Using the *Strings* Metadata to Detect the Source Language of the Binary

Ashish Adhikari and Prasad A. Kulkarni

EECS, University of Kansas, USA
`adhikariashish2@ku.edu; prasadk@ku.edu`

Abstract. We explore the question of determining the source language of program binaries. Programs compiled from different source languages are susceptible to different classes of software attacks. Therefore, knowing the source language can help the end-users assess the security risk of the binary software and allow human experts and automated binary analysis tools focus their efforts to adequately protect the software from the appropriate classes of attacks. Previous works in the related area of program provenance use complex analysis over the binary code to determine the compiler and flags used to generate the binary, but do not attempt to identify the source language of the binary. In this work we develop a simple approach that only uses the *strings* exposed by the binary to reliably determine the source language without requiring analysis of the underlying binary code, even when the binary is *stripped*. Our technique employs different machine-learning based classifiers over this simple program meta-data to accurately determine the source language over a large real-world benchmark set and 6 programming languages. We find that our simple approach can achieve an accuracy of over 98% to determine the source language over a large real-world benchmark set in all stripped and unstripped binary configurations.

Key words : Binary, classification, source language.

1 Introduction

Different high-level programming languages have differing properties, strengths and weaknesses with regards to security, robustness, resilience, and performance of the resulting binary software. Binaries from different source languages are susceptible to distinct classes of security vulnerabilities and exploits. Therefore, knowledge of the source language can provide much useful information to the customers and end-users of binary software. This knowledge can also inform security experts and automated binary analysis tools on how to best audit, protect and harden the binary against potential security relevant bugs and attacks.

Yet, to our knowledge, there is no current approach that attempts to determine the source language of the binary. Related techniques have been developed that can identify the compiler suite used to build the binary program. These techniques may use customized signature files [16] or employ complex analysis over the binary code [27] to determine the source compiler. Techniques employing signatures need a specific code pattern for each compiler. Many compiler frameworks, including GCC [15] and LLVM [29], provide frontends for multiple source languages and it is unclear if techniques to identify the source compiler can be directly used to also determine the source language.

In this work we explore this important issue of identifying the source language for software binaries. We found that a simple approach that only uses easily extracted binary meta-data and does not require complex analysis of the binary code to search for patterns is sufficient in most cases to identify the source language.

Many programs have external dependencies on library functions that are resolved at run-time. The binary needs to hold this information for the runtime system. This information along with other language and compiler specific meta-data is encoded as *strings* in the binary, and is present even when the binary is *stripped* of all symbol information. Simple OS tools can extract this information from the binary. Our technique uses these extracted string identifiers (using the Linux 'strings' tool) and encodes the relevant information as a vector. This vector is then fed into machine-learning (ML) based classification algorithms.

We use three ML classification algorithms: Multinomial Naive Bayes, Random Forest Classifier, and the Bernoulli Classifier for this work. Our experiments use hundreds of open-source binaries generated from six source languages and multiple compilers. We experimented with binaries generated with embedded symbol information and those that were stripped of all (unneeded) symbol and metadata information. We found that the best ML models can identify the source language of the binary with an accuracy of over 98% in most cases.

Thus, our objective in this work is to explore and develop techniques to identify the source language of any given binary software. We find that simple ML based techniques that only study the binary meta-data, and do not analyze the binary code, are sufficient in most cases to identify the source language, even for stripped binaries. We make the following contributions in this work.

1. To our knowledge, this is the first work that attempts to develop generalized techniques to reliably recover the source language information from binaries.
2. We develop several machine-learning based models to resolve this challenge and evaluate and compare their performance.
3. We compile a comprehensive benchmark set comprising hundreds of open-source programs over six source languages (C, C++, Fortran, Swift, Rust and Go) and multiple compilers for this work.

The rest of this paper is organized as follows. We present background on the security properties of our selected languages in Section 2. We describe our approach using ML based classification models to determine source language information from binaries in Section 3. We describe our experimental setup, benchmarks, experiments and results in Section 4. We present related work in Section 5. Finally conclusions and future work are presented in Section 6.

2 Background

Our primary goal in this work is to develop techniques to identify the source language of binary software. Several academic [3] and industry [34,36,22] studies report that binaries generated from different programming languages expose the end-users to different kinds and levels of security threats. For instance, one study found that programs written in the 'C' language account for the highest fraction of open-source security vulnerabilities [36]. Another study reports that while CWE-400 [1], CWE-125 [2], and CWE-20 [3] are the most common errors in C programs, CWE-119 [4] dominates C++ program errors, while CWE-400 is most common in Go programs. We also know that while buffer overflow errors (CWE-119) are common in C/C++ programs [36], these errors are not possible in memory-safe languages, like Rust and Go. Thus, knowledge of the source language can prove immensely useful for manual and automated efforts to develop the most effective strategy to secure binary software.

