Motivation

The original goal of this research was investigate the effectiveness of document signature approaches for source code classification tasks. The idea was that, if an effective low-dimensionality representation of source code could be found that preserves all the necessary information, it could be used to assist in classification tasks when paired with a sufficiently large corpus of marked-up source code. This representation can then be translated into signature space, within which high performance retrieval and clustering approaches can be utilised for the purpose of creating a highly scalable source code classification service.

As applying signature techniques to classification tasks is a simple matter, the research portion of this task consists of determining the ideal representation of source code.

Data sets

The extent to which meaningful classification of source code can be achieved is limited by the training data we have available. In terms of the resources that were available to this project, the following three internal Oracle data sets were used:

Begbunch Accuracy

Begbunch is a collections of test data sets used internally at Oracle for benchmarking Oracle's *Parfait* static analysis tool. Begbunch is divided into two main sub-collections, of which one is the a*ccuracy* collection. As the name suggests, the *accuracy* collection is used for benchmarking the accuracy of Parfait. It consists of several suites of test programs which have had their source code marked up to indicate the location and nature of various bugs (such as buffer overflows, memory leaks and use of uninitialised data).

```
int main(int argc, char *argv[])
{
  int inc_value;
  int loop_counter;
  char buf[10];

  inc_value = 4105 - (4105 - 1);

  for(loop_counter = 0; ; loop_counter += inc_value)
  {
    if (loop_counter > 4105) break;
/* <bug buffer-overflow> */    buf[loop_counter] = 'A'/* </bug> */;
  }

  return 0;
}
```

Figure 1: Example of marked-up source code

Of the available data sets, this is the one that is most useful to this particular research problem, due to the high quality of the curated bug definitions. After some redundant or unnecessary suites were removed from the data set, the following test suites remained:

Test suite	Suite type	# of functions
Cigital	artificial	50
Iowa	artificial	1686
Parfait	real	547
Parfait-2	real	417
Parfait-examples	artificial	25
Samate	artificial	2366
unportable	real	162
Memory-leak	artificial	181

These suites can be further classified into two subtypes – artificial and real:

- The artificial examples consist of code that was created specifically to feature a particular bug. For example, a function that creates an array of a certain size and then sets an element outside the bounds of that array to test the analysis tool's ability to detect a buffer overrun.
- The real examples consist of code, usually from open-source projects, that happens to contain a particular bug. In many cases the surrounding code may be stripped away so that all that remains is the bugged function and a test harness to call it, but the code exhibiting the bug can be assumed to be real-world code.

Approximately 20% of the functions in the Begbunch Accuracy data set are real, while the remaining functions are artificial. For the purposes of our research the real data is the most helpful, as it identifies what real functions that exhibit particular vulnerabilities look like.

Begbunch Scalability

The other side to the Begbunch collection is the *scalability* set of test suites. These consist of a set of independently compilable software projects that are intended to test the execution time of Parfait; as a result, while the code may or may not contain bugs, there is no mark-up within the source files in this data set to indicate the presence of bugs. The size of the data set is considerable, with a total of 64,662 functions making up this data set; however, for it to be used as training or test data for any classifier, it needs to be marked up using some other approach (such as a static analysis tool like Parfait) first.

OpenSolaris

The largest of the data sets, the OpenSolaris collection is a copy of the source code from an OpenSolaris release from 2008. While the code has not been marked up like the Begbunch Accuracy data set has been, there is a certain amount of independently verified ground truth available for this set. This ground truth is in the form of 10,101 bugs that have been reported by a run of an old version of Parfait and manually evaluated for false positives, with bugs marked as

“verified”, “unverified”, “false” or “wontfix”. Two limitations prevent this data set from being quite as useful as the Begbunch *accuracy* data set:

- As the metadata consists of verified Parfait bugs only, bugs that were not caught by Parfait (false negatives) are not included.
- The metadata only lists bugs, not the files and functions that Parfait was run on. Hence it is impossible to tell if a particular file is unrepresented in the data because Parfait did not detect any bugs in it, or because the file was never processed by Parfait.

