2020

# ControlFlag: A Self-supervised Idiosyncratic Pattern Detection System for Software Control Structures

Niranjan Hasabnis

Justin E. Gottschlich

# ControlFlag: A Self-supervised Idiosyncratic Pattern Detection System for Software Control Structures


## Abstract

Software debugging has been shown to utilize upwards of 50% of developers' time. Machine programming, the field concerned with the automation of software (and hardware) development, has recently made progress in both research and production-quality automated debugging systems. In this paper, we present ControlFlag, a system that detects possible idiosyncratic violations in software control structures. ControlFlag also suggests possible corrections in the event a true error is detected. A novelty of ControlFlag is that it is entirely self-supervised; that is, it requires no labels to learn about the potential idiosyncratic programming pattern violations. In addition to presenting ControlFlag's design, we also provide an abbreviated experimental evaluation.

# ControlFlag: A Self-supervised Idiosyncratic Pattern Detection System for Software Control Structures

**Niranjan Hasabnis**
Intel Labs

**Justin Gottschlich**
Intel Labs

## Abstract

Software debugging has been shown to utilize upwards of 50% of developers' time. *Machine programming*, the field concerned with the automation of software (and hardware) development, has recently made progress in both research and production-quality automated debugging systems. In this paper, we present ControlFlag, a system that detects possible idiosyncratic violations in software control structures. ControlFlag also suggests possible corrections in the event a true error is detected. A novelty of ControlFlag is that it is entirely *self-supervised*; that is, it requires no labels to learn about the potential idiosyncratic programming pattern violations. In addition to presenting ControlFlag's design, we also provide an abbreviated experimental evaluation.

## 1 Introduction

*Machine programming* (MP) is concerned with the automation of software (and hardware) development [11]. Recently, there has been a flurry of research in MP due to advances in machine learning, formal methods, data availability, and computing efficiency, amongst other things [3, 7, 8, 12, 14, 16, 20, 21]. In general, MP systems aim to improve programmer productivity as well as the quality of the software they produce. Some examples of recent MP tools are code recommendation systems [12], automatic bug detection systems [7], automatic generation of performance regression tests [1], and automatic completion of program constructs in integrated development environments (IDEs) [9, 17].

In this work, we present ControlFlag, a novel *self-supervised* framework that automatically identifies potential errors in code. In addition to identifying such potential errors, ControlFlag produces suggestions as corrections for the potential errors it finds. ControlFlag focuses on one specific class of bugs: those associated with *control structures*. Control structures are programming language structures that can alter a program's execution behavior, which are oftentimes the result of a logical condition being satisfied or not. Some examples of control structures in high-level languages such as C and C++ are: *(i)* selection statement: `if`, `else if`, `else` and `switch`, *(ii)* repetition statements: `for` loops, `while` loops, `do while` loops, *(iii)* jump statements: `goto`, `throw` statements. In this abbreviated presentation of ControlFlag, we focus principally on `if` statements to keep the problem scope tractable. Consider, for example, the following valid C++ code:

```
if (x = 7) y = x;
```

In the above code, it is *likely* that the programmer's intention is to set `y` equal to `x` if `x` is equal to `7`. Unfortunately, due to the omission of the second `=` in the `if` conditional expression, the condition is transformed into an assignment that sets `x` to the value of `7`, which subsequently returns true when evaluated and then sets `y` equal to `x`. The corrected code, would read as:

```
if (x == 7) y = x;
```

34th Conference on Neural Information Processing Systems (NeurIPS 2020), ML for Systems Workshop, Vancouver, Canada.

Moreover, it is important to note that in this particular case, we can only speculate that the code does not properly capture the programmer's intention. It is possible that the programmer's intention was in fact to set `x = 7` and then set `y = x` and to achieve this in an obfuscated fashion. Consequently, the correction provided by ControlFlag is also probabilistic. In fact, it is from the analysis of various repositories of code that ControlFlag makes a determination about whether a particular control structure is a potential error based on the commonality of its idiosyncratic pattern throughout the analyzed code repositories.

Identifying such a typographical coding errors is challenging for at least two reasons. First, the assignment of variables within conditionals is legal syntax in C/C++ (e.g., `if (x = 7)`). As such, this type of error might not be flagged by a compiler because the code's syntax is, in fact, legal. Second, compilers and static analyzers can use data-flow analysis to identify such errors[1], but data-flow analyses have their own limitations (e.g., locally scoped versus globally scoped analyses, etc.). Nonetheless, compilers such as GCC and LLVM already use a rules-based approach to warn programmers in a variety of potentially erroneous cases. For instance, GCC-10.2.0's `-Wall` option warns programmers of the above code as:

```
test.cpp:3:9: warning: suggest parentheses around assignment used as truth value
[-Wparentheses]
                          if (x = 7) y = x;
                              ~~^~~
```

However, rules-based approaches to identifying such errors tend to have at least two associated limitations. First, they can be ***labor-intensive.*** New rules generally need to be added to the systems (such as compilers or static analyzers) to flag new types of potential errors [7]. Second, they may ***require a compilable program.*** Compiler-based warnings tend to require code that is compilable to flag such issues. Moreover, it seems unlikely that such compiler-based approaches may be practical as a recommendation system that dynamically identifies potential issues in a live programming environment, where code in such environments is often not compilable.

