Noname manuscript No.
(will be inserted by the editor)

An Empirical Study of Identifier Splitting Techniques

Emily Hill[‡], David Binkley[†], Dawn Lawrie[†], Lori Pollock[*], K. Vijay-Shanker[*]

Received: date / Accepted: date

Abstract Researchers have shown that program analyses that drive software development and maintenance tools supporting search, traceability and other tasks can benefit from leveraging the natural language information found in identifiers and comments. Accurate natural language information depends on correctly splitting the identifiers into their component words and abbreviations. While conventions such as camel-casing can ease this task, conventions are not well-defined in certain situations and may be modified to improve readability, thus making automatic splitting more challenging. This paper describes an empirical study of state-of-the-art identifier splitting techniques and the construction of a publicly available oracle to evaluate identifier splitting algorithms. In addition to comparing current approaches, the results help to guide future development and evaluation of improved identifier splitting approaches.

Keywords software engineering tools · program comprehension · identifier names · source code text analysis

1 Introduction

While a program's statements and structure convey the computational intent to the compiler, the naming of program elements is used to convey domain concepts useful to programmers reading and modifying the program. Like human programmers, software engineering tools built to support programmers maintain software can leverage the natural language information inherent in program identifiers and comments. In particular, tools for program search,

[‡] Department of Computer Science, Montclair State University, Montclair, NJ 07043, E-mail: hillem@mail.montclair.edu · [†]Department of Computer Science, Loyola University Maryland, Baltimore, MD 21210 E-mail: {binkley, lawrie}@cs.loyola.edu · [*]Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716 E-mail: {pollock, vijay}@cis.udel.edu

concern location, code reuse, and documentation-to-source traceability can all benefit from extracting accurate natural language information from source code.

A key first step in analyzing the words that programmers use is to accurately split each identifier into its component words, abbreviations, and acronyms. This need arises as programmers often compose identifiers from constituent words and abbreviations (e.g., ASTVisitorTree, newValidatingXMLInputStream, jLabel6, buildXMLforComposite). Automatic splitting of identifiers with multiple words is straightforward when programmers follow conventions to separate words and abbreviations, such as using non-alphabetic characters (e.g., "_") or camel-casing, where the first letter of each word is upper case (except for the very first letter in some cases) [5,8,16,18].

Unfortunately, camel casing is not well-defined in certain situations. For example, no convention exists for including acronyms within camel case identifiers where the whole abbreviation may be capitalized, as in ConvertASCIItoUTF, or just the first letter, as in SqlList. The decision depends on the readability of the token. In particular, SqlList is arguably more readable than SQLList, and more closely follows camel case guidelines than SQLlist. Strict camel casing may be sacrificed for readability, especially for prepositions and conjunctions, as in DAYofMONTH, convertCEtoString, or PrintPRandOSError. Thus, while splitting on the transition from lower case to upper case letters is sufficient, it is incomplete, creating a need for more sophisticated techniques.

The goal of an identifier-splitting algorithm is to take an identifier as input, and output a list of substrings that partition the identifier. These substrings can be dictionary words, where its meaning is obvious, abbreviations, which represent a single dictionary word, or acronyms, which represents several dictionary words. Research has shown that in the absence of execution data, software engineering tools such as feature location techniques perform significantly better with manual splitting over automatic splitters [9]. Thus, there is a need to study, compare, and improve automatic identifier splitters. While existing splitting techniques have been individually evaluated when introduced, there is a need for a uniform test oracle to support comparison of existing and future techniques. To this end, the main contributions of this paper are

- construction of a publicly available test oracle to evaluate current and future identifier splitting algorithms, and
- support for future splitter improvements based on analysis of five state-of-the-art splitters, plus a conservative camel-case splitter as a baseline, and investigating under what conditions each splitter performs best and where further refinements can be made. This includes empirical results and analysis from a thorough study comparing the dictionary-based Greedy [11], frequency-based Samurai [10], metrics-based with dictionary GenTest [15], speech-recognition with dictionaries referred to as DTW [12], and dictionary-based with special processing for identifiers with digits (Identifier Name Tokeniser Tool, or INTT) [4].

After setting up the problem in Section 2 and describing the five splitting approaches under investigation in Section 3, Section 4 presents the construction of the test oracle. The resulting oracle can be obtained from `www.cs.loyola.edu/~binkley/ludiso`, which also provides programs written in Java, C, and C++ for reading the data and extracting subsets based on different criteria. Next, Sections 5 and 6 detail the empirical study's methodology and results, followed by related work in Section 7, and concluding with a summary and future work in Section 8.

2 The Identifier-Splitting Problem

Formally, an identifier t is a sequence of characters, $c_0, c_1, c_2, ...c_n$, where character c_i represents an individual letter, digit, or special character. Empirically, just over half of all identifiers include multiple, well-separated parts, referred to as *hard words* [2]. For example, the identifier babel_fish includes two hard words: babel and fish. Hard words are identified entirely on the basis of orthography, which provides hard evidence of the existence of a split. Two common methods for separating hard words are the inclusion of special characters (such as underscores or digits) and the use of camel casing. A camel case split exists at the transition from lower case to upper case letters (e.g., getString or setPoint). Splitting into hard words is referred to as *conservative split*.

When all the hard words are words that can be found in a dictionary or are common abbreviations, such as in getString or setGPS_Point, the identifier is split. However, for some identifiers, separation into hard words alone is insufficient. Specifically, some hard words are composed of multiple parts, referred to as *soft words*. For example, the identifier hashtable_entry includes two hard words, hashtable and entry. The hard word hashtable thus includes two soft words, hash and table, while the second hard word includes the single soft word, entry.

The goal of identifier splitting is to split an identifier into its constituent soft words such that each soft word is either a dictionary word, where its meaning is obvious, an abbreviation, representing a single dictionary word, or an acronym, representing several dictionary words. This task can be subtle when the programmer has not used camel casing or underscores, and thus has provided no hard evidence as to whether an identifier or hard word consists of multiple words. For example, generalizing from the identifier GPSstate, which suggests splitting on the transition from upper case to lower case, we would errantly produce ASTV isitor from the identifier ASTVisitor. The problem of splitting a hard word can be divided into two sub problems: the *mixed-case splitting problem* (e.g., ASTVisitor), and the *same-case splitting problem* (e.g., hashtable, notype, databasefield, USERLIB, or COUNTRYCODE). The mixed-case splitting problem is particularly complicated by the use of abbreviations and acronyms (e.g., MAXstring or GPSstate).

3 Identifier Splitting Techniques

An identifier splitting algorithm automatically separates an identifier into substrings. Better splitting algorithms identify substrings that software engineers associate with hard and soft words, that is, strings that the developer used to express some concept. Although the goal of any splitter is to be 100% accurate, in practice, some splitters perform better on particular types of identifiers. In this section, we describe the salient features of the identifier splitting techniques compared in our study, in historical order. These were the main identifier splitting techniques in existence at the time the study was performed.

Greedy. The greedy approach [11] uses a dictionary word list (from ispell), a list of known abbreviations, and a stop list of keywords which includes predefined identifiers, common library functions and variable names, and single letters. After returning each hard word found in one of the three word lists as a single soft word, the remaining hard words are considered for splitting. The algorithm recursively looks for the longest prefix and suffix that appear in one of the three lists. Whenever a substring is found in the lists, a division marker is placed at that position to signify a split and the algorithm continues until the remainder is a dictionary word or contains no dictionary words. Thus, the greedy approach is based on a predefined dictionary of words and abbreviations, and splits are determined based on whether the word is found in the dictionary, with longer words preferred.

Samurai. Inspired by mining potential expansions for abbreviations from the source code [13], this approach [10] is based on the premise that strings composing multi-word identifiers in a given program are frequently used elsewhere in the same program or in other programs. Thus, string frequency is used to determine identifier splits. Samurai mines string frequencies from source code, building a program-specific frequency table and a global frequency table from mining a large corpus of programs. The frequency tables are used in the scoring function applied during both mixed-case splitting and same-case splitting. The global and program-specific frequency tables are mined from hard words after camel-case and non-alphabetic delimiters are used for splitting. The splitting is performed as a recursive left-to-right scan, dampening the score for shorter words.

GenTest. While Samurai focuses on the mixed-case splitting problem, GenTest [15] focuses on the same-case splitting problem. Given a same-case term, GenTest first generates all possible splittings. Assuming that hard words are typically short, this potentially exponential number of splittings is much more manageable in practice. Then, each potential split is scored (i.e., tested) and the highest scoring split is selected. The scoring function works on the premise that expanded soft words should be found co-located in the documentation or in general text, thus a similarity metric is computed from co-occurrence data. A set of metrics ranging over soft word characteristics, metrics using external

information (dictionaries and information from non-source code artifacts), and metrics using internal information (derived from the program itself or corpus of programs) are computed and used by the scoring function. Some of the internal metrics are similar to Samurai's frequency tables while dictionaries like Greedy are used in the external metrics.