In addition to these limitations, due to the age of this particular OpenSolaris release and the complexities associated with setting up a complete build environment for it, many of the source files were not able to be recompiled for testing against updated static analysis tools. It is likely that many of the files unrepresented in the metadata were unable to be compiled at the time; however, the overlap between the functions that could be successfully recompiled and the files represented in the metadata is a lot smaller than is ideal. In total, there appear to be approximately 648,007 functions in the OpenSolaris data set; this was determined through source-only analysis (examining the source text for indicators that a function is present without need for preprocessing)

Feature Extraction

In order to perform rudimentary machine learning on the available source code and determine what information any signature representation will need to incorporate, it is necessary to establish a set of features that can be extracted that contain the information needed. Owing to the problems associated with determining the value of a feature without testing it, the initial approach taken in this instance was to extract a number of different features and progressively trim that list down until it reached a reasonable size.

Op-code n-grams

For code to be analysed with Parfait it first needs to be translated into LLVM’s intermediary bitcode format, which is an IL that can also be represented as an assembly language. This format also presents us with some useful cross-platform features that allow some of the control flow of a function to be captured without the syntax. In this case we make use of n-grams of the instruction op-codes (e.g. load, store, br), where 1-grams simply function as a histogram of how many times each instruction was used in a particular function, while 2-grams count how many times a particular sequence of two op-codes appear consecutively. While the amount of individual detail these features can give is low, the idea is that it may be possible that certain instructions or pairs of instructions present an elevated risk profile with respect to certain types of bugs; alternatively, the absence of certain instructions or pairs of instructions may make it impossible for certain types of bugs to exist in the code.

Code complexity features

The *complexity* tool was developed by Oracle alongside Parfait and its function is quite simple; on input of an LLVM bitcode file, it outputs a set of complexity metrics for each function in that file. These metrics relate to how complex the control flow of the code is, among other factors. The

rationale being using these features is the fact that they are relatively inexpensive to compute and, if common-sense associations between code complexity and code bugginess apply, these could be useful metrics for highlighting risky pieces of code.

Cyclomatic
Dataflow
DefUseChains
Edges
Effort
FuncEndLine
FuncStartLine
Knots
Length
Level
LineCount
Nesting
Operators
Vertices
Vocabulary
Volume

Text features

These features consist of tokens taken directly from the text of the source code from each function. This is performed with a custom tokenisation approach that attempts to preserve C language tokens (for example, keeping != and >> together) and the output acts as a histogram of each identifier and syntactical atom that appears in a given function. The rationale behind incorporating these features into the model is that there may be useful properties that only the original text can detect; for instance, when a buggy portion of code is duplicated.

This is an example of some of the text features extracted from the Begbunch accuracy data set:

! ( ) , 00 1 10 101 1024 12 1900 2 20 25 320x200 4G 55	CV_INT Con_DrawRuler2 Con_GetAlias Con_GetByte Con_GetCommand Con_GetFloat Con_GetVariable DDKEY_ENTER DEFAULT DEFAULT_LOG_FORMAT DHparams_dup DIAGstr EVP_PKEY_EC EXITCODE_OK Expired FILE FONT	In Input Logged MagickEpsilon MagickExport Mel Only PATH_SEP PKCS7_TEXT PlayMusic REPLY_LEDS_CMD RESERVED RXON_FILTER_ACCEPT_GRP_MSK RX_QUEUE_SIZE Record Rectify

<table>
<tr>
<td>553
8000
==
APLOG_ERR
APMG_CLK_REG_VAL_BSM_CLK_RQT
BAD
CV_CHARPTR
CV_FLOAT</td>
<td>FSM_COMPATIBILITY
FSM_IDADD
FSM_IDDEL
FSM_MACRO
HITN
HSP_TYPE_LOCAL
IEEE80211_CHAN_ANYC
ImageWidthsOrHeightsDiffer</td>
<td>SCD_DRAM_BASE_ADDR
SCD_INTERRUPT_MASK
SCD_QUEUE_RDPTR
SDSSC_OKAY
SGE_INTR_MAXBUCKETS
SGE_PL_INTR_MASK
SGE_RX_COPY_THRESHOLD
SLC_IP
SSL3_MT_CERTIFICATE_VERIFY
...etc…</td>
</tr>
</table>

Output from static analysis tools

Although these features clearly fail the test of being time-efficient to compute, it also made sense to use the results from running various static analysis tools (including Parfait) on the source code to generate features for that code. As well as providing data to benchmark this work against, it also opens the possibility of potentially combining multiple static analysis tools to produce better results than if only a single tool were used.