In this preliminary paper, ControlFlag takes a statistical approach to identify typographical errors in program code by recasting it as an anomaly detection problem. We hypothesize that *using certain patterns (such as assignment) inside an* `if` *statement in C/C++ language is relatively rare*. We test this hypothesis by mining idiosyncratic patterns found in the control structures of C/C++ programs found in open-source GitHub repositories. The mined patterns form a *dictionary* that is then used to check a user's typed pattern and suggest automatic corrections in case of divergence. An advantage of such an approach is that it does not require labeled training data — ControlFlag uses the mined patterns from *semi-trusted* GitHub repositories in a *self-supervised* fashion, eliminating the need to know all the possible typographical errors.

Although we used typographical error detection as a motivating example above, ControlFlag's approach of learning idiosyncratic patterns and checking for divergence with respect to them has several applications. Learning typing rules that associate operators with types for a programming language is another application. These rules can be used to flag anomalies related to the usages of types. Take, for instance, `if (keepon > true)` anomaly that ControlFlag flagged in CURL open-source project. The anomaly was flagged based on a learned rule that captured the observation that, typically, > operator in C (or any other language for that matter) is used with numeric types and not boolean. We raised this anomaly to CURL developers and received an acknowledgment along with a fix (more details in Section 3). ControlFlag could also learn that checking for a pointer being NULL before a dereference is a typical programming pattern and can flag missing NULL pointer checks.

This preliminary paper makes the following technical contributions:

- We present ControlFlag, which, to our knowledge, is the first-of-its-kind approach that is entirely *self-supervised* in its ability to learn idiosyncratic pattern violations for a given code repository, which may (or may not) indicate erroneous code.

[1] If we compile above code with -O2, a compiler can eliminate `if` statement and replace it by `x = 7; y = 7;`. A compiler can flag the removal of `if` to the user.

- Although we only demonstrate ControlFlag for C/C++, we have designed to be programming language agnostic such that it should be capable of adapting itself to learn the idiosyncratic signatures of any type or form of control structure in any given programming language.
- We present initial results of ControlFlag's ability to identify idiosyncratic pattern violations in `if` statements of C/C++ programs. These results span $\approx 6,000$ GitHub repositories and are checked against nominal patterns from popular open-source packages of OpenSSL and CURL. The anomaly flagged in CURL led to a source code change, which we believe improves the robustness of the code. Overall, our findings revealed interesting results, which suggest possible improvements and future work for ControlFlag.

## 2 ControlFlag Design

Figure 1 shows the overview of ControlFlag, which consists of two main phases: *(i)* pattern mining and *(ii)* scanning. ControlFlag's *pattern mining* phase consists of learning the common (and uncommon) idiosyncratic coding patterns found in the user-specified GitHub repositories, which, when complete, generates a precedence dictionary that contains acceptable and unacceptable idiosyncratic patterns. ControlFlag's *scanning* phase consists of analyzing a given source code repository against the learned idiosyncratic patterns dictionary. When anomalous patterns are identified, ControlFlag notifies the user and provides them with possible alternatives.

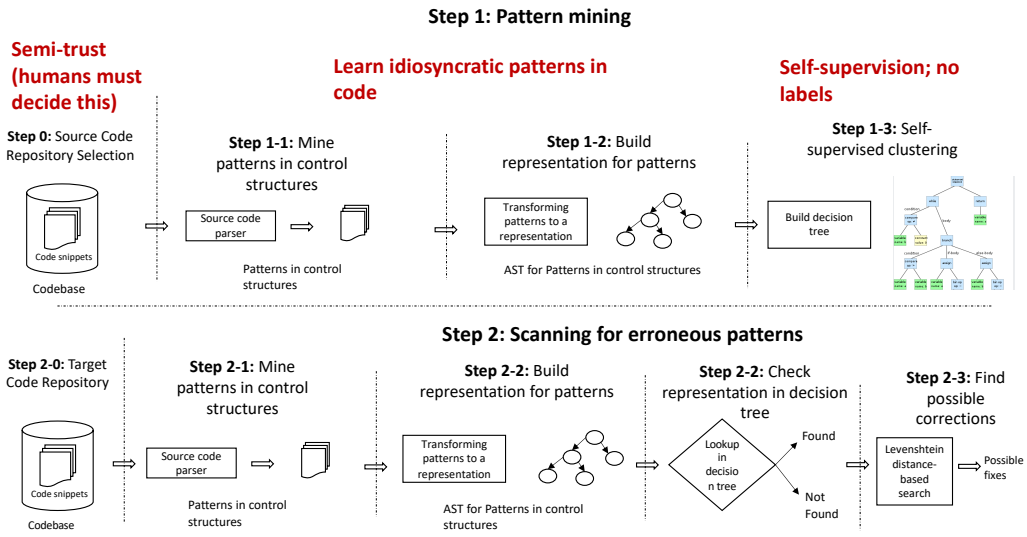



Figure 1: Overview of ControlFlag System.