DTW. This approach [12] is based on the observation that programmers build new identifiers by applying a set of transformation rules to words, such as dropping all vowels or dropping one or more characters. Using a dictionary containing words and terms belonging to an upper ontology, to the application domain, or both, the goal is to identify a near optimal matching between substrings of the identifier and words in the dictionary, using an approach inspired by speech recognition. The identifier is considered a signal of unknown meaning described by a vector. Each dictionary word is then used as a second (known) signal described by a feature vector. The algorithm performs a dynamic time warping (DTW) of the two vectors to find the optimal match between the vectors. The "time warp" part of the search allows the lengths of the two vectors to differ and allows abbreviations to be accounted for in the splitting domain. The optimal match is made by computing the local distances and then choosing matches that minimize the overall distance using dynamic programming.

INTT. The INTT approach [4] seeks to perform more accurate splitting than previous techniques by using a specialized heuristic for handling identifiers with digits as opposed to separating digits from the remaining text string early in the splitting process. In particular, a large dictionary, a list of abbreviations and acronyms, and a list of acronyms containing digits are used in dealing with mixed-case splitting, where the greedy approach is adapted to better split same-case hard words while reducing the original algorithm's tendency to over-split. The key modification is replacing greedy by two algorithms, greedy and greedier, which can tokenize same-case identifiers without the requirement that the identifier begin or end with a known word. The dictionaries include 120 words common in computing and Java.

Summary. Some identifier splitters use lexical clues more than others. For example, INTT, has rules specific to dealing with digits. In contrast, DTW's use of a dictionary of solution- and domain-specific terms makes it more conceptual. The other three techniques are similar in that they are predominantly lexically-based approaches. Greedy, for example, favors long dictionary word prefixes and suffixes. This leads it to split thenewestone as *then ewe stone*. The observation that these three words do not often appear together led to GenTest's use of co-occurrence information. The words of *the newest one* co-occur much more often. Rather than co-occurrence, Samurai exploits frequency data in an attempt to identify words likely to be used in an identifier.

4 Creating the Test Oracle

Any comparison of identifier splitting algorithms first requires a test oracle (i.e., capturing ground truths in a *gold* set) of split identifiers. Creating the test oracle is broken into three phases: the collection process, the organization and analysis of the raw data, and finalizing the test oracle. The next three subsections address these three phases.

4.1 Collecting Identifier Splitting Judgements

The collection process has two subparts: gathering a set of identifiers and then gathering *splitting judgements* for those identifiers. The identifiers were extracted from a source code corpus of 2,117 open-source programs ranging in size from 1,423 to 3,087,545 LOC and covering a range of application domains (*e.g.*, accounting, operating systems, program environments, movie editing, games, etc.) and styles (command lines, GUI, real-time, embedded, etc.). The majority of the programs are Java codes randomly selected from a set of 9,000 open source Java programs downloaded from Source Forge. The remaining Java, C, and C++ programs are also open source from a variety of sources. These are described in prior studies [17, 10, 13].

In all, 434,392 unique C identifiers, 258,946 unique C++ identifiers, and 7,091,945 unique Java identifiers were obtained. For each language, 4,000 identifiers were randomly selected and then duplicates across languages were removed. (This removal did not have a large impact on the sets. For example, only 5% of the C identifiers were found in the C++ collection.) Finally, the remaining identifiers were randomly ordered.

The second part of the collection gathered splitting judgements from human annotators familiar with programming. Each judgement captures a set of *annotator-inserted splits* where an annotator (who has experience[1] with programming) inserted a split (denoted by a hyphen) into the hard words of an identifier, creating two (or more) soft words. A judgement includes the original identifier, the identifier as split by an annotator, the annotator's self-rated confidence, and a system-assigned random session ID. Note that from the original and annotator-split identifiers the annotator-inserted splits can be easily extracted. For example, the annotator who produced max-run-length from the original identifier maxrunlength inserted two splits, creating a total of three soft words.

Splitting judgements were gathered using a web-based Java applet that first provided brief instructions and then gathered the annotator's experience level. Preferring to probe annotators' untainted intuition and avoid bias, rather minimal instructions were given, which appear in Appendix A. Experience was solicited with the prompt:

[1] Annotator experience ranged from second year students to practicing professionals with almost fifty years of experience. The average experience was 13.1 years while the median was 7.0 years and the standard deviation 12.8 years.

> Before you begin, please tell us the number of years you have been programming.

This was the only personal information collected about each annotator. The applet next presented the annotator with a sequence of identifiers, with a text field pre-populated with the identifier split into hard words (using conservative splitting), to minimize annotator workload. The annotators were instructed to split the hard words further into their constituent soft words as they deemed appropriate. In addition, they were asked to remove any automatically inserted conservative splits that they deemed inappropriate. Recall that conservative split separates an identifier into hard words by inserting splits between lower to upper case letters, around sequences of one or more digits, and at underscores. After inserting additional splits between soft words (or removing hard word splits), the annotators also rated their confidence on a scale from 0, for no confidence, to 2, for high confidence. While an annotator could choose to stop at any time, they were shown at most 100 identifiers before the applet terminated.

One goal in creating the oracle was to collect sufficient identifier splittings to support statistically significant conclusions. To collect a sufficient volume of split identifiers requires reducing annotator workload to the extent possible. Thus, annotators saw an isolated identifier, without its source code context. In addition, the identifier was shown pre-split into hard words. This hopefully reduces the annotator's workload so that they can focus on identifying the more challenging splits and helps avoid mistakes due to boredom. However, pre-splitting hard words can potentially bias annotators in favor of default hard word splits (this is considered in Sections 4.2.4 and 5.5). Furthermore, showing the identifier in isolation potentially reduces annotation quality as the original source is not made available. This trade-off between data set size, annotator workload, and overall annotator quality represents a classic tradeoff when building any similar oracle.

Annotator judgements are necessarily judgements by readers of the identifiers, not the creators of the identifiers. To handle the potential ambiguity in the judgements and its potential impact on how different identifier splitting algorithms are judged, an identifier was *promoted* from receiving judgements to the *annotation complete* status when it received three judgements with confidence greater than zero, or five total judgements. By the end of the data collection, 8,522 judgements of 2,733 promoted identifiers had been collected during 112 different sessions. As some annotators returned for multiple sessions, there were fewer than 112 annotators. The exact number of annotators is unknown because human-subject rules do not allow the collection of identifying information. Most judgements (86.2%) were of high confidence with only 3.8% being of zero confidence. As a result, 90.2% (2,466 of the 2,733 identifiers) required only three judgements to promote. Of the remainder 7.7% (211) received four judgements and 2.1% (56) five.

No. of unique splittings	data for all annotations		data for annotations with non-zero confidence	
1	2,020	74%	2,071	76%
2	624	23%	598	22%
3	86	3%	64	2%
4	2	0%	0	0%
5	1	0%	0	0%

Table 1 Number of unique splittings per identifier overall, and without zero confidence.

4.2 Raw judgement organization and analysis

Having collected the raw data, the second phase of the oracle construction was to organize the gathered data. We organize the data based on the number of *unique splits*, which refers to the cardinality of the set of all annotator splits for a given identifier. For example, a uniquely split identifier has a singleton set, and thus only one unique splitting. Table 1 presents a breakdown by the number of unique splittings each identifier received. For example, the first row includes identifiers for which all judgements agreed on a single splitting while the second row includes those identifiers that received two distinct splittings. The central columns labeled "data for all annotations" include all confidence levels while the right columns labeled "non-zero confidence" exclude judgements of zero confidence.

From the breakdown in Table 1, about three quarters of the identifiers received unique, confident splittings. These identifiers, found in the first row of the table, proved easier for annotators to consistently split. Notice that when we remove zero-confidence judgements, the number of identifiers with a single unique splitting increases. This is because annotators with low confidence tended to introduce additional unique splittings for an identifier.

Data in the bottom two to three rows of the table, in particular under the "all data" columns, represent identifiers with multiple and low confidence splittings. For example, the two identifiers with four different unique splittings, defarcangpnt and wcspbrk, and the one with five unique splittings, calcmandfpasmstart, are shown in Table 2 along with the correct splitting and an English expansion of each identifier. The correct splitting was derived by examining the source code directly (something not available to the annotators using the applet) as well as conducting several internet searches. These identifiers are clearly very challenging to correctly split.

The examples shown in Table 2 illustrate the complexity found in the last two rows of the table. There is too much data summarized in the first few rows to explain through examples; thus several key subsets are considered and compared. The following analysis considers three subsets: *UniqueSplit* – identifiers with one unique splitting (the first row), *HC-Core* – the highest-confidence core of UniqueSplit, which contains those identifiers with one unique splitting where all judgements received the highest confidence score, and *2UniqueSplits* – identifiers with two unique splittings (the second row).