For this work we experiment with six popular *compiled* programming language binaries according to the TIOBE language index [5] that also have a large open-source code base. The compiled languages we selected (along with their most recent TIOBE index) are: C (ranked #2), C++ (ranked #4), Go (ranked #13), Swift (ranked #14), Rust (ranked #28), and Fortran (ranked #31). All these six are compiled languages that generate native binary executables. In the remainder of this section we describe important security relevant properties about these six languages used in this work.

C : 'C' is a low level programming language popular in computer and embedded systems applications. It can achieve high performance and is widely supported. However, memory unsafe languages, like C and C++, score poorly for security and safety [13]. These languages do not natively guaranty memory or type safety. Memory management is the responsibility of the programmer [23]. It does not support *exceptions* so programs may unintentionally ignore critical errors [37] (chapter 11 Error Handling). Secure libraries have been recently made available that can, for instance, replace vulnerable functions like `strcpy` with more secure variants, like `strncpy` and `strcpy_s`.

C++ : C++ is an object oriented extension to the 'C' language that supports encapsulation, abstraction, inheritance and polymorphism. C++ suffers from many of the same security properties and challenges as C. C++ too does not impose memory safety and there is no garbage collector in typical implementations [11]. C++ is also not type safe, and the misuse of casts and unions can lead to type and memory violations. Safer APIs are also available for use in C++ programs.

Fortran : FORTRAN is popular in the domain of high-performance scientific and numeric computing. Buffer overflows are relatively harder to execute as

[1] https://cwe.mitre.org/data/definitions/400.html
[2] https://cwe.mitre.org/data/definitions/125.html
[3] https://cwe.mitre.org/data/definitions/20.html
[4] https://cwe.mitre.org/data/definitions/119.html
[5] https://www.tiobe.com/tiobe-index/

Source Language	Memory Safety	Type Safety	Thread Safety	Availability safe APIs	Overall Security
C	X	X	Y	Y	Low
C++	X	X	Y	Y	Low
Fortran	Some	X	Y	Y	Medium
Swift	Some	Some	Y	Y	Medium
Go	Y	Y	Y	Y	High
Rust	Y	Y	Y	Y	High

Table 1: Summary and comparison of the safety properties of the programming languages employed in this work

strings have defined length and arrays statically specify upper and lower bounds that a compiler can check for errors. However, Fortran cannot resolve invalid memory access at run-time. There is greater type safety in Fortran, and is typically considered safer as compared to C/C++ [19].

Swift : Swift is a general-purpose, powerful and popular compiled language developed by Apple Inc. as a replacement to 'Objective-C'. When 'pure' Swift is used, overflows and underflows cause a runtime trap. But interaction of Swift code with C-style pointers is permitted, and could be unsafe. Swift provides more secure library functions. Memory management is automatic. While there is no garbage collection, ARC (or Automatic Reference Counting) is used to manage memory. Swift is generally safe, but the possibility of interaction with C code can introduce hazards. Swift also suffers from insecure cryptography.

Go : Go is a compiled programming language designed at Google. This is a fairly new language that is gaining popularity in recent years. 'Go' is memory and type-safe. Bounds checking is automatic and pointer arithmetic is not permitted. Memory safety is ensured by a built-in garbage collector. However, Go programs can have data races that could be exploited to cause memory corruption. Safe API functions are available. The Go language aims to achieve both program safety and high performance.

Rust : Rust is a relatively new, open-source, statically typed programming language designed for performance and safety. Both type safety and memory management are supported by the language. Rust can guarantee memory safety by using a borrow checker to validate references. Rust is also a type safe language. It enforces thread safety of all code and data. Rust does not use *null*, so accidental null dereference errors are avoided. Safe APIs are available for programming in Rust. A Rust program is safe against data races, unlike Go.

Table 1 summarizes the safety properties of the six chosen programming languages. Among the programming languages chosen for this study, we categorize C and C++ as low-security languages, Fortran and Swift as moderately secure, and the newer Go and Rust as highly secure languages.