Static analysis tools used to provide additional features include:

- Parfait
- Splint
- Clang –analyze
- Uno

Dimensionality reduction

As the full set of features discussed in the previous section is considerable, it makes sense to develop some initial method of ranking them in terms of their representational capacity as far as their effectiveness at successfully classifying source code is concerned. This way the low-value features can be excluded without much additional work. To this end, the Leave One Out Nearest Neighbour Error (LOONNE) method is employed; the nearest neighbour error is calculated as the number of errors that exist when each function is classified as possessing the same bug as the nearest function in terms of Euclidean distance in the current feature space. For each feature in the current feature set, the nearest neighbour error is calculated for the feature set with that feature omitted. Through this process the feature that when removed results in the lowest nearest neighbour error for the remaining feature space can be determined. That feature is then permanently excluded and the process repeated until all features have been removed. The result is a ranked list of features in terms of their value, with the first-removed features being the least valuable.

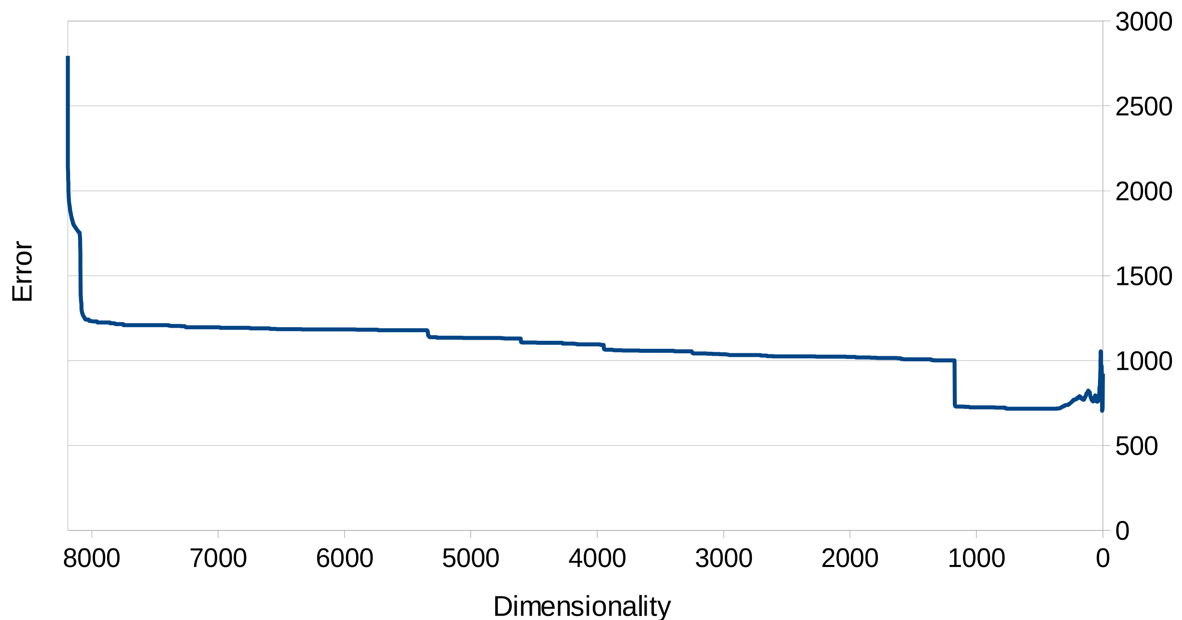


Figure 2: Progressively removing the least valuable feature at each iteration results in the global error shrinking, then increasing again once important features start disappearing

Most useful features

The output from the LOONNE tool is a ranked list of features. From this list we can have a look at the features that were found most useful for classifying functions in this data set:

Feature type	**Feature name**	**Rank**
Parfait	buffer-overflow	1
Text	_csl_to_argv	2
Splint	Undocumented global use	3
Splint	For body not block	4
Splint	buffer-overflow	5
Splint	If body not block	6
Splint	Path with no return in function with return type	7
Splint	Return value ignored	8
Parfait	Uninitialised var	9
Splint	Function exported but not used outside	10
Splint	Operands of comparison have incompatible types	11
LLVM 2-gram	load, fpext	12
Complexity	FuncStartLine	13
Splint	Test expression for if not boolean	14
Splint	Fresh storage not released before return	15
Complexity	Nesting	16
Text	!	17
LLVM 2-gram	add, call	18

The utility of some of these features is obvious; Parfait and Splint both report suspected buffer overflow bugs and there is clearly a strong correlation between this reporting and the presence of buffer overflow bugs in the code. Splint's “Fresh store not released before return” is also clearly indicative of a certain type of memory leak and a strong association with memory leaks can therefore be expected. For other features the association is far less clear.