Unique Splits for defarcangpnt	Unique Splits for wcspbrk	Unique Splits for calcmandfpasmstart
def arc ang pnt ×2	wc sp brk	cal cmand fp asm start
def arcang pnt	wc spbr k	calc m and fp as m start
defarcang pnt	wcsp brk	calc m and fp asm start
defarcangpnt	wcspbrk ×2	calc mand fp as m start
		calcm and fpasm start
Correct Splitting		
def arc ang pnt	w c s p brk	calc mand f p asm start
English Expansion		
define arc	wide character string	calculate Mandelbrot floating
angular point	pointer break	point assembler start

Table 2 Examples with multiple unique splittings. The first four characters of wcspbrk illustrate how complex the splitting problem can become.

The remainder of this section considers four inspections of the data. The first considers the size of three sets, 2UniqueSplits, UniqueSplit, and HC-Core, and the second considers the language balance within these sets. Next the analysis turns to the splittings and investigates the percentage of hard words that require further splitting overall and by language. Finally, the last subsection considers those identifiers that had hard word splits removed by an annotator.

4.2.1 Subset Sizes

Of the 2,733 identifiers promoted by the applet from collection-of-annotations status to the raw oracle data, UniqueSplit includes 2,071, which includes HC-Core's 1,758 identifiers, and 2UniqueSplits includes the 598 identifiers that received two unique splittings. Unlike UniqueSplit, 2UniqueSplits provides a space for understanding the aggressiveness of a splitting algorithm. More aggressive approaches might prove better for techniques that incorporate natural language from non-source code sources such as the requirements. Here introducing more splits often better matches the words found in the documentation's natural language. For example, 2UniqueSplits includes BSD_REGEX. This identifier was split aggressively into BSD-REG-EX, which better matches natural language phrases such as (BSD) regular expression. In contrast, the less aggressive splitting BSD-REGEX by including REGEX, is likely closer to the vocabulary found in documents more closely related to the code.

4.2.2 Language Balance

One of the goals of the oracle generation was to balance identifier language origin. The initial set started with 4,000 C, 4,000 C++, and 4,000 Java identifiers. Ideally, this language balance would exist in the overall data set and the sets UniqueSplit, HC-Core, and 2UniqueSplits. This is not guaranteed because the identifiers were shown in a random order and only promoted after receiving

	All Identifiers		UniqueSplit		HC-Core		2UniqueSplits	
Lang.	total	soft split	total	soft split	total	soft split	total	soft split
C	916	281 (31%)	682	173 (25%)	549	107 (19%)	406	234 (58%)
C++	906	240 (26%)	706	147 (21%)	618	137 (22%)	364	220 (60%)
Java	911	248 (27%)	683	142 (21%)	591	154 (26%)	426	250 (59%)
All	2,733	769 (28%)	2,071	462 (22%)	1,758	398 (23%)	1,196	704 (59%)

Table 3 Number of identifiers with annotator-inserted soft splits introduced: overall, by language, and by level of uniqueness (UniqueSplit, HC-Core, 2UniqueSplits).

sufficient annotations. However, statistical balance was achieved in the overall data. Formally, a χ^2 proportions test finds no difference in the proportion of identifiers from each of the three languages. This is also true in UniqueSplit and its high-confidence core, HC-Core. However, it is not true for 2UniqueSplits where there are fewer C++ identifiers ($p < 0.0001$). Thus, there is some aspect of C++ identifiers that makes their splitting more consistent. Note that the lower consistency does not come from C++ identifiers dominating the 64 identifiers receiving three or more unique splittings, where 30 are C, 19 C++, and 15 Java. This difference means that care should be taken when drawing conclusions about the uniformity of the C++ identifiers from 2UniqueSplits.

4.2.3 Need for Splitting

Having attained reasonable balance across the three languages, the next question deals with the need to split identifiers beyond the conservative splitting into hard words. This question is first considered overall and then broken down by language to determine if there are any language-influenced trends. In both cases a separate answer is provided for all identifiers, UniqueSplit, and 2UniqueSplits.

The results are summarized in Table 3, which presents counts and percentages of the number of identifiers with annotator-created soft words. From the last row in the table the overall percentage of identifiers where at least one hard word was split into soft words is 28% for all identifiers, 22% for UniqueSplit, 23% for HC-Core, and 59% for 2UniqueSplits. Note that each identifier from 2UniqueSplits is counted twice (once for each unique split). One of the two must differ from the original, so the percentage modified will always be greater than 50%. This makes comparisons involving 2UniqueSplits meaningless.

The next comparison considers the average confidence for identifiers that went unchanged compared to those into which annotators inserted one or more splittings. As seen in Table 4, the unchanged splitting always received a higher average confidence than the changed splitting. This difference is statistically significant for all the data and those identifiers receiving one or two unique splittings ($p < 0.0001$ – unless otherwise stated p-values are from student's t-test). It is not significant for the 86 identifiers with three unique splittings or the three identifiers having four or five splittings. The lower confidence when inserting a split may indicate a hesitancy of annotators to make changes. If this is the case then the oracle underestimates the need for splitting.

Unique splittings	comparison with identifier as shown			
	all	unchanged	changed	t-test(unchanged, changed)
all	1.82	1.90	1.66	< 0.0001
1	1.91	1.94	1.78	< 0.0001
2	1.66	1.72	1.60	< 0.0001
3	1.35	1.40	1.32	$p = 0.36$
4	0.50	0.67	0.43	N/A
5	0.80	N/A	0.80	N/A

Table 4 Mean judgement confidence by number of unique splittings.

Turning to the by-language aspect of the question, the proportions of the identifiers for each language requiring additional splitting were compared. For 2UniqueSplits there is no statistical difference. UniqueSplit hard words from C identifiers receive marginally more splittings than C++ or Java ($p = 0.044$ and $p = 0.0045$). Finally, considering all identifiers, hard words from C identifiers received more splittings than C++ ($p = 0.048$), but not Java ($p = 0.104$). Given the age and history of C programming this is an expected result [17].

4.2.4 Removed Splits

The final inspection of the data considers hard word splits removed from an identifier by an annotator. In all, 73 identifiers had a split removed. As this is only 2.7% of the identifiers, it is not a common occurrence. Just over half (41) of these identifiers involved digits. The hard splitting rule for digits separates strings of digits from surrounding letters. This correctly splits an identifier such as err2string into the three hard words err 2 string, which corresponds to the natural language phrase *error to string*. Other cases should only be split on one side such as play3DMovie (break before) and mpeg4player (break after). Finally, some require no spits such as the V4L2 in V4L2_CAP_TIMEPERFRAME. As the number of special cases involved is small, in practice, it may be appropriate to handle these few cases using a small dictionary of digit-involving acronyms such as mp3, p2p, and 2D (as is done by INTT).

For identifiers not involving digits, the annotator's removal always violated the conservative splitting rules used for the default hard word splitting. For example, the underscore in CTL_HOME leads to the two hard words CTL and HOME, which were joined together by an annotator producing CTLHOME. A similar removal was made replacing init Image Buffer (from the identifier initImageBuffer) with init ImageBuffer, which violates the conservative splitting rules.

4.3 Finalizing the Oracle

Finally, from the raw data it is possible to derive several *oracles* that range in how demanding they are. For example, accepting any split that is found in

the raw data is very permissive. A more strict requirement would be to accept only the split that had the highest confidence sum or average confidence.

To capture the full range of complexity found in the raw data without requiring the challenge of dealing with multiple correct answers, the oracle used in the following evaluation includes only those splittings with the highest majority-weighted confidence. To calculate this set, the confidence scores for each split were summed. Identifiers with multiple splits that had the same sum were excluded from the oracle. In total 70 identifiers were removed because of confidence sum ties. The resulting oracle of 2,663 identifiers can be used to compare current and future splitting algorithms. In addition, we have made the raw data available so that future researchers can systematically choose any subset that suits their curiosity.

5 Empirical Study Methodology

Our goals in empirically studying identifier splitting algorithms are to

- identify the identifier splitting algorithm that provides the best overall effectiveness
- determine how each technique's effectiveness varies among identifiers from applications written in different programming languages and in different forms
- identify key similarities and differences in how different identifier splitting algorithms perform in different splitting situations
- determine the relative impact of dictionaries and frequency lists on splitter performance, irrespective of splitting algorithm

One cannot predict the kinds of identifiers that may exist in a given set of applications that the splitters are analyzing. Thus, we investigate the overall effectiveness of the approaches, and then perform deeper analysis into how they vary among different programming languages and different forms of identifiers (those following camel case, those multiword identifiers with all the same case, etc.). We also examine the identifiers that all splitters successfully split, those that only a subset of splitters correctly split, and those that no technique correctly splits.