### 2.1 Pattern Mining

The pattern mining phase is further divided into the following sub-phases: *(i)* source repository selection, *(ii)* code parsing, and *(iii)* decision tree construction.

**Source repository selection.** In our experiments, we used the number of stars of a GitHub repository as a measure of its quality. Without any easy and quantifiable measure of the quality of the programs in those repositories other than GitHub stars, we believe that such a selection gives us semi-trust in the data. We, however, believe that this is an open problem as quality is a subjective term.

**Parsing control structures.** After a source repository is selected, ControlFlag generates a list of programs in the high-level programming language of interest. Every program from this list is then parsed. The resulting parse trees – currently in the form of abstract syntax trees (ASTs) – are then converted into a higher-level abstracted tree to prune unnecessary details from ASTs. While ControlFlag parses these programs, it does not discard programs containing parse errors. For

ControlFlag's purpose, all that is needed are control structures do not have parse errors; parse errors elsewhere in the code may not impact ControlFlag's functionality. We believe this characteristic of ControlFlag is important, given that we have found that the majority of the C/C++ programs found in open-source are not compilable. Additionally, such an approach would enable ControlFlag to eventually be used in a live programming environment or an integrated development environment (IDE).

**Levels of abstractions.** We used TreeSitter APIs [18] to parse the control structures in the source programs and used its AST representation[2] as the lowest-level tree representation (referred to as an L1 abstraction level). But during the experimentation, we found that AST representation would capture exact structural form of the expressions inside the control structures. For instance, for expression from C/C++ language such as `a.b.c & (1 « e))`, the AST would look like:

```
(binary_expr ("&")
  (field_expr (field_expr ((identifier)(field_identifier)))(field_identifier))
  (parenthesized_expr (binary_expr ("<<")(number)(identifier))))
```

We observed that capturing such a precise structural form would later reduce the chances of matching a target pattern during the scanning phase. The higher-level abstraction (referred to as an L2[3] abstraction level), developed on top of an AST, drops the precision in the structural form by keeping the ASTs to a finite and small height by introducing a new tree node type (named `non_terminal`) to represent a pruned tree. The higher-level tree for the aforementioned AST would then take the form:

```
(binary_expr ("&") (non_terminal_expr)(non_terminal_expr))
```

Note that the higher-level tree above drops both the children of `&` and marks them as `non_terminal`. This would intuitively increase the number of false negatives. Nonetheless, it allows ControlFlag to check for idiosyncratic violations in `&` during the scanning phase rather than declaring the whole pattern as absent from the dictionary.

**Decision tree construction.** During parsing, a pattern at both the L1 and L2 abstraction levels can be dumped out in textual format using a pre-order tree traversal. In fact, we represent the mined patterns internally by building a prefix tree[4] (or trie [4]) over the text strings for the trees of the mined patterns. Every path in the prefix tree that ends with a terminal node, corresponding to a valid pattern, also stores the number of occurrences of that pattern. We build different prefix trees for text strings that are at different levels of abstractions. By doing this, we can convert idiosyncratic patterns into different abstraction levels and check them against their respective prefix trees.

### 2.2 Scanning for Unusual Patterns

When a user specifies a target repository to scan for unusual patterns, ControlFlag first obtains a list of idiosyncratic patterns that occur in the specified control structures, which it had learned from the prior mining phase. For every pattern in the list, it builds a respective parse tree at an L1 abstraction level and checks it against the prefix tree at L1 level. If the pattern is present in the tree, then it is considered a non-violation and the search terminates. If the pattern is missing, then ControlFlag checks it against the prefix tree at L2 level. ControlFlag triggers an automatic correction phase irrespective of whether a pattern is found or not found (at L1 and L2 levels). In the former case, although the pattern was found, it could be rare; in the latter case, it is known to be rare, because it is absent. ControlFlag's automatic correction phase then either flags the pattern as anomalous or not based on a heuristically calculated threshold. If the pattern is flagged as anomalous, ControlFlag also suggests possible corrections based on tree similarity criteria.

**Automatic correction.** Automatic correction of strings [4] is a well-studied problem in computer science. In our first embodiment of ControlFlag, we use edit distance between strings (using dynamic programming algorithm) to suggest possible corrections of the target string. For future embodiments, we plan to explore other forms of similarity and dissimilarity analysis.

[2]To keep the description brief, we do not specify its grammar here.

[3]We use L1 and L2 terminology from the concept of caches in computer architecture. An item missing at L1 has a high chance of being at L2.

[4]Given the preliminary nature of this study, we found that prefix tree structure performed reasonably well (time complexity of search and space usage) in our experiments. And we did not invest in optimizing it to reduce the space usage, but such an optimization using deterministic finite automaton (DFA) is possible.

