from which we draw the following observations:

- Some minified names are exactly reverted to the original names by Autonym but not JSNice (*e.g.*, parameters `e` and `r` on line 1, restored to `req` and `res`, respectively), while other names are exactly reverted by JSNice but not Autonym (*e.g.*, parameter `r` on line 6 restored to `index`).
- Both techniques sometimes suggest approximate matches (*e.g.*, for `n` on line 5, `sep` instead of `separator`; also `e` on line 15) which, although different than the originals, capture the same spirit. This reiterates the need for a future, carefully controled human study to evaluate the quality of the suggested names beyond our automated approach.
- Some names are exactly recovered by all methods (*e.g.*, `t` on line 2), while others are not reverted by any (*e.g.*, `i` on line 3).

5 RELATED WORK

5.1 Obfuscation & Deobfuscation

Code obfuscation is a general topic, spanning a wide range of goals: in addition to intellectual property protection [15], applications include *inter alia*, tamper-resistance [4, 44], and water-marking [12]. Obfuscation is also used in malice, to evade detection by software defensive mechanisms [6]. There are some negative theoretical results by Barak *et al* [5] concerning the possibility of code obfuscation in general; still, obfuscation has durable commercial and intellectual charms; a constant stream of new techniques, papers, and patents has continued over the years, despite Barak *et al.*

More specifically, for JavaScript, there are some idiosyncratic constraints that impose some natural limits on excessive use of obfuscation. First, with the current browser ecosystem, code must be shipped in source code form. Second, typical JavaScript programs use quite a few APIs; the API calls must use the correct name, which appears in the clear in the source code. Third, in general, run-time efficiency is a concern, so that obfuscations that require a run-time overhead are undesirable. Likewise, obfuscations that increase code size are undesirable, due to impact on bandwidth use—indeed, "small is beautiful" applies. For all these reasons, code minifiers are a good trade-off: they are simple, easy to use, and make the code smaller and difficult to read, without affecting performance. UglifyJS is thus very popular: for the majority of potential adversaries, the output of UglifyJS provides a sufficient deterrence.

Turning to *de*-obfuscation, a key focus in reverse engineering and deobfuscation is on static & dynamic analysis techniques [11, 13]. This has great relevance for malicious code detection [10, 32], and has received a great deal of attention. Given the constraints on obfuscation use for JavaScript, these approaches are ill-suited to the most common use case, which is to recover full, natural, identifier

names which are corrupted by minifiers like JavaScript. Code analysis tools generally focus on the *semantics* of obfuscated programs, to recover intent; however, what most JavaScript programmers need is a way to make minified programs easier to read, with natural, well-suited identifiers that promote human understanding. Thus, in this setting, a statistical approach, which helps make minified programs look "familiar", *viz*, textually similar to most JavaScript programs that do the same thing, is precisely what is needed; thus SMT techniques, trained over large corpora are specially well-suited.

5.2 The Naturalness of Software

Gabel & Su observed [20] that most short code sequences are not unique; following this work, Hindle *et al* [24] showed that statistical language models were just as effective (in fact more so) for software as for natural language corpora, thus suggesting that software is also natural. Language models are central to the great success of NLP techniques in speech recognition, translation and so on; thus Hindle *et al*'s work suggests great promise for the use of language models in code. There have been substantial further applications of statistical models for code, in areas such as coding standards mining and checking [1], code summarization [17], idiom mining [3, 37], and bug localization [41]. The key insight of this work is that *identifier names are natural*; *i.e.*, programmers choose "natural sounding" identifier names, in regular, predictable, repetitive ways that reflect the context of use, so as to convey a predictable, unsurprising intent to the reader. Thus, even though Javascript minifiers shorten variables to single letters, there is sufficient information in the context to predict which names make the most sense. Furthermore, even if minifiers might contrive to map many different names in different, (or overlapping) contexts to the same single-letter names, there is sufficient regularity in joint distribution of the context of both "clear" and "minified" context that allows us to get a good statistical prediction on what the unminified name should be. Allamanis *et al.* [2] use this idea to suggest more "natural" method and class names for otherwise unaltered code; we use it here to recover any/all minified identifier names in code subject to minification.

5.3 SMT in Software Engineering

There have been efforts to apply SMT to software engineering problems along the two directions below.

Migration. It is straightforward to imagine a potential use of SMT in software engineering: if programming languages are "natural," can we automatically translate between them the way we translate from English to French?