In addition, the Greedy, GenTest, and Samurai splitting techniques use similar dictionary and frequency word lists, although they used different lists in their original publications. To study the effect of word list variations on splitter performance, independent of splitting algorithm, we use a consistent set of word lists that are not necessarily the lists used in training the approaches in their original publication, as described in Table 5.

5.1 Identifier Splitters

In addition to the splitters introduced in Section 3, the experiments include conservative splitting as a baseline, to put the results of the more advanced

Splitting Algorithm	Dictionary	Frequency List
Conservative Split		
DTW [12]		
INTT [4]		
GenTest [15]	Small[3], Medium[4], Large[5]	C, C++, Java, All
Greedy [11]	Small, Medium, Large	
Samurai [13]		C, C++, Java, All

Table 5 Identifier splitter configurations studied. For GenTest, Greedy, and Samurai, we investigated the degree to which dictionary size (small, medium, large) and language-based frequency list training (C, C++, Java) affects splitter performance. These lists were held constant across all three techniques, and are not necessarily the same lists evaluated in the original publications.

splitters into perspective. The conservative splitting algorithm is the same as the one used to split identifiers into the hard words shown to annotators by the applet.

Including conservative split, we compared 22 different identifier splitter configurations based on splitting algorithm, dictionary, and frequency list trained by programming language[2]. Not all the splitting algorithms use every configuration of dictionary or frequency list. Table 5 shows the splitter configurations used in the study. Other than conservative split, each is followed by a citation to the original publication of the algorithm, which discusses to varying degrees the pros and cons of alternate configurations. Other than the variations described in Table 5, each algorithm was run in its default settings. DTW has just one configuration. INTT was used in its default configuration although it is possible to replace its dictionaries and abbreviation lists. GenTest has 12 configurations, Greedy 3, and Samurai 4. In addition to the head-to-head comparisons of splitters, we performed both an intra-technique analysis to determine the most competitive configurations for GenTest, Greedy, and Samurai (sometimes selecting multiple configurations of a particular splitter), and then compared the most successful configurations in an inter-technique analysis.

5.2 Subject Identifiers

As subjects, we use the oracle outlined in Section 4, which consists of 2,663 identifiers with 6,912 annotator-inserted splits. The identifiers can be analyzed in two different units: per-identifier and per-split. Per-identifier analysis considers each identifier as a single unit, with zero or more splits. Per-identifier

[2] Information concerning all of these splitters as well as how each split the identifiers in the oracle can be found in the replication package at `www.cs.loyola.edu/~lawrie/id-splitting-data`.

[3] A dictionary with 50,276 entries defined by the concatenation of Kevin Atkinson's SCOWL word list sizes 10 thru 35 [1]

[4] A dictionary with 98,569 entries defined by the concatenation of Kevin Atkinson's SCOWL word list sizes 10 thru 50

[5] A dictionary with 479,625 entries distributed by Red Hat 4.1.2-14 as /usr/share/dict

Type of Split	C	C+++	Java	Total
Conservative Split (CS)	746	816	833	2396
Same-Case Split (SC)	252	165	155	573
Acronym Split (ACR)	27	82	91	201
Alternating Case Split (ALT)	3	7	6	16

Table 6 Number of identifiers containing different types of splits in the test oracle, by programming language.

analysis provides information about how well an identifier splitting technique will perform when splitting an arbitrary identifier in practice. However, it is possible that a splitting technique will make mistakes on certain types of splits more than others, which is obscured when analyzing the identifier as a whole (since a single identifier may contain different types of splits). For example, a technique may be very good at splitting acronyms in identifiers like XMLTerminal or aRGBMap, but not as good at splitting sections of identifiers with no case difference, like strcpy or containerinfo. To better understand these differences, we also perform a per-split analysis of the identifiers by analyzing the following subsets of splits, where 'a' and 'u' are used to represent a lower case letter, 'A' and 'U' to represent upper case, and '9' to represent digits:

Conservative Split (CS): splits between aA, a_a, a9, 9a
Same-Case Split (SC): splits between aa, splits AAU → A-AU, and AA$ → A-A$ (where $ represents the end of the identifier)
Alternating-Case Split (ALT): splits between Aa
Acronym Split (ACR): splits AAu → A-Au

In addition to analyzing the identifiers as a whole, we also split the identifiers into different subsets to better understand the differences between the various techniques. At the whole-identifier level, we separately analyze identifiers with and without digits, when the annotator inserted a split beyond conservative split, and when the identifier had no split in the oracle. Table 6 shows the number of types of splits for the test oracle, separated by programming language of the identifiers' source code.

5.3 Measures

For the per-identifier analysis we use four measures: accuracy, precision, recall, and F-measure; for the per-split analysis we use a modified version of accuracy.

Per-identifier accuracy is a binary measure, one if the splitter output is identical to the oracle, zero otherwise. The mean accuracy approximates the likelihood of an identifier splitter perfectly splitting an identifier. Formally, for a given identifier i, a splitter s and an oracle o, we define the per-identifier accuracy to be:

$$accuracy_{pi}(i) = \begin{cases} 1, & s(i) = o(i) \\ 0, & s(i) \neq o(i) \end{cases}$$

In addition to accuracy, we calculate variations on precision and recall adapted for the identifier splitting problem. Ideally, we would like to define precision and recall measures in terms of the number of correctly inserted splits. However, such definitions make identifiers with no splits difficult to calculate, since the correct number of splits would be 0. To avoid this scenario, we approximate the number of correctly split words by adding one to the number of inserted splits. Thus, for a given splitting of an identifier i by splitter s, we define precision to be

$$precision(i, s) = \frac{1 + tp_s(i)}{1 + n_s(i)}$$

where tp_s is the number of correctly inserted splits and $n_s(i)$ is the total number of splits inserted by splitter s for identifier i. Intuitively, precision approximates the degree of over-splitting by a technique, where low precision implies that a technique tends to over-split. Similarly, for a given splitting of an identifier i by splitter s, we define recall to be

$$recall(i, s) = \frac{1 + tp_s(i)}{1 + n_o(i)}$$

where tp_s is the number of correctly inserted splits and $n_o(i)$ is the total number of splits inserted by the oracle, o, for identifier i. The F-measure uses a harmonic mean to combine precision and recall and is high only when precision and recall are similarly high.

Per-split accuracy captures the percent of correct splits, regardless of how the splits are distributed across all identifiers (in contrast, per-identifier measures depend on the distribution of splits *within* an identifier). Per-split accuracy is the total number of correct splits inserted by the splitter across all identifiers, divided by the total number of unique splits inserted by both the tool and the oracle. Formally, for an identifier splitter s, the set of all splits inserted by the splitter, S, and the set of all splits inserted by the oracle, O, we define per-split accuracy to be:

$$accuracy_{ps}(s) = \frac{|O \cap S|}{|O \cup S|} = \frac{tp_s}{n_o + n_s - tp_s}$$

where tp_s is the number of correctly inserted splits, n_o is the total number of splits inserted by the oracle, o, and n_s is the total number of splits inserted by splitter s. We define per-split accuracy not just in terms of the number of oracle splits, but also the number of tool-inserted splits, so we can fairly compare the accuracy of greedier techniques that tend to over-split. If we only considered oracle splits, a technique that inserted all possible splits would have better accuracy than a technique that inserted fewer splits in the wrong places.

The purpose of per-split accuracy is to better understand how each splitter handles particular types of splits that may not occur in every identifier, such as same-case, etc. The above formula can be modified to apply only to a subset of splits, such as acronyms, same-case, splits occurring only in identifiers that

contain digits or a particular language, etc. Because this measure is a total calculated once over the entire data set (or per subset of type of split), rather than a mean, it is impossible to evaluate statistical significance for per-split analysis.

5.4 Analysis Methodology

We analyzed the per-identifier results using a combination of figures and statistics. First, we created box plots for each measure and configuration to get a high level view of the data. Next, we used the ANOVA F-Test [21] to determine if means of any of the measures were significantly different. Then we used Tukey's Honest Significant Difference [21] test to evaluate the degree and direction of the mean differences. Finally, we updated the boxplots by annotating them with significant difference information.

We define our analysis into two phases: intra-technique and inter-technique analysis. During intra-technique analysis, we analyzed differences within the configuration of each technique as well as selecting the most competitive configurations for the inter-technique analysis phase. We selected the most competitive techniques using means alone in the absence of significant differences. Next, in the inter-technique analysis phase, we comparatively studied how the splitters performed relative to one another.

When analyzing paired subsets of the identifiers for further comparison, such as identifiers with and without annotator-inserted splits, we used the Wilcoxon-Mann-Whitney u-test (a.k.a. the Wilcoxon rank sum test) [21] to determine if the accuracy for the subsets were significantly different for each technique. The u-test is a non-parametric alternative to the independent two-sample t-test that does not require normally distributed data.