We experimented with three approaches to suggest corrections of a possibly erroneous target pattern: a naive approach, a Norvig et al. auto-correction approach [13], and Symmetric Delete [10][5]. For our purposes, the Norvig et al.'s approach and symmetric delete did not perform well and encountered temporal and spatial complexity that was super-linear compared to the naive approach. As the temporal and spatial complexity of the naive approach is reasonable compared to other approaches, we used it for our evaluation. We have, nonetheless, optimized it in the following obvious ways: *(i)* caching the results of automatic correction process, *(ii)* compacting the string representation of the parse trees by using short IDs for unique tree nodes, and *(iii)* employing parallelism while traversing the prefix trees.

**Ranking the results of automatic correction.** The outcome of the automatic correction process is a list of possible corrections at various edit distances and their occurrences in the dictionary for a given target string. When ControlFlag presents the results of the automatic correction to the user, it first sorts them in the increasing order of edit distance and secondly by the number of occurrences that can correct the string given some `k` number of edits. This simple heuristic is based on the intuition that the probability of a typographical violation of one character has a greater probability than a typographical violation in more than one character. ControlFlag uses the sorted results of the automatic correction process to determine a threshold level to define what is and is not anomalous.

**Anomaly threshold.** ControlFlag uses two criteria for the purpose of declaring a certain pattern as anomaly. First, a target pattern that is missing from the dictionary is declared anomalous because it is missing. Second, if a target pattern is not missing from the dictionary, but its automatic correction results satisfy the following formula is declared anomalous.

$$\frac{n_0 \times 100}{\sum_{i=0}^{max\_cost} max(n_i)} < \alpha \quad \forall (p, n) \in C$$

$C$ is the set of automatic correction results, in which every result contains a corrected pattern $p$ and its occurrences $n$. $\alpha$ is a user-defined anomalous threshold, and $max$ is a function that calculates the maximum of a list of occurrences.

Intuitively, we calculate the percentage contribution of the number of occurrences of a possible incorrect target pattern ($n_0$) against the maximum number of occurrences at every edit distance. In other words, if the possible corrections of a target string at smaller edit distances (such as 1) occur more frequently than the target string, then it is likely that the target string is an idiosyncratic violation. $\alpha$ is the anomaly threshold that is user controllable, and, by default, it is set to $5\%$. Reducing the anomaly threshold reduces the number of flagged anomalies.

## 3 Experimental evaluation

In this section, we present preliminary results of ControlFlag in flagging idiosyncratic pattern violations in `if` statements of C/C++ programs.

### 3.1 Setup

**Software and hardware setup.** All the experiments in this section were performed on an Intel 56-core Xeon Platinum 8280 CPU with hyper-threading enabled and $\approx$ 200GB of memory. The operating system and compiler used was CentOS-7.6.1810 with GCC-10.2.

**Source repository selection.** For the pattern mining phase, we chose the top 6000 open-source GitHub repositories that used C/C++ as their primary programming language and had received at least 100 stars. As we mentioned previously, we use GitHub stars as a mechanism to infer quality of software ControlFlag is trained on.

**Target repository selection.** In our experiments, we used `OpenSSL-1.1.1h` and `CURL-7.73` repositories to scan for patterns violations. There was no particular reason to choose these packages besides the fact that both of them are popular open-source software packages and use C as their primary language.

[5] As the details of these approaches are not necessary to understand the core concepts of the main paper, we place a more detailed analysis of these techniques in Appendix B.

### 3.2 Results

**Mining patterns from source repositories.** The 6000 repositories that we used for the pattern mining phase consisted of a total of 2.57M C/C++ programs. These programs had ≈ 38M total patterns, ≈ 831K unique patterns at the L1 abstraction level, and precisely 468 unique patterns at the L2 abstraction level. Figure 2 shows the cumulative percentage plot of the number of occurrences of unique patterns at both the abstraction levels. As expected, ≈ 90% of the unique patterns at the L1 level have low frequency (close to 10). At L2 abstraction level, however, because of the grouping of multiple patterns at L1 level, ≈ 90% of the patterns have higher frequency. We provide a more detailed analysis of pattern frequency in Appendix A.

Mining patterns from 6000 source repositories with 56 worker threads took around two hours, and building prefix trees at the L1 and L2 abstraction levels from those patterns took close to three minutes. We also dumped the patterns at the L1 and L2 levels in a textual format into a file, which was 11GB in size. The memory consumption of ControlFlag after building the prefix trees was ≈ 8GB, which has a reduced spatial footprint due to compression performed by prefix tree.

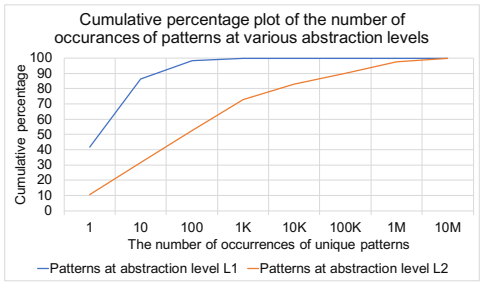



Figure 2: Cumulative percentage plot of the number of occurrences of unique patterns at different abstraction levels.