Visualisation of feature subsets

To assist in understanding how the feature subsets are linked to the bug definitions, a visualisation tool was prepared that uses principal component analysis (PCA) to project vectors (functions in a given feature space) into a 2D space with pairwise distances preserved as much as possible. The PCA can also be used on 1D vectors (in the case where only one feature is being investigated) – this simply causes the points to be distributed along the diagonal.

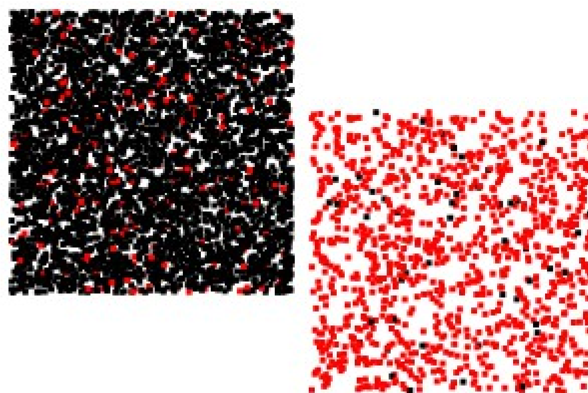


Figure 3: PCA of Parfait's buffer-overflow feature

As an example, Figure 3 depicts the relationship between Parfait's buffer-overflow feature and the buffer overflow bug. Each function in the data set is a point and a large amount of jitter has been added to the points so that the distribution of buggy and non-buggy functions in each group is clear (otherwise only two points would be visible, as there are only two possible values for the buffer-overflow feature.

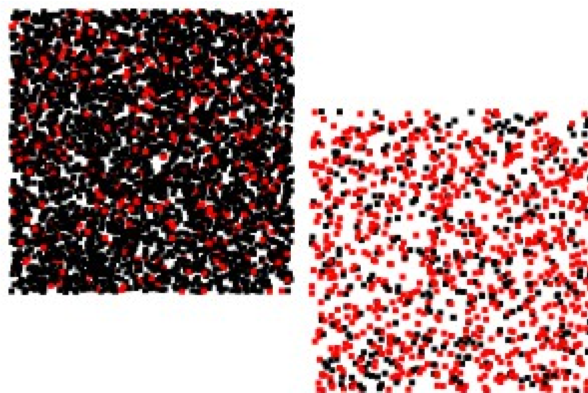


Figure 4: PCA of Splint's buffer-overflow feature

As another example, Figure 4 shows Splint's buffer-overflow feature. We can see from the comparison that Parfait's buffer overflow reporting is a good deal more useful than Splint's – both the red and black groups are considerably purer in Parfait's feature visualisation than Splint's. To

better illustrate the connection between the two, both Splint's and Parfait's buffer-overflow features can be examined at once.

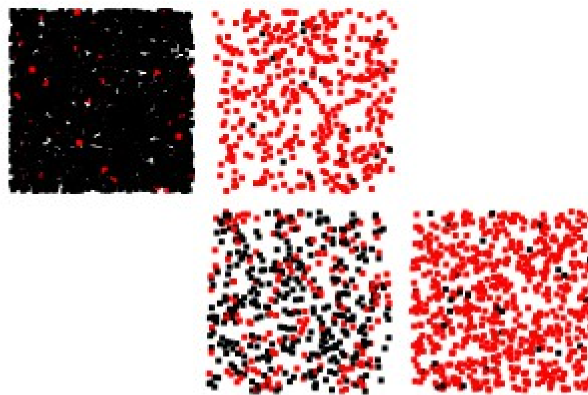
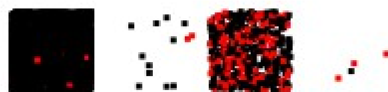


Figure 5: PCA of Parfait's and Splint's buffer-overflow features

As Figure 5 shows, this divides the functions into four categories. The most intriguing part of this image is the fact that the purest group is not the one in which Parfait and Splint both agree that a buffer overflow is present (the bottom right group), but instead the one in which Parfait claims a buffer overflow and Splint doesn't (the top right group). The functions that Splint finds a buffer overflow in and Parfait does not (the bottom left group) are the tricky ones – while less than half of the functions in this group actually have a buffer overflow, this group still contains many buffer overflow functions that were missed entirely by Parfait. This example shows how incorporating multiple features increases the confidence with which predictions can be made – not only can more accurate predictions be made as a result, but situations the algorithm is less confident about (such as when Splint reports a buffer overflow but Parfait does not) can be identified.