Nguyen *et al.* [34] were among the first to address this question, by experimenting with translation from Java to C#. The authors treat source code as a sequence of lexical tokens (each code token is the equivalent of a word; a method is the equivalent of a sentence), which enables them to apply a standard phrase-based SMT model [8] out-of-the-box. Empirical evaluation on a parallel corpus of around 13,000 Java-to-C# method translations, automatically mined from two open-source projects available in both languages, found the approach imperfect but promising: more than half of all translated methods were syntactically incorrect, yet users would not have to edit more than 16% of the total number of tokens in the translations in order to correct them. Based on their experiments, the authors advocate for more program-oriented SMT models instead of purely lexical ones. In follow-up work, they propose several such models aimed at migration of API usages [33, 35, 36].

Karaivanov *et al.* [25], as did Nguyen *et al.* [34], experimented with translation from C# to Java on a parallel corpus of around 20,000 C#-to-Java method translations mined from open-source projects available in both languages. However, they also trained hybrid phrase and rule-based SMT models that take the grammatical structure of the target language into account. Experimental evaluation on a sample of 1,000 C# methods confirmed that the approach is promising: SMT was sometimes able to learn how to translate entire methods, and map one set of API calls to another, especially with the more program-oriented models (roughly 60% of the resulting translations compiled). Still, the authors note that obtaining a parallel corpus of translated programs is challenging.

Documentation. Pseudo-code written in natural language can be a valuable resource during program comprehension for developers unfamiliar with the source code programming languages. Can we automatically translate source code into pseudo-code using SMT? Oda *et al.* [19, 40] experimented with generating English and Japanese pseudo-code from Python statements, reporting positive outcomes. The authors first created Python-to-English (18,000 statements) and Python-to-Japanese (700 statements) parallel corpora, by hiring programmers to add pseudo-code to existing source code. Then, they trained different phrase-based SMT models that vary in their level of program orientation, ranging from a purely lexical one to one that operates on modified abstract syntax trees (ASTs). Experiments showed that all models generate grammatically correct and fluent pseudo-code for at least 50% of the statements, with the more syntax-aware models performing better.

A related problem is generating descriptive summaries in natural language (*i.e.*, *docstrings*) for functions written in programming languages, using SMT. Cabot [7] reports on successful experiments with generating English docstrings for Python functions. He builds phrase-based SMT models on a parallel corpus of around 70,000 docstring-documented functions extracted from open-source projects: the phrases are the linearized ASTs extracted from these functions (input) and the English docstrings (output); alignment is performed using standard models [39].

6 CONCLUSIONS

Minified JavaScript code is ubiquitous on the web; the minified, overloaded names in these programs impede program understanding. Using statistical machine translation tools, with some post-processing (to both handle name scopes, and to choose the best available name consistently) we first built a tool that essentially equals the performance of the state-of-the-art tool, JSNice. During evaluation, we noticed that JSNice, and our tool, Autonym, perform well in different settings; so we then built an opportunistic mix of the two, JSNaughty, that statistically significantly beats both. Our tool is available online; based on its best-in-class performance, we expect it should be the preferred tool for developers seeking name recovery for minified programs found "in the wild".

ACKNOWLEDGEMENTS

BV, CC, PD are partially supported by the NSF grant 1414172.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proc. International Symposium on Foundations of Software Engineering (FSE)*. ACM, 281–293.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proc. 2015 Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 38–49.

[3] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proc. International Symposium on Foundations of Software Engineering (FSE)*. ACM, 472–483.

[4] David Aucsmith. 1996. Tamper resistant software: An implementation. In *Proc. International Workshop on Information Hiding*. Springer, 317–333.

[5] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2001. On the (im) possibility of obfuscating programs. In *Proc. Annual International Cryptology Conference*. Springer, 1–18.

[6] Jean-Marie Borello and Ludovic Mé. 2008. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology* 4, 3 (2008), 211–220.

[7] Michael Anthony Cabot. 2014. *Automated Docstring Generation for Python Functions*. Master's thesis. University of Amsterdam.

[8] Daniel Cer, Michel Galley, Daniel Jurafsky, and Christopher D. Manning. 2010. Phrasal: A Statistical Machine Translation Toolkit for Exploring New Model Features. In *Proc. NAACL HLT 2010 Demonstration Session*. Association for Computational Linguistics, 9–12.