5.5 Threats to Validity

We attempted to ensure generalizability of the results by randomly selecting identifiers from programs written in three different languages: C, C++, and Java. However, the results may not generalize to all identifiers in these languages or identifiers written in other languages. To obtain annotations of as many identifiers as possible in the oracle, the identifiers were presented to the annotators out of their source code context and thus the human annotators did not have contextual information that would be available to a programmer reading the actual source code. However, presenting the identifiers out of context enabled our human annotators to give us many more identifier splitting judgements, so our set could reflect a wider variety of identifiers. Future work could include a hybrid design that analyzes the effects of source code context on identifier split accuracy.

The human annotators were all volunteers who are assumed unlikely to have direct experience with any of the identifiers seen (based on the sheer vol-

ume of the source code considered). However, knowing where to insert identifier splits can be an ambiguous and subjective task. This is especially true when dealing with compound words, abbreviations of common concepts, and digits. The annotators are judging identifiers that some other developer created to express a concept. They do not know what the developer was trying to express or have that same context. We have tried to limit this threat by obtaining at least three judgements per identifier with confidence greater than zero. Sometimes, the annotators kept together substrings, such as SWTSwing, as one concept. For some splitters, especially lexical-based ones, this causes the splitter to over-split. For instance, there were 38 such cases where only CS matched the annotators. In these situations, the annotators agreed that the identifier represents a single concept.

Lastly, care must be taken in interpreting the study results, as the oracle captures a more conceptual human interpretation of identifier name splitting that is inherently challenging to more lexically-based splitting algorithms. For example, this might manifest itself in a splitter over-splitting relative to annotators. This difference may favor some splitters over others and thus underscores the complexities of providing a level playing field when performing such comparisons.

6 Empirical Results

This section begins with the intra-technique results that indicate which configurations of the techniques give the best overall performance. We then describe the inter-technique results from comparing the performance among the different techniques overall and in detailed subsets. Finally, we summarize the main conclusions from the study.

6.1 Intra-Technique Analysis

Table 7 shows the mean accuracy, precision, recall, and F-measure, as well as standard deviations for each configuration evaluated in the study. The techniques are ordered by the mean accuracy of the best configuration, and the configurations selected for inter-technique analysis are starred and highlighted.

From this table, we can see that the configuration can have a significant impact on a technique's performance. For instance, Greedy can take different dictionaries. Our results show that a large or small dictionary provides significantly better accuracy, precision and F-measure with no difference in recall as compared to using a medium dictionary. Although this may seem like an unintuitive result, the medium dictionary adds a number of short abbreviations that Greedy finds in identifiers. Although the large dictionary also contains these short abbreviations, it adds technical jargon, which offsets the misdirections introduced by the medium dictionary. Thus, for inter-technique analysis, we use Greedy with a large dictionary (Greedy_lg) and Greedy with a small

Technique & Configuration		Acc	sd	P	sd	R	sd	F	sd
Samurai	Java	0.82	0.38	0.97	0.10	0.96	0.11	0.96	0.10
	All*	0.82	0.38	0.97	0.10	0.96	0.12	0.96	0.10
	C	0.81	0.39	0.98	0.08	0.95	0.13	0.96	0.10
	C++*	0.81	0.39	0.98	0.08	0.95	0.13	0.96	0.10
GenTest	Large, C++	0.80	0.40	0.97	0.09	0.96	0.12	0.96	0.10
	Large, All*	0.80	0.40	0.97	0.09	0.96	0.12	0.96	0.10
	Medium, All	0.80	0.40	0.97	0.09	0.96	0.12	0.96	0.10
	Medium, C++	0.80	0.40	0.97	0.09	0.96	0.12	0.96	0.10
	Large, C	0.79	0.41	0.97	0.09	0.96	0.12	0.96	0.10
	Medium, C	0.79	0.41	0.97	0.09	0.96	0.12	0.96	0.10
	Small, All	0.77	0.42	0.96	0.10	0.96	0.13	0.95	0.10
	Small, C++	0.77	0.42	0.96	0.10	0.96	0.13	0.95	0.10
	Small, C	0.76	0.43	0.96	0.10	0.96	0.13	0.95	0.10
	Medium, Java*	0.74	0.44	0.95	0.12	0.97	0.11	0.95	0.10
	Large, Java	0.73	0.45	0.94	0.12	0.97	0.12	0.95	0.11
	Small, Java	0.69	0.46	0.94	0.12	0.96	0.12	0.94	0.11
INTT*		0.75	0.43	0.98	0.09	0.93	0.14	0.95	0.11
Conservative Split (CS)*		0.71	0.45	1.00	0.01	0.90	0.18	0.94	0.12
DTW*		0.68	0.47	0.93	0.15	0.94	0.14	0.93	0.14
Greedy	Large*	0.60	0.49	0.89	0.16	0.97	0.09	0.92	0.12
	Small*	0.56	0.50	0.88	0.16	0.97	0.09	0.92	0.12
	Medium	0.51	0.50	0.86	0.17	0.97	0.09	0.90	0.12

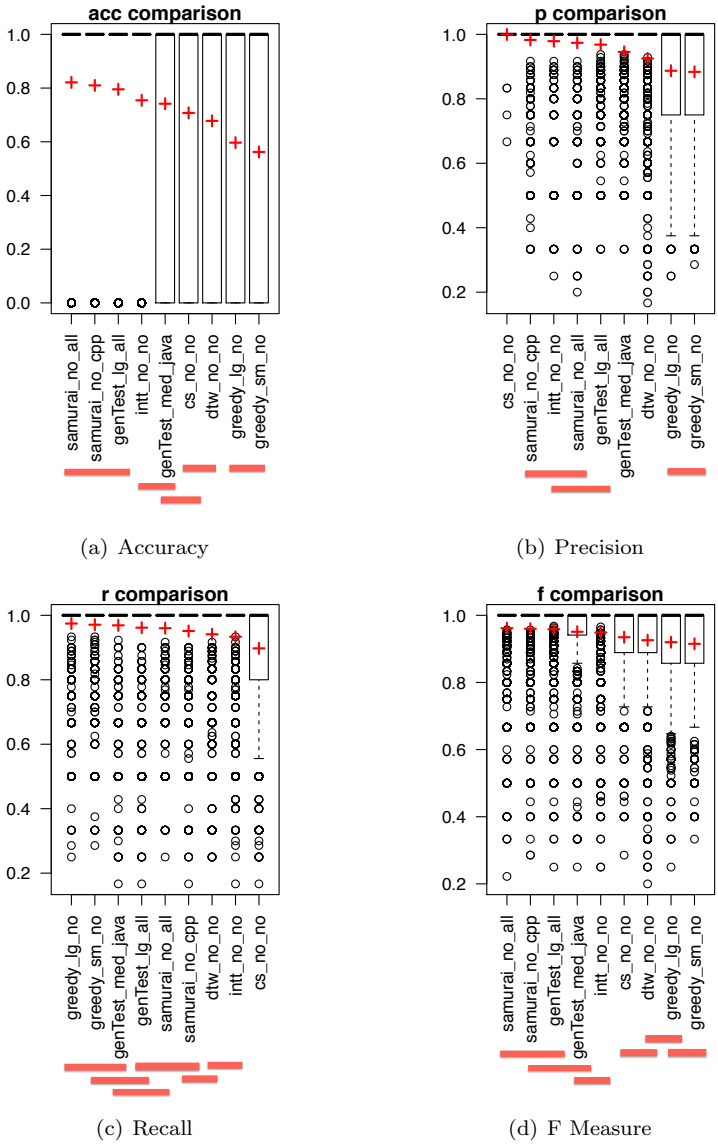
Table 7 Mean per-identifier accuracy (Acc), precision (P), recall (R), F-measure (F), and standard deviations (sd) for each configuration, ordered by best mean accuracy within each technique. ** Configurations selected for inter-technique analysis.*

dictionary (Greedy_sm). Similar to Greedy, GenTest can be configured with different dictionaries and frequency lists. Since the combination that provides the highest precision is a large dictionary with the union of all frequency lists, for inter-technique analysis we use GenTest_lg_all as one configuration and GenTest_med_Java as the configuration providing the highest recall.

For Samurai, the main configurable factor is the frequency list. In comparing C, C++, Java, and the union of these frequency lists, there are tradeoffs between precision and recall. For inter-technique analysis, we use Samurai with the C++ frequency list (Samurai_cpp), which provides significantly higher precision. Although the Java frequency list provides significantly higher recall, for inter-technique analysis we also chose Samurai with the union of all frequency lists (Samurai_all) because it has very similar recall and this choice makes it more consistent with the configurations of the other techniques. The remaining techniques (DTW, INTT, CS) do not require any external dictionary or frequency list, and thus only one configuration was considered. It is possible with INTT for a user to replace the tool's dictionaries and abbreviation lists.