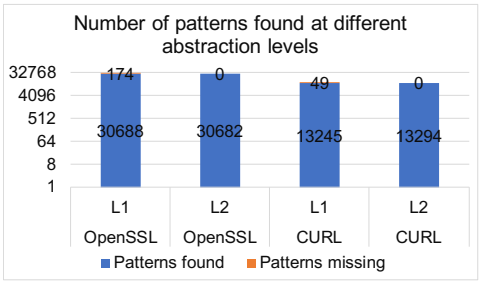
**Scanning target repositories.** We scanned 1212 C/C++ programs from OpenSSL and 736 C/C++ programs from CURL to flag potential idiosyncratic violations in `if` statements. With 56 scanner threads, ControlFlag took ≈ 3 hours to scan OpenSSL and 30 minutes to scan CURL. Figure 3 shows the results. In total, ControlFlag scanned 30,862 patterns in OpenSSL and 13,294 patterns in CURL. [6]

Figure 3a shows the effect of having two abstraction levels on the number of patterns that are found in the prefix trees. In summary, all the patterns are found at the L2 level, while a few are missing at the L1 level. Figure 3b shows the number of patterns flagged as anomalies at the anomaly thresholds of 1% and 5%. As expected, the number of anomalies flagged at 5% are higher than that at 1%. This raises an obvious and deeply studied question: what should the anomalous threshold be set to? We could set the threshold to a value smaller than 1%, especially, since Figure 2 shows that many patterns have low frequencies at the L1 level. Yet, because the absolute number of anomalies flagged at the L1 level (after removing duplicates) are reasonable for a manual inspection, we chose not to experiment with reducing the anomaly threshold further (at least for this early work).

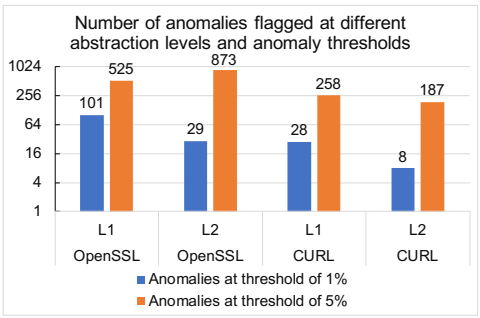
**Anomalies flagged in the scans.** Now we discuss some of the anomalous pattern violations ControlFlag found in OpenSSL and CURL. Detailed report of these anomalies along with their automatic correction results are discussed in Appendix A. Note that we have not yet confirmed if these anomalies are bugs — we can only confirm if that is the case by applying the changes suggested in the automatic corrections and running the automated tests or contacting developers.

***Anomaly 1.*** CURL's `lib/http_proxy.c` uses `s->keepon > TRUE` expression at line number 359. This was flagged as anomalous because there were only 4 source patterns that contained a boolean value `TRUE` in `>`. ControlFlag's top suggested correction `s->keepon > number` at edit distance of 2 had 127K occurrences. We found that `s->keepon` is of type `int` in CURL, while `TRUE` is defined as `true`, which is defined as integer `1` in C language. So this expression is a comparison between a signed 32-bit integer and an integer value of a boolean, which is why GCC did not flag it. We believe that `> true` expression, however, is ambiguous for two reasons: boolean values are typically used with logical or bitwise operators, and in C language, any non-zero value is considered as `true`. We conveyed this ambiguity to CURL developers [6] and proposed `s->keepon > 1` as a better expression. They acknowledgement the ambiguity and resolved it [5] by using `enum` type to

[6] Duplicates were not removed. We suspect the total number of patterns would decrease if duplicates were removed.




(a) The number of patterns that are found and missing at various abstraction levels




(b) The number of patterns that are flagged as anomalies at various abstraction levels and anomaly thresholds

Figure 3: Results of scanning OpenSSL and CURL for possible typographical errors

model different values of `keepon` as states. We find it motivating to see that an anomaly flagged by ControlFlag improved the robustness of CURL by eliminating a potential ambiguity.

***Anomaly 2.*** OpenSSL's `test/testutil/tests.c` uses expression `(s1 == NULL) ^ (s2 == NULL)` at line number 268. The expression was flagged as anomalous because it had only 8 occurrences. While the top two suggested corrections `(s1 == NULL) | (s2 == NULL)` at edit distance 1 and `(s1 == NULL) || (s2 == NULL)` at edit distance 2 had 32 and 6808 occurrences in the dictionary, respectively.

***Anomaly 3.*** OpenSSL's `test/evp_test.c` uses expression `(-2 == rv)` at line number 1898. It was flagged as anomalous because it had 16529 occurrences, while its possible correction `variable == rv` at edit distance 1 had 1.1M occurrences. We believe that the base expression has lower occurrences because fewer programmers use that style of writing an equality check. We, however, believe that `number == variable` is a better style than `variable == number`, as it avoids possible typographic errors because compiler's lvalue check will prevent assignment to a constant.