[9] Stanley F Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In *Proc. Annual Meeting of the Association for Computational Linguistics*. ACL, 310–318.

[10] Mihai Christodorescu and Somesh Jha. 2006. *Static analysis of executables to detect malicious patterns*. Technical Report. DTIC Document.

[11] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 184–196.

[12] Christian S. Collberg and Clark Thomborson. 2002. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on Software Engineering* 28, 8 (2002), 735–746.

[13] Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Giacobazzi. 2006. Opaque predicates detection by abstract interpretation. In *Proc. International Conference on Algebraic Methodology and Software Technology*. Springer, 81–95.

[14] Arthur P Dempster, Nan M Laird, and Donald B Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (methodological)* (1977), 1–38.

[15] Premkumar T Devanbu and Stuart Stubblebine. 2000. Software engineering for security: a roadmap. In *Proc. Conference on the Future of Software Engineering*. ACM, 227–239.

[16] Andy Field. 2009. *Discovering statistics using SPSS*. Sage.

[17] Jaroslav Fowkes, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. 2014. Autofolding for Source Code Summarization. *arXiv preprint arXiv:1403.4503* (2014).

[18] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. 2015. Cacheca: A cache language model based code suggestion tool. In *Proc. International Conference on Software Engineering (ICSE)*, Vol. 2. IEEE, 705–708.

[19] Hiroyuki Fudaba, Yusuke Oda, Koichi Akabe, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Pseudogen: A Tool to Automatically Generate Pseudo-Code from Source Code. In *Proc. International Conference on Automated Software Engineering (ASE)*. IEEE, 824–829.

[20] Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *Proc. International Symposium on Foundations of Software Engineering (FSE)*. ACM, 147–156.

[21] Edward M Gellenbeck and Curtis R Cook. 1991. An investigation of procedure and variable names as beacons during program comprehension. In *Proc. Fourth Workshop on Empirical Studies of Programmers*.

[22] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: GitHub's data from a firehose. In *Proc. Working Conference on Mining Software Repositories (MSR)*. IEEE, 12–21.

[23] Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H. Clark, and Philipp Koehn. 2013. Scalable Modified Kneser-Ney Language Model Estimation. In *Proc. Annual Meeting of the Association for Computational Linguistics*. ACL, 690–696.

[24] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[25] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proc. 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 173–184.

[26] Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.

[27] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, and others. 2007. Moses: Open source toolkit for statistical machine translation. In *Proc. 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*. Association for Computational Linguistics, 177–180.

[28] Philipp Koehn, Franz Josef Och, and Daniel Marcu. 2003. Statistical phrase-based translation. In *Proc. 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*. Association for Computational Linguistics, 48–54.

[29] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *Proc. International Conference on Program Comprehension (ICPC)*. IEEE, 3–12.

[30] Christopher D Manning and Hinrich Schütze. 1999. *Foundations of statistical natural language processing*. MIT Press.

[31] Daniel Marcu and William Wong. 2002. A phrase-based, joint probability model for statistical machine translation. In *Proc. ACL-02 Conference on Empirical Methods in Natural Language Processing - Volume 10*. Association for Computational Linguistics, 133–139.

[32] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring multiple execution paths for malware analysis. In *Proc. IEEE Symposium on Security and Privacy (SP)*. IEEE, 231–245.

[33] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *Proc. International Conference on Automated Software Engineering (ASE)*. ACM, 457–468.

[34] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 651–654.

[35] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 544–547.

[36] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-Conquer Approach for Multi-phase Statistical Migration for Source Code (T). In *Proc. International Conference on Automated Software Engineering (ASE)*. IEEE, 585–596.

[37] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. 2015. Learning api usages from bytecode: A statistical approach. *arXiv preprint arXiv:1507.07306* (2015).

[38] Franz Josef Och. 2005. Statistical machine translation: Foundations and recent advances. Tutorial at MT Summit. (2005). http://www.mt-archive.info/MTS-2005-Och.pdf.

[39] Franz Josef Och and Hermann Ney. 2003. A systematic comparison of various statistical alignment models. *Computational Linguistics* 29, 1 (2003), 19–51.

[40] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T). In *Proc. International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

[41] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the naturalness of buggy code. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 428–439.

[42] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proc. SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 111–124.

[43] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*. IBM Corp., 174–188.

[44] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. *Software tamper resistance: Obstructing static analysis of programs*. Technical Report. Technical Report CS-2000-12, University of Virginia, 12 2000.