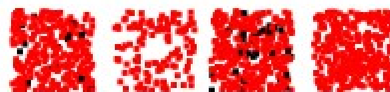


Figure 6: PCA of Parfait's, Splint's and Uno's buffer-overflow features

When the output from a third system – the static analysis tool Uno – is added,, even more comprehensive predictions can be made (Figure 6). In this visualisation the groups in the bottom row are the one Parfait reports a buffer overflow for and the top groups are the ones Parfait does not report a buffer overflow for. Within those rows, the leftmost group is not reported by either Splint nor Uno, while the rightmost group is reported by both. The second from the left is reported by Uno but not Splint and the second from the right is reported by Splint but not Uno. One thing that immediately stands out is that when Uno and Parfait both agree that a buffer overflow is present, there is a buffer overflow present 100% of the time, which is a very useful thing to know. While Parfait is still the best performing individual system, by combining these systems we can get both

answers and a level of confidence about how likely that answer is to be correct, given what we know from the data.

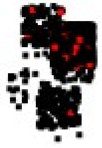


Figure 7: Parfait PCA (Buffer overflow)

Figure 8: Parfait PCA (Memory leak)

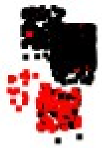


Figure 9: Parfait PCA (Read outside array bounds)

Figure 10: Parfait POCA (Uninitialised variable)

When multiple features cover different bug types, groupings around different bug types (or even different combinations of bug types, in instances where multiple bugs appear in the same functions) begin to emerge. Figures 7 through 10 show the result of using a PCA on all 14 features reported by Parfait (buffer-overflow, call-mismatch-extra-args, call-mismatch-types, deprecated-function, double-free, file-desc-leak, file-ptr-leak, file-ptr-not-init, integer-overflow, memory-leak, null-pointer-deref, read-outside-array-bounds, uninitialised-var, use-after-free). Unsurprisingly, the strongest correlation present is between each bug and the Parfait feature that results from detecting for that particular bug (uninitialised variables are most strongly associated with the uninitialised-var report from Parfait, for example).

Artificial data and real data

Figure 11: PCA of top-performing features (Buffer overflows highlighted)

As more and more features are added, patterns emerge and groups form that allow many different bug types to be identified (along with how confident the system is about their predictions) at a high rate of accuracy. Figure 11 shows the results of a PCA on the best-performing features identified by the LOONNE algorithm. Many of these small groups have very high purity and as expected these features allow a large number of buffer overflow errors to be accurately identified. Functions with different bug types also form into small groups. However, the level of purity can be misleading and the machine learning system can be tricked into believing that certain features are more useful than they actually are due to quirks in the data sets used.

As an example, there is a small cluster of points in Figure 11 - functions that contain buffer overflow and memory leaks that are not recognised by Parfait. A closer look at these points reveals that the functions – two from the Iowa dataset and one from the Memory-leak dataset – have very close or identical values for several features, including FuncStartLine, FuncEndLine, a number of identical reports from Splint and the 'store, store' LLVM instruction 2-gram. As it turns out, these different functions are actually practically identical implementations of a main function that allocates a block of memory and loops over it, going past the end of the allocated memory, then returns without freeing that memory. Hence the features that enabled this function to be identified were helped most by the fact that the function was almost identical. (to be continued...)

Machine learning

With the reduced set of features generated through the LOONNE approach, machine learning can be performed

TODO: Include results showing effectiveness with different sets of features and with different pairings of training / evaluation data

TODO: Discuss the architecture of these tools + how they work together (possibly)

TODO: Discuss preliminary investigations into CNN, RNN, LSTM work.

TODO: Discuss plans for going forward – how to resolve issues with available data, shortcomings we’ve identified with performing machine learning on source code in general