***Anomaly 4.*** OpenSSL's `crypto/x509/t_req.c` uses expression `(BIO_puts(bp, ":") <= 0)` at line 141. This expression was flagged as anomalous as it had 475 occurrences. What we find interesting about this expression is that it compares the result of a function call with 0 and negative values, which is OpenSSL's approach for evaluating error codes. ControlFlag's top two suggested corrections were `(BIO_puts(bp, ":") == 0)` and `(BIO_puts(bp, ":") < 0)`, which, based on the data we analyzed, seemed to indicate more appropriate patterns — 0 being a successful return code (as in standard libc) and comparison with the negative values for erroneous return codes. OpenSSL's expression somehow combines both the typical patterns together, resulting in a highly abnormal combination.

## 4 Related work

Automated program repair and automated bug detection and correction are growing and active areas of research in MP [1, 2, 7, 15, 19, 20]. Most of these techniques rely on a learning based approach to detect and fix bugs. Hoppity [7], in particular, is a recent work that uses deep learning model to detect and correct a variety of bugs in JavaScript code. They define their bug detection problem as an anomaly detection problem: if a code fragment deviates from the fragments seen during model training, then that code fragment is considered buggy. AutoPerf [1] takes a similar approach using autoencoders to detect anomalous behaviors in program performance that exceed some learned threshold.

ControlFlag is different than these approaches in that it is not specific to detecting bugs. An anomaly flagged by ControlFlag may or may not be a bug — this largely depends upon the accepted idiosyncratic patterns within a given program's source code. In this sense, ControlFlag can notify programmers of anomalies, even before test cases or a program specifications are checked. To our knowledge, ControlFlag may be the first of its kind to identify typographical anomalies, which may be erroneous, based entirely on a self-supervised learning system.

## 5 Conclusion

In this preliminary study, we presented ControlFlag, a system to automatically detect possible typographical errors in the control structures of high-level programming languages. ControlFlag also suggests possible corrections to such errors. The fundamental approach ControlFlag takes is to recast typographical errors as anomalies, where a self-supervised system that is trained on a large enough semi-trusted code, will automatically learn which idiosyncratic patterns are acceptable and which are not. Our initial findings from scanning C/C++ programs from OpenSSL and CURL across 2.57 million programs reveal interesting anomalies (as well as some unusual programming styles). Concretely, the anomaly flagged by ControlFlag in CURL was acknowledged by the developers and was fixed promptly. We plan to use our initial findings as guidance for future refinement of the system.

## References


[1] Mejbah Alam, Justin Gottschlich, Nesime Tatbul, Javier S Turek, Tim Mattson, and Abdullah Muzahid. A Zero-Positive Learning Approach for Diagnosing Software Performance Regressions. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlchBuc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, NeurIPS 2019, pages 11623–11635. Curran Associates, Inc., 2019.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs, 2018.

[3] Lujing Cen, Ryan Marcus, Hongzi Mao, Justin Gottschlich, Mohammad Alizadeh, and Tim Kraska. Learned garbage collection. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2020, page 38–44, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[5] CURL. http_proxy: use enum with state names for 'keepon'. `https://github.com/curl/curl/pull/6193`, 2020.

[6] CURL. Re: Potential confusion in http_proxy.c and a recommendation. `https://curl.se/mail/lib-2020-11/0028.html`, 2020.

[7] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *International Conference on Learning Representations*, 2020.

[8] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. An abstraction-based framework for neural network verification. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 43–65, Cham, 2020. Springer International Publishing.

[9] Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachi Nagappan, and Ashish Tiwari. Feedback-driven semi-supervised synthesis of program transformations. In *OOPSLA*. ACM, October 2020. Conditionally accepted.

[10] Wolf Garbe. Symspell. `https://github.com/wolfgarbe/SymSpell`, 2020.

[11] Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B. Tenenbaum, and Tim Mattson. The Three Pillars of Machine Programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 69–80, New York, NY, USA, 2018. ACM.

[12] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: Code Recommendation via Structural Code Search. *Proc. ACM Program. Lang.*, 3(OOPSLA):152:1–152:28, October 2019.

[13] Peter Norvig. How to write a spelling corrector. `https://norvig.com/spell-correct.html`, 2016.

[14] Augustus Odena and Charles Sutton. Learning to represent programs with property signatures. In *International Conference on Learning Representations*, 2020.