6.2 Overall Inter-Technique Analysis

Figure 1 shows box plots of the accuracy, precision, recall, and F-measure results for the nine techniques in the inter-technique comparison. Each data point represents the accuracy, precision, recall, and F-measure calculation,



(a) Accuracy

(b) Precision

(c) Recall

(d) F Measure

Fig. 1 Overall Inter-Technique Analysis Results. Techniques labelled with the same line are not significantly different.

respectively, for a single identifier, with a total of 2,663 data points used to calculate the mean, median, and quartiles for each splitting technique. The box represents the inner 50% of the data, the heavy middle line represents the median, the plus represents the mean, and '○' indicates outliers. As shown in Figure 1(a), aside from a few outliers, both versions of Samurai, GenTest_lg, and INTT have perfect accuracy for more than 75% of the identifiers in the

study. Because accuracy is a binary measure, the mean accuracy captures what percentage of the time each identifier splitter correctly splits an entire identifier.

In each figure, the lines under the technique labels are used to indicate statistical significance: techniques connected by a line are not significantly different at $\alpha = 0.05$. For example, in Figure 1(a), both versions of Samurai and GenTest_lg perform similarly, and outperform the remaining six techniques with statistical significance ($\alpha = 0.05$). Although GenTest_med is not significantly different from INTT or ConservativeSplit (CS), it significantly outperforms DTW and the greedy techniques. Likewise, INTT significantly outperforms CS, DTW, and the greedy techniques for accuracy.

The approach that yields the highest precision is CS, followed by Samurai and INTT. On the other hand, for recall, Greedy and GenTest appear to be the best performing systems. If the goal is to consider both recall and precision in terms of accuracy and F-measure, then GenTest_lg and Samurai_all perform the best. While INTT is never the best in terms of raw mean, it is often not significantly different from Samurai or GenTest. Greedy and CS are not competitive unless the focus is only on recall or precision, respectively.

6.3 Detailed Inter-Technique Analysis

This section explores the results when all or none of the techniques correctly split the identifiers and when only one technique correctly splits identifiers. There are 656 (25%) identifiers that were correctly split by all the techniques. These identifiers predominantly involve camel case, underscore, and cases where the individual split strings are common, well-known words. For example, identifiers such as ASE_KeyMESH_ANIMATION, GetFrontWindowOfClass, CERT_DecodeCertificatePoliciesExtension, and AddBorderPaddingToMaxElementSize are all easily split by camel casing and consist of well-known or frequently-occurring words in source code. In addition, there are 43 cases where no technique correctly splits the identifier, all due to same-case splits. Although each technique has mechanisms to handle same-case splits, these examples involve uncommon words that are unlikely to occur in dictionaries or frequency lists, making them especially challenging. For example,

PL_strdup (PL-str-dup),
NS_UNICODETOMACROMAN_CONTRACTID
(NS-UNICODE-TO-MACRO-MAN-CONTRACT-ID), and
COMPUTIME_FORMAT (COMPU-TIME-FORMAT).

However, some identifiers such as

audio_strerror (audio-str-error) and
IID_IHTMLElementCollection (IID-I-HTML-Element-Collection),

which contain commonly occurring words, are also not correctly split by any technique.

We examined cases where only a single technique correctly splits the identifier. For all techniques except DTW, there were only a handful of such cases

Technique	All	C	C++	Java
Samurai_all	0.82	0.79	0.85	0.83
Samurai_cpp	0.81	0.77	0.85	0.81
GenTest_lg_all	0.80	0.78	0.82	0.78
INTT	0.75	0.70	0.78	0.78
GenTest_med_java	0.74	0.75	0.77	0.71
CS	0.71	0.68	0.72	0.72
DTW	0.69	0.75	0.66	0.65
Greedy_lg	0.60	0.59	0.66	0.54
Greedy_sm	0.56	0.58	0.59	0.51
Count	2663	885	887	891
% of data	100%	33%	33%	33%

Table 8 Mean per-identifier accuracy for each programming language subset.

and no obvious pattern of characteristics of those examples emerged. However, DTW is the only correct technique for 24 identifiers, due to a carefully handcrafted dictionary which enables splitting of same-case identifiers with system-customized abbreviations such as req, imap, num, cpy, ffe, id, and cpu. In addition, there are 38 cases where only CS correctly splits the identifier. These involve identifiers that contain soft words beginning with multiple consecutive upper case letters, such as XPath, SWTSwing, and XShm. Multiple annotators have chosen to view each of these as single concepts (i.e., compound words) and kept them together, while the other techniques chose to split them into X-Path, SWT-Swing, and X-Shm.

6.4 Subsets by Programming Language

Tables 8 and 9 show the per-identifier accuracy, precision, recall, and F-measure results for each programming language subset of identifiers. Samurai, INTT, and CS are less accurate for C than for identifiers from other languages. We hypothesize this is due to the fact that they were predominantly designed for Java. In addition, both versions of Samurai perform better on C++ rather than Java identifiers, which we suspect is due to better performance on identifiers containing acronym and same-case splits for these languages. GenTest performs about the same on all languages, performing slightly better on C++. In our per-split accuracy analysis, we find that INTT performs the best on C++ identifiers overall, although performing much worse on same-case splits for C++ as compared to the other languages and techniques.

Although not the best performing technique on C identifiers, DTW is much more accurate for C identifiers than for other languages. This is due to two reasons. First, DTW uses a customized dictionary for C, which includes system-like abbreviations that are common in C programs, allowing DTW to correctly handle same-case splits. Because same-case splits are more prevalent in the C identifiers than the other two languages in the oracle, we believe this accounts for DTW's increased performance on C. The per-split data shows that DTW has 50% accuracy on same-case identifiers for C, and just 40% and 30% accu-

Technique	Measure	All	C	C++	Java
Samurai_all	P	0.97	0.96	0.98	0.98
	R	0.96	0.94	0.97	0.97
	F	0.96	0.95	0.97	0.97
Samurai_cpp	P	0.98	0.98	0.98	0.98
	R	0.95	0.93	0.96	0.96
	F	0.96	0.94	0.97	0.97
GenTest_lg_all	P	0.97	0.97	0.97	0.96
	R	0.96	0.94	0.96	0.98
	F	0.96	0.95	0.96	0.96
INTT	P	0.98	0.97	0.99	0.98
	R	0.93	0.91	0.94	0.95
	F	0.95	0.93	0.95	0.96
GenTest_med_java	P	0.95	0.95	0.95	0.94
	R	0.97	0.96	0.97	0.98
	F	0.95	0.95	0.95	0.95
CS	P	1.00	1.00	1.00	1.00
	R	0.90	0.88	0.90	0.91
	F	0.94	0.92	0.94	0.95
DTW	P	0.93	0.94	0.92	0.92
	R	0.94	0.96	0.91	0.95
	F	0.93	0.95	0.91	0.93
Greedy_lg	P	0.89	0.89	0.91	0.86
	R	0.97	0.96	0.98	0.98
	F	0.92	0.91	0.94	0.91
Greedy_sm	P	0.88	0.89	0.90	0.86
	R	0.97	0.96	0.98	0.98
	F	0.92	0.91	0.93	0.91

Table 9 Mean precision (P), recall (R), and F-measure (F) for each programming language subset, ordered by mean overall accuracy.

racy for C++ and Java, respectively. Secondly, DTW does not take advantage of camel case rules, causing it to perform poorly on C++ and Java identifiers, where conservative splitting is prevalent.

Greedy has poorer per-identifier precision and accuracy for Java than C and C++. We believe this is due to Greedy's poor performance on same-case Java identifiers. In terms of per-split accuracy analysis, Greedy_lg has just 30% accuracy for Java same-case splits, whereas it achieves 33% and 40% accuracy on same-case splits for C and C++, respectively.

6.5 Subsets by Form of Identifier

In this section we analyze the results in different contexts, including identifiers with and without digits (*Digit* and *NoDigit*, respectively). We analyzed when the annotator made no changes beyond the conservatively split hard words (*NoChg*), and analyzed when the annotator either removed hard word splits or inserted additional soft word splits (*Change*). Finally, we analyzed when the identifier had no split in the oracle (*NoSplit* vs *Split*). Table 10 shows the relative accuracy of each technique for these subsets.