3 Source language classification model

In this section we explain our machine-learning (ML) based approach for determining the source programming language for a given binary executable. We

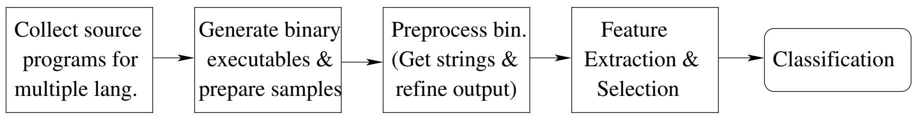

Collect source programs for multiple lang.
Generate binary executables & prepare samples
Preprocess bin. (Get strings & refine output)
Feature Extraction & Selection
Classification

Fig. 1: Phases in our algorithm to identify the source language for binaries

can organize our approach in the following steps or phases: (a) Collect a large dataset of binary executables for each chosen source language, (b) Extract meta-data information from the binary using the available *strings* information, select and extract relevant features, and (c) Develop and explore machine-learning based classification algorithms for experimentation. These steps are illustrated in Figure 1 and explained in more detail in the remainder of this section.

3.1 Gathering Binary Executables

We collect a large and diverse benchmark set for experimentation. We use the open-source project repository, Github to find representative programs for each of our six selected programming languages. We compile a comprehensive set of 120 benchmark programs for each of these 6 languages for experimentation.

All the open-source projects used in this work were distributed as source-code files, with scripts to build them into binary executables included in many cases. The choice of the compilation framework can not only affect the generated binary code, but also the binary meta-data. To study the effect of the compilation framework on the ability of ML models to correctly identify the source language, we employ multiple compilers for our C, C++ and Fortran language programs. For C and C++ we use three compiler frameworks: the GNU Compiler, Clang and ICC (Intel C++ Compiler) compilation frameworks. We use two compilers for building Fortran binaries: GNU Fortran and Intel Fortran compilers.

The older C, C++, and Fortran programming languages have a highly diverse compiler ecosystem. In contrast, a single reference compiler is usually used to build programs written in the three newer programming languages in our set, Swift, Go and Rust. We use the reference 'Swift' [30] compiler, the default 'Go' [9,14] compiler, and the standard 'rustc' [28] compiler for building the Swift, Go and Rust programs, respectively. In the future, we will also experiment with alternative compiler frameworks for these languages.

Thus, in this phase, we gathered all the benchmark programs, generated or updated the build scripts, installed the compiler frameworks, and then built the executables for each configuration. The binaries were built with compiler optimizations switched on and the binaries were optionally stripped of all symbol information. All our programs were built for the x86-64 instruction set architecture running the Ubuntu Linux version 20.04 LTS.

3.2 Extract Meta-Data Information

We use the meta-data information extracted by the Unix *strings* tools from each (stripped) binary program. The *strings* tool outputs all printable character sequences in the binary that are at least 4 characters long (and are followed by an unprintable character). We process the raw *strings* output to remove strings that are not directly useful for our task.

In this processing step, we manually identify 'string' sequences that seem useful to identify the source language. For instance, strings describing the compiler, libraries (like libstdc++.so, GLIBC, etc.), number of libraries were deemed useful. We use a simple regular expression combined with string manipulation to process the raw strings data. This extracted information is then represented in a vector form to input to our machine learning models.

3.3 Machine-learning Based Classification

There are multiple machine-learning based classification models that could be used for this work [7]. We arbitrarily choose the following 3 popular classification algorithms for this study. The first, called Multinomial Naive Bayes classifier, is suitable for classification with discrete features of strings. Since our feature set includes the frequency of certain strings that appear in the binary, this classification scheme seems useful. The second model we use is called the Random Forest classifier. This model uses several decision tree classifiers on various subsamples of the datasets and improves its accuracy by using averaging and control over-fitting. The third classification model we use is the popular Bernoulli model.

Each of these ML models need to be *trained* first to build the classifier *models*, which can then be tested for accuracy in detecting the source language of the binary. For our experiments in this work, we select 20% of our feature sets as test cases, and use the remaining 80% of the sets to train the model.

4 Experimental Results

In this section we present results of our experiments that use the trained ML models to identify the source language of binary executables. We configure three different datasets. The first dataset uses binaries that are unstripped. Unstripped binaries may embed more useful 'strings' information. Our second dataset uses only the stripped binaries. Finally, a more diverse third dataset combines binaries from both the stripped and unstripped categories in equal proportions.

As mentioned earlier, multiple compilers are used to build binaries for the C, C++ and Fortran programs. In these cases, we include an equal number of programs from each compiler in our dataset for that source language. For instance, the dataset for the C source language contains an equal number (one-third) of binaries built by the GNU GCC, Intel ICC and Clang compiler frameworks. Each compiler may use different libraries and embed different metadata strings in the binary. A more diverse dataset should improve the robustness of the machine learning models when encountering the test binaries.