[15] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[16] Alexander Ratner, Dan Alistarh, Gustavo Alonso, David G. Andersen, Peter Bailis, Sarah Bird, Nicholas Carlini, Bryan Catanzaro, Jennifer Chayes, Eric Chung, Bill Dally, Jeff Dean, Inderjit S. Dhillon, Alexandros Dimakis, Pradeep Dubey, Charles Elkan, Grigori Fursin, Gregory R. Ganger, Lise Getoor, Phillip B. Gibbons, Garth A. Gibson, Joseph E. Gonzalez, Justin Gottschlich, Song Han, Kim Hazelwood, Furong Huang, Martin Jaggi, Kevin Jamieson, Michael I. Jordan, Gauri Joshi, Rania Khalaf, Jason Knight, Jakub Konečný, Tim Kraska, Arun Kumar, Anastasios Kyrillidis, Aparna Lakshmiratan, Jing Li, Samuel Madden, H. Brendan McMahan, Erik Meijer, Ioannis Mitliagkas, Rajat Monga, Derek Murray, Kunle Olukotun, Dimitris Papailiopoulos, Gennady Pekhimenko, Theodoros Rekatsinas, Afshin Rostamizadeh, Christopher Ré, Christopher De Sa, Hanie Sedghi, Siddhartha Sen, Virginia Smith, Alex Smola, Dawn Song, Evan Sparks, Ion Stoica, Vivienne Sze, Madeleine Udell, Joaquin Vanschoren, Shivaram Venkataraman, Rashmi Vinayak, Markus Weimer, Andrew Gordon Wilson, Eric Xing, Matei Zaharia, Ce Zhang, and Ameet Talwalkar. Mlsys: The new frontier of machine learning systems, 2019.

[17] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 2727–2735, New York, NY, USA, 2019. Association for Computing Machinery.

[18] TreeSitter Github team. An incremental parsing system for programming tools. `https://tree-sitter.github.io/tree-sitter/`, 2017.

[19] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair, 2019.

[20] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning (ICML)*, 2020.

[21] Fangke Ye, Shengtian Zhou, Anand Venkat, Ryan Marcus, Nesime Tatbul, Jesmin Jahan Tithi, Niranjan Hasabnis, Paul Petersen, Timothy Mattson, Tim Kraska, Pradeep Dubey, Vivek Sarkar, and Justin Gottschlich. Misim: A novel code similarity system, 2020.

| AST | Occurrences | Example C expressions |
|---|---|---|
| `(identifier)` | 4.3M | `if (x)` |
| `(unary_expr (“!”) (identifier))` | 2.09M | `if (!x)` |
| `(field_expr (identifier)(field_identifier))` | 1.3M | `if (p->f)` |
| `(binary_expr (“==”) (identifier)(identifier))` | 1.16M | `if (x == y)` |
| `(binary_expr (“<”) (identifier)(number))` | 1.13M | `if (x < 0)` |
| `(binary_expr (“==”) (identifier)(number))` | 1.09M | `if (x == 0)` |
| `(call_expr (identifier)(arg_list (identifier)))` | 1.05M | `if (foo(x))` |
| `(binary_expr (“==”) (identifier)(null))` | 790K | `if (p == NULL)` |
| `(binary_expr (“==”) (field_expr (identifier) (field_identifier))(identifier))` | 732K | `if (p->f == y)` |
| `(binary_expr (“!=”) (identifier)(identifier))` | 636K | `if (x != y)` |

Table 1: A table showing the most frequently occurring patterns at L1 abstraction level

| AST | Occurrences | Example C expressions |
|---|---|---|
| `(binary_expr (“=”) (identifier)(number)))` | 487 | `if (x = 0)` |
| `(binary_expr (“=”) (identifier)(identifier)))` | 476 | `if (x = y)` |
| `(binary_expr (“=”) (identifier)(call_expr (identifier)(arg_list))))` | 356 | `if (x = foo(y))` |
| `(binary_expr (“%”) (identifier)(number)))` | 6468 | `if (x % 2)` |
| `(binary_expr (“\|”) (identifier)(identifier)))` | 1137 | `if (x \| y)` |
| `(binary_expr (“^”) (identifier)(identifier)))` | 813 | `if (x ^ y)` |
| `(binary_expr (“==”) (number)(number)))` | 236 | `if (0 == 0)` |

Table 2: A table showing some of the patterns with less than 1% occurrences at L1 abstraction level

# A Appendix: Evaluation results

## A.1 Analysis of source patterns

Table 1 shows top 10 frequently occurring patterns, the types of expressions in C language they correspond to, and the number of occurrences at the L1 abstraction level. Most of these patterns are expected; it is also good to see the NULL check in the list — in our opinion, it talks about good programming practice. Table 2, on the other hand, shows patterns, such as `if (x = 7)`, that have less than 1% occurrences of 38M. It is interesting to see bitwise operators in C, such as `|` and ^, in the list. We believe that these operators are more common in low-level code that operates close to hardware. This observation also suggests that the selection strategy for source repositories could be different in which we consciously ensure a uniform mix of repositories that contain code belonging to different technical domains or areas.

## A.2 Flagged anomalies and theirs possible corrections

Below we show some of the interesting anomalies found while scanning OpenSSL and CURL packages. The results also contain the suggested corrections and their occurrences.