Technique / Subset	All	*NoDigit*	*Digit*	*Split*	*NoSplit*	*NoChg*	*Change*
Samurai_all	0.82	0.82	0.83	0.83	0.61	0.94	0.54
Samurai_cpp	0.81	0.81	0.83	0.81	0.81	0.95	0.48
GenTest_lg_all	0.80	0.80	0.80	0.79	0.88	0.87	0.61
INTT	0.75	0.79	0.29	0.76	0.67	0.91	0.39
GenTest_med_java	0.74	0.74	0.75	0.74	0.77	0.79	0.63
CS	0.71	0.70	0.78	0.70	1.00	1.00	0.00
DTW	0.68	0.68	0.59	0.68	0.54	0.69	0.64
Greedy_lg	0.60	0.62	0.31	0.61	0.26	0.61	0.56
Greedy_sm	0.56	0.59	0.16	0.57	0.38	0.55	0.60
Count	2663	2483	180	2594	69	1887	776
% of data	100%	93%	7%	97%	3%	71%	29%

Table 10 Mean per-identifier accuracy for subsets of identifiers.

As expected, most of the techniques perform worse when the human oracle inserts a split beyond CS (*Change*), as these are the more difficult cases to split correctly. In fact, some of the highest performing techniques (Samurai and INTT) have the worst performance on the *Change* set, except for GenTest. This is likely because Samurai and INTT were originally developed on Java, whereas GenTest was originally developed on C, C++, and Java. All of the techniques compared in this study perform worse on the *Change* set with statistical significance, except for Greedy_sm, which performs significantly better ($\alpha = 0.05$, u-test). It is interesting to note that DTW has the best overall accuracy for these difficult cases. Perhaps a hybrid approach using camel case followed by DTW would yield better results. The *Change* set comprises 30% of the identifiers in our test set, making it an important subset for further study.

The *NoSplit* set contains the cases where the annotators inserted no splits (i.e. the identifier consisted of a single atomic concept), and serves to evaluate how greedy each technique is and the degree of false positives. As expected, the greedy techniques perform poorly on this set, with Samurai_all, DTW, and INTT also significantly less accurate on this set ($\alpha = 0.05$, u-test). The exception is GenTest and Samurai_cpp, the two techniques trained on non-Java identifiers. CS gets 100% accuracy on this set since it inserts no splits and the oracle contains no splits. Note that there are only 69 identifiers (3%) with no splits in our test set.

Finally, we also analyze the performance of the identifiers with and without digits. Digits are particularly challenging for splitting since determining whether a digit should be tokenized with a word as a concept can vary. For example, it may make sense to leave mp3 together, but not convert 2 string. Thus, we have separated out this analysis to better help researchers select the appropriate splitting technique based on the needs of their data. Note that there are only 180 identifiers (7%) containing digits in our test set. Although Samurai, GenTest, and CS separate out all digits by default, they have the highest accuracy on this subset (between 75-85%). In contrast, INTT, DTW, and Greedy perform significantly worse on the subset of identifiers containing digits ($\alpha = 0.05$, u-test).

Technique / Split Type	All	CS	ALT	ACR	SC
Samurai_all	0.91	1.00	0.25	0.82	0.38
Samurai_cpp	0.91	1.00	0.06	0.81	0.32
GenTest_lg_all	0.90	1.00	0.21	0.79	0.37
INTT	0.87	0.98	0.21	0.80	0.19
GenTest_med_java	0.88	1.00	0.31	0.80	0.34
CS	0.86	1.00	0.00	0.00	0.00
DTW	0.81	0.93	0.07	0.73	0.39
Greedy_lg	0.77	0.98	0.12	0.83	0.24
Greedy_sm	0.78	0.96	0.18	0.82	0.29
Count	6912	5969	16	206	721
% of data	100.0%	86.4%	0.2%	3.0%	10.4%

Table 11 Per-split accuracy, ordered by mean overall accuracy, for all splits (All), conservative splits (CS), alternating-case splits (ALT), acronym splits (ACR), and same-case splits (SC).

6.6 Per-Split Analysis

While Table 6 shows the number of identifiers containing each type of split and Tables 9 and 10 report mean per-identifier accuracy across all identifiers, Table 11 shows the overall per-split accuracy of each technique for each type of split, regardless of how the splits are distributed among the identifiers.

All the techniques do extremely well on CS. Although DTW does not use letter casing in its split analysis, it still has 93% accuracy for the CS splits across all identifiers. The next most prevalent split case is SC, representing 10% of the data. All the techniques uniformly perform poorly on same-case situations, all below 40% accuracy. However, DTW, Samurai_all, and GenTest_lg_all perform the best overall for SC. Although mixed-case (ALT + ACR) comprises just 3% of the splits in our test set, Samurai_all, GenTest, and INTT all perform competitively.

6.7 Summary

From this evaluation, we observe that a splitter's configuration can have a significant impact on performance. We observed that 25% of the identifiers are correctly split by all techniques. These identifiers predominantly involve camel case, underscore, and cases where the individual split strings are common, well-known words. Across all the techniques, we found that in terms of per-identifier accuracy and F-measure, GenTest and Samurai appear to perform the best overall. While INTT is never the best in terms of raw mean, it is often not significantly different from Samurai or GenTest. Greedy and CS are not competitive unless the focus is only on recall or precision, respectively. DTW would likely perform better across all identifiers if it pre-split hard words following camel case rules.

In terms of programming language, Samurai, GenTest, and INTT perform the best for C++ and Java identifiers. These three techniques also perform

the best when splitting C identifiers, with DTW also doing well, and CS and Greedy performing particularly poorly on this subset.

For identifiers that are difficult to split (have splits beyond conservative splitting), DTW, GenTest, and Greedy with a small dictionary perform the best overall. For same-case splits across all identifiers, DTW, Samurai, and GenTest perform the best. Given that GenTest is also one of the most accurate techniques, it would make a good overall splitter for hard-to-split code bases. For codes that more closely follow naming conventions, Samurai's slight edge in overall accuracy might be better suited.

There were 43 identifiers that were not split correctly by any technique (all due to same-case splitting mistakes). This observation motivates considering a hybrid approach that can call on the other approaches. In the ideal scenario, the hybrid algorithm would produce the correct splitting if any of the splitters it can call upon produced the correct splitting. Such a hybrid splitter would only be wrong in 43 of more than 2600 cases (less than 2%). Thus, for example, it might be interesting to train a genetic algorithm to select between the different splits in an attempt to achieve this potential 98% correctness.

7 Related Work

The work most closely related to identifier splitting deals with identifier expansion. This section describes four existing expansion algorithms. It does not discuss splitting techniques, which were considered in Section 3. Nor does it consider the multitude of techniques that have used some form of splitting.

The process of vocabulary normalization (bringing the vocabulary of source code into line with that of other software artifacts such as requirements) both splits an identifier into its constituent parts and then expands each part into one or more full dictionary words. The result better matches the vocabulary found in other software artifacts. This section considers the expansion part of vocabulary normalization.

The need to normalize vocabulary in support of IR-based tools was first noted by Feild, et al. [11,16]. This early work incorporated a limited form of wildcard expansion: words from source code and then a dictionary are searched using a pattern based on the soft word. For example, the pattern `a*v*g*` is used for the abbreviation `avg` (here a '*' matches any sequence of characters). When there is a single match, it is returned as the expansion for a soft word. Despite the algorithm simply failing when zero or more than one match occurs, it correctly expanded 40% of a sample of 64 identifiers. Two similar approaches have been considered. One expands soft-words using a manually created dictionary of common abbreviations [22]. The other restructures identifiers to conform to a standard in both the lexicon of the composed words and in syntactic composition [6]. In the latter, identifiers are split and then each soft word is looked for in a standard dictionary and a synonym dictionary. Similar to the approach taken by Feild, et al., no automatic abbreviation expansion is attempted.

Since this initial work, four improvements have been investigated. The first improvement, AMAP, works with Java code, where it applies a specific series of regular expression searches in an ever-expanding syntactic context [13]. This starts with the JavaDoc comments where, for example, the pattern

@param abbreviation abbreviation[a-z0-9A-Z]*

is used to expand an abbreviation formed by truncating the expanded word. For example, this search succeeds in expanding the abbreviation len when the JavaDoc includes the comment @param len - length of the wall. This approach works well, correctly expanding 60% of 250 non-dictionary soft words extracted from Java identifiers. Increasing the correctness would be possible if the vocabulary needed for an expansion could be selectively acquired. For example, such vocabulary is often found in a class or file defining a type rather than at the type's point-of-use. The challenge with incorporating wider sources of information is filtering out irrelevant vocabulary.

The second improvement applies dynamic time warping (i.e., DTW) to split and expand identifiers [19]. Dynamic time warping aligns two signals (classically two speech utterances) by "warping" the time when certain key attributes of the speech occur. Applied to an abbreviation and a potential expansion, the warping is used to align the letters of the abbreviation with those of a potential expansion. The technique requires a reasonably precise dictionary because an abbreviation such as len is easier to warp into lent than length.