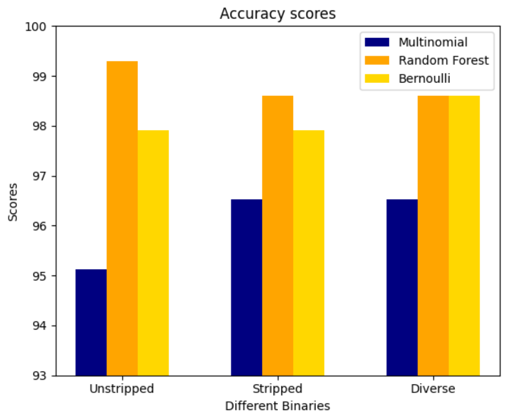


Fig. 2: Accuracy of our ML models on different datasets

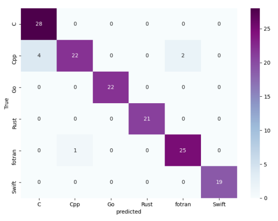
Figure 2 shows the overall results of our experiments with the 3 selected classification models. Thus, we find that the best ML models achieve an accuracy of up to 99.3% when identifying the source language using the unstripped dataset. The best models achieve an overall accuracy of 98.61% when using the stripped and diverse datasets.

The results show that the Multinomial Naive Bayes algorithm achieves the poorest accuracy among all our models. While the other two ML models perform better, the Random Forest classifier achieves accuracy that is marginally higher than the Bernoulli classifier in the *stripped* and *unstripped* configurations.

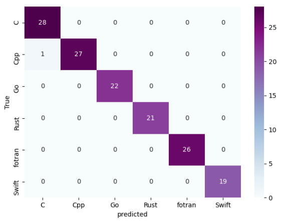
Figures 3, 4, and 5 show further details about cases when the ML models fail to detect the correct source language for a binary in the *unstripped*, *stripped* and *diverse* experimental configurations, respectively. These figures employ a matrix representation, where the the X- and Y-axis refer to the programming languages C, C++, Go, Rust, Fortran and Swift, respectively.

The numerals along the diagonal show the number of programs that are correctly identified, in each case. Thus, Figure 3(a) shows that there are 28 C binaries, 22 C++ binaries, 22 Go binaries, 21 Rust binaries, 25 Fortran binaries, and 19 Swift binaries that are correctly identified by the Multinomial Naive Bayes algorithm in the *unstripped* benchmark configuration. Likewise, there are 4 C++ binaries that are misclassified as C binaries, and 2 C++ binaries that are misclassified as Fortran binaries in this same configuration.

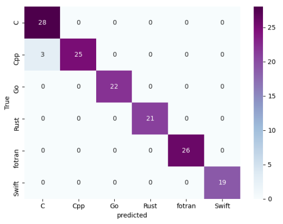
Overall, we find that most of the misclassifications occur due to C++ binaries identified as C programs. Also, as expected, the *unstripped* and *diverse* datasets achieve accurate classification due to the availability of more useful meta-data information. However, it was surprising to find that the models achieve high accuracy even in the *stripped* configuration. Thus, our findings indicate that distinctive features are retained in the binary meta-data even after it is stripped of all symbol information (that is not needed at run-time), which allows these ML models to accurately detect the source language even for stripped binaries.



(a) Multinomial

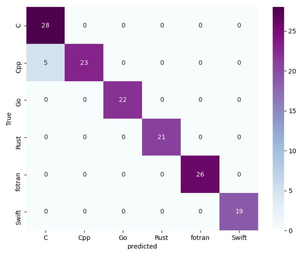


(b) Random Forest

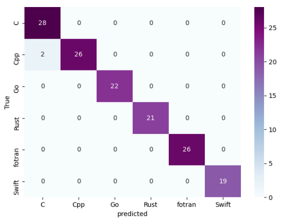


(c) Bernoulli

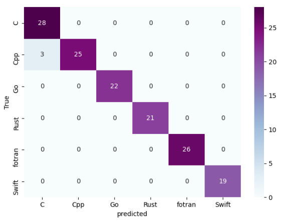
Fig. 3: In an Unstrippped dataset



(a) Multinomial

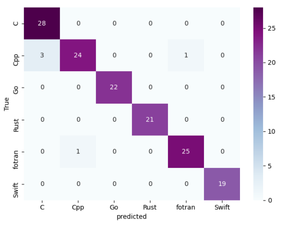


(b) Random Forest

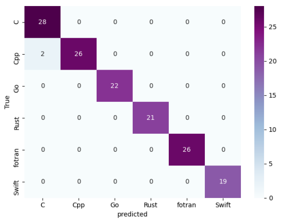


(c) Bernoulli

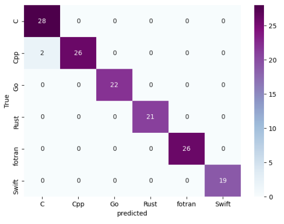
Fig. 4: In a strippped dataset



(a) Multinomial



(b) Random Forest



(c) Bernoulli

Fig. 5: In a Diverse dataset

5 Related Works

In this section we discuss prominent research in areas that are related to this work. While there is limited research to identify the source programming language from a given binary, there is much prior research work in the related areas of compiler provenance and code authorship identification.