---

Potential anomaly: `((s1 == NULL) ^ (s2 == NULL))`
Location: `openssl-1.1.1h/test/testutil/tests.c:268`
Possible corrections:
`((s1 == NULL) ^ (s2 == NULL))`, edit distance 0, occurrences 8
`((s1 == NULL) | (s2 == NULL))`, edit distance 1, occurrences 32
`((s1 == NULL) || (s2 == NULL))`, edit distance 2, occurrences 6808
`((s1 == NULL) && (s2 == NULL))`, edit distance 2, occurrences 521

---

Potential anomaly: `( -2 == rv )`
Location: `openssl-1.1.1h/test/evp_test.c:1898`
Possible corrections:
`( -2 == rv )`, edit distance 0, occurrences 16529
`( variable == rv )`, edit distance 1, occurrences 1164852
`( -2 != rv )`, edit distance 1, occurrences 6483
`( -2 <= rv )`, edit distance 1, occurrences 2170
`( -2 >= rv )`, edit distance 1, occurrences 265

---

Potential anomaly: `(BIO_puts(bp, ":") <= 0)`
Location: `openssl-1.1.1h/crypto/x509/t_req.c:141`
Possible corrections:
`(BIO_puts(bp, ":") <= 0)`, edit distance 0, occurrences 475
`(BIO_puts(bp, ":") == 0)`, edit distance 1, occurrences 80350
`(BIO_puts(bp, ":") != 0)`, edit distance 1, occurrences 4559
`(BIO_puts(bp, ":") < 0)`, edit distance 1, occurrences 1431

---

Potential anomaly: `(s->keepon > TRUE)`
Location: `curl/lib/http_proxy.c:359`
Possible corrections:
`(s->keepon > TRUE)`, edit distance 0, occurrences 4
`(s->keepon > number)`, edit distance 2, occurrences 127540
`(s->keepon > variable)`, edit distance 2, occurrences 56475

---

## B Appendix: Approaches for automatic correction

In this section, we provide a brief and informal description of all three approaches that we evaluated to automatically suggest possible corrections to an erroneous pattern. Since these approaches are not the contribution of this paper, we do not provide a formal description.

For the sake of comparing these approaches, let us consider that the parameters for an automatic correction algorithm consist of (1) target string of length $N$ and its correction of length $M$, (2) a dictionary consisting of $D$ strings among which to search for possible corrections that are within the edit distance of $E$, and (3) the target string and its corrections draw characters from a vocabulary set of size $V$.

- ***A naive approach.*** A naive approach to look for corrections of a target string against a dictionary would be to calculate the edit distance between the target string and every string from the dictionary. In other words, this approach traverses the prefix tree completely.

  The time complexity of this naive approach is linear to the size of the dictionary, and more precisely, it is $O(N \times M \times D)$, where $M$ is the average size of the strings from $D$.

- ***Norvig et al. approach.*** An alternative approach suggested by Norvig et al. [13] eliminates the need to go over all the strings from the dictionary to find possible corrections. Instead, it relies on generating candidate correction strings within the given maximum edit distance from the target string. The candidate correction strings are then checked against the dictionary. If a candidate is found in the dictionary, then it is a possible correction of the target string.

  The time complexity of Norvig et al. algorithm, however, grows exponentially in the order of the edit distance. Specifically, the number of candidate correction strings that are at edit distance of 1 from the target string are $O(V \times N)$, considering typical typing corrections such as insertion, deletion and replacement of a single character. In order to calculate the candidate correction strings at edit distance of 2, all of the candidate strings at edit distance 1 go through typing corrections of a single character. In other words, the number of candidate corrections at edit distance of 2 would be $O(V \times N^2)$. This approach works in practice when $N$ is small (for English language, average value of $N$ is 5) and hence $E$ is at max $N$

(typically, in practice, $E$ is 2 or 3 for English language). In our experiments, we found that with a vocabulary size of 50 and the target string of length 80, the algorithm generates 8000 candidates at edit distance 1 and 2M candidates at edit distance 2, out of which less than 5% would be valid candidates.

- ***Symmetric Delete approach.*** Symmetric Delete approach [10], introduced by Garbe et al., is another correction candidate generation approach that relies on using character deletes only as edits for generating candidates from the target string. Specifically, it observes that the approach by Norvig et al. produces an exponential number of candidates in the edit distance because it considers insertions, deletions and replacements as edits. It avoids the exponential number of candidates of a target string by only considering character deletions as edit operations. Although the number of candidates generated from a target string of length $N$ are still upper-bounded by $O(N!)$, they are independent of the vocabulary size $V$. And this is why symmetric delete approach generates far less correction candidates than the approach by Norvig et al.

  The downside of the symmetric delete approach is that it has to generate similar correction candidates for all the strings from the dictionary. The correction candidates generated using the dictionary are then compared with the candidates generated using a target string to suggest possible corrections to the target string. In other words, it trades the memory to store the correction candidates generated from the dictionary to reduce the time to find possible corrections. In fact, the correction candidates computed from the dictionary can be pre-computed. The space required to store the pre-computed correction candidates, however, is $O(D \times M!)$, and it proved prohibitive in our case as we increased the number of source repositories.