The Normalize algorithm breaks an identifier into parts and then expands any abbreviations and acronyms to full words [14,15]. Non-dictionary soft words are assumed to be abbreviations or acronyms. The heart of expansion is a similarity metric computed based on co-occurrence data derived from a general text data set of over a trillion words extracted by Google and distributed by the Linguistic Data Consortium [3]. This data is used because it has proven useful in resolving translation ambiguity [20]. In other words, Normalize relies on the fact that expanded soft words should be found co-located in general text. To further guide the selection, co-occurrence with *context information* is also considered. For example, the soft word dir may expand to direction or directory. If the local context includes the words forward and backward, a higher probability of these words co-occurring with direction, as compared to their co-occurring with directory, is expected to lead to direction being selected as the correct expansion. Thus, this information helps to ground the expansions to a context. It also enables expansion of singleton soft words (i.e., where the entire identifier is a single soft word such as num). In the algorithm, the set of context words is simply the dictionary words found in close proximity to the identifier. The current implementation takes "close proximity" to be the identifier's surrounding function.

Recently, the LINSEN approach was developed [7]. Like Normalize and DTW, LINSEN splits identifiers and expands abbreviations. LINSEN uses an efficient approximate string matching algorithm, BYP, in conjunction with nested context-based dictionaries that represent both high-level and domain-

dependent words. Initial results show that for identifier splitting, improvements range from 3-37% over Normalize and 1-4% over DTW.

8 Conclusion and Future Directions

In this paper, we present an empirical study comparing five state-of-the-art identifier splitting techniques (Samurai, GenTest, INTT, DTW, and Greedy) and a commonly used baseline (conservative split, or CS). Our data is based on over 2,600 identifiers and 6,900 annotator-inserted splits collected from 112 different splitting sessions. Our results show that Samurai and GenTest perform the best overall, with INTT performing comparably. GenTest and DTW perform best on the most challenging cases (i.e., identifiers where the human oracle inserted a split beyond conservative splitting). DTW would likely perform better across all identifiers if it pre-split hard words using camel case or Samurai.

What does future work hold for identifier splitting research? Digits and splits beyond conservative splitting, such as same-case splits, remain open problems that future identifier splitters need to address. In addition, the impact of identifier splitting techniques on specific software engineering problems needs to be further evaluated [9]. As overall identifier splitting accuracy improves, further gains may come by making the splitting approaches customizable for specific software engineering tasks, programming languages, natural languages, and types of software artifacts. In addition, answers to some more foundational questions might be of value to the software engineering community. For example, a per-program study of the percentage of camel case and underscores using identifiers that require no soft word splitting.

9 Acknowledgments

Special thanks to all the participants as this work would not be possible without your time and also to Chris Morrell for help with the statistics. Support for this work was provided by NSF grant CCF 0916081.

References

1. Atkinson, K.: Spell checking oriented word lists (scowl). URL `http://wordlist.sourceforge.net/`
2. Binkley, D., Davis, M., Lawrie, D., Maletic, J., Morrell, C., Sharif, B.: The impact of identifier style on effort and comprehension. Empirical Software Engineering **18**, 219–276 (2013). DOI 10.1007/s10664-012-9201-4
3. Brants, T., Franz, A.: Web 1t 5-gram version 1 (2006). Linguistic Data Consortium, Philadelphia
4. Butler, S., Wermelinger, M., Yu, Y., Sharp, H.: Improving the tokenisation of identifier names. In: Proceedings of the 25th European conference on Object-oriented programming, ECOOP'11, pp. 130–154. Springer-Verlag, Berlin, Heidelberg (2011). URL `http://dl.acm.org/citation.cfm?id=2032497.2032507`

5. Caprile, B., Tonella, P.: Nomen est omen: Analyzing the language of function identifiers. In: WCRE '99: Proceedings of the 6th Working Conference on Reverse Engineering, pp. 112–122 (1999)
6. Caprile, B., Tonella, P.: Restructuring program identifier names. In: ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00), p. 97. IEEE Computer Society, Washington, DC, USA (2000)
7. Corazza, A., Martino, S.D., Maggio, V.: Linsen: An approach to split identifiers and expand abbreviations with linear complexity. In: Proceedings of the 2012 IEEE International Conference on Software Maintenance, ICSM '12. IEEE Computer Society, Washington, DC, USA (2012)
8. Deissenboeck, F., Pizka, M.: Concise and consistent naming. Software Quality Control **14**(3), 261–282 (2006). DOI http://dx.doi.org/10.1007/s11219-006-9219-1
9. Dit, B., Guerrouj, L., Poshyvanyk, D., Antoniol, G.: Can better identifier splitting techniques help feature location? In: 2011 IEEE 19th International Conference on Program Comprehension (ICPC), pp. 11 –20 (2011). DOI 10.1109/ICPC.2011.47
10. Enslen, E., Hill, E., Pollock, L., Vijay-Shanker, K.: Mining source code to automatically split identifiers for software analysis. Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 **0**, 71–80 (2009). DOI http://doi.ieeecomputersociety.org/10.1109/MSR.2009.5069482
11. Feild, H., Binkley, D., Lawrie, D.: An empirical comparison of techniques for extracting concept abbreviations from identifiers. In: Proceedings of IASTED International Conference on Software Engineering and Applications (SEA'06) (2006)
12. Guerrouj, L., Di Penta, M., Antoniol, G., Guéhéneuc, Y.G.: Tidier: an identifier splitting approach using speech recognition techniques. Journal of Software Maintenance and Evolution: Research and Practice pp. n/a–n/a (2011). DOI 10.1002/smr.539. URL `http://dx.doi.org/10.1002/smr.539`
13. Hill, E., Fry, Z.P., Boyd, H., Sridhara, G., Novikova, Y., Pollock, L., Vijay-Shanker, K.: AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In: MSR '08: Proceedings of the 5th International Working Conference on Mining Software Repositories. IEEE Computer Society, Washington, DC, USA (2008)
14. Lawrie, D., Binkley, D.: Expanding identifiers to normalizing source code vocabulary. In: ICSM '11: Proceedings of the 27th IEEE International Conference on Software Maintenance (2011)
15. Lawrie, D., Binkley, D., Morrell, C.: Normalizing source code vocabulary. In: Reverse Engineering (WCRE), 2010 17th Working Conference on, pp. 3 –12 (2010). DOI 10.1109/WCRE.2010.10
16. Lawrie, D., Feild, H., Binkley, D.: Extracting meaning from abbreviated identifiers. In: SCAM '07: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), pp. 213–222 (2007). DOI http://dx.doi.org/10.1109/SCAM.2007.9
17. Lawrie, D., Feild, H., Binkley, D.: Quantifying identifier quality: An analysis of trends. Journal of Empirical Software Engineering **12**(4) (2007)
18. Liblit, B., Begel, A., Sweetser, E.: Cognitive perspectives on the role of naming in computer programs. In: Proceedings of the 18th Annual Psychology of Programming Workshop (2006)
19. Madani, N., Guerrouj, L., Di Penta, M., Gueheneuc, Y., Antoniol, G.: Recognizing words from source code identifiers using speech recognition techniques. In: Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, pp. 68 –77 (2010). DOI 10.1109/CSMR.2010.31
20. Nie, J., Gao, J., He, H., Chen, W., Zhou, M.: Resolving query translation ambiguity using a decaying co-occurrence model and syntactic dependence relations. In: SIGIR '02: Proceedings of the 2002 SIGIR. ACM, New York, NY, USA (2002)
21. Ott, R.L., Longnecker, M.: An Introduction to Statistical Methods and Data Analysis, 5 edn. Duxbury (2001)
22. Runeson, P., Alexandersson, M., Nyholm, O.: Detection of duplicate defect reports using natural language processing. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, pp. 499–510. IEEE Computer Society, Washington, DC, USA (2007). DOI http://dx.doi.org/10.1109/ICSE.2007.32

A Instructions to Annotators

The following rather minimal instructions were given to the annotators when asking them to provide the oracle version of the split of the identifiers:

> What: Please split some program identifiers into atomic units by adding spaces. We consider atomic units to be individual words or abbreviations. Some splits are easily recognized from artifacts in the identifier. Those splits will be automatically inserted. Here are some examples:
> – "theblueHouse" → "the blue House"
> – "FDARequirement" → "FDA Requirement"
> – "unparse_voidptr" → "unparse void ptr"
> Some are easy. Some are hard. So let us know when you guess.
>
> Purpose: We are developing algorithms to automatically determine the most likely splits of program identifiers. An automatic identifier splitter is the first important step in a variety of automatic analysis of software natural language. Your splitting decisions will help to guide and evaluate our research on automatic identifier splitting. The split collection of identifiers will be made publicly available.
>
> Disclaimer: Your identity will not be revealed.
>
> Thanks for helping us out!
>
> Dave, Dawn, Emily, Lori, and Vijay