Research in program or compiler provenance attempts to identify the compiler tool-chain and/or optimization levels used to generate a binary program. These methods take a signature-matching or a machine-learning based approach. Signature matching methods search the binary against a set of code signatures manually generated by experts to identify the compiler used. PEiD [4] and IDA Pro [17] are examples of tools that use this signature-based method. This method relies on expert knowledge and can be easily confused by small differences in signatures. The other strategy employs ML based models to capture compiler-specific patterns from the binary code [27,6,24,33,1]. These patterns include instruction sequences or idioms [27], instruction type and frequency [5], and subgraphs from the program's CFG [25]. All these approaches require looking and deciphering properties about the binary code. Our work has a different goal and does not rely on parsing the binary code.

The topic of identifying the author of a given binary software is also related to our work [20,26,21,1,2]. This problem, called code authorship attribution, aims to categorize the authors of malware. This categorization along with information about the tools and techniques used by a set of malware that is written by the same author can help determine how the malware can spread and evolve and help in cyber-crime investigations. Most of these approaches also employ ML based methods to model various aspects of the binary code and control-flow to accurately predict the binary code authors or if different software are written by the same author.

Machine-learning based approaches are also popular is other binary analysis problems, including the detection of a virus in binary software [31], code clone detection to find the same source code in binaries compiled for different architectures [39], detection of malicious software [8,10,18], function recognition [35] and similarity [12] in binaries, and many others [38]. We too use a machine-learning based approach for this work.

Finally, Tian et al. proposed a classification system of identifying trojan and virus families based on printable strings [32]. We too employ the *strings* available in program binaries in this work to identify the source language of binaries.

6 Conclusions and Future Work

In this paper we explore the issue of identifying the source programming language for any given binary software. We build an experimental framework that we populate with binaries from 6 different source languages and compiled using multiple different compilers. We found that our simple machine-learning based approach that only considers the *strings* exposed by the binary and does not analyze the binary code is sufficient to accurately identify the source language, even for binaries that are optimized and stripped of all symbol information.

In the future, we will study if this approach continues to work with more programming languages. We will also experiment with binaries that may contain complex code that is obfuscated, hand-written and constructed using multiple different languages. Finally, we will explore the performance and accuracy of other ML models, including ensemble ML to further improve results.

References

1. Alrabaee, S., Debbabi, M., Shirani, P., Wang, L., Youssef, A., Rahimian, A., Nouh, L., Mouheb, D., Huang, H., Hanna, A.: Compiler Provenance Attribution. Springer (2020)
2. Alrabaee, S., Wang, L., Debbabi, M.: Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs). Digital Investigation **18**, S11–S22 (2016)
3. Antal, G., Mosolygó, B., Vándor, N., Hegedundefineds, P.: A data-mining based study of security vulnerability types and their mitigation in different languages. In: Computational Science and Its Applications – ICCSA 2020: 20th International Conference, Cagliari, Italy, July 1–4, 2020, Proceedings, Part IV. p. 1019–1034. Springer-Verlag (2020)
4. Arghire, G.: Detect packers, cryptors and compilers bundled withpe executables with the help of this reliable piece of software that boasts a high detection rate. https://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/PEiD-updated.shtml (2017)
5. Austin, T.H., Filiol, E., Josse, S., Stamp, M.: Exploring hidden markov models for virus analysis: A semantic approach. In: 2013 46th Hawaii International Conference on System Sciences. pp. 5039–5048 (2013)
6. Benoit, T., Marion, J.Y., Bardin, S.: Binary level toolchain provenance identification with graph neural networks. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 131–141 (2021)
7. Bonaccorso, G.: Machine Learning Algorithms: A Reference Guide to Popular Algorithms for Data Science and Machine Learning. Packt Publishing (2017)
8. Coogan, K., Debray, S., Kaochar, T., Townsend, G.: Automatic static unpacking of malware binaries. In: 2009 16th Working Conference on Reverse Engineering. pp. 167–176. IEEE (2009)
9. Cox, R., Griesemer, R., Pike, R., Taylor, I.L., Thompson, K.: The go programming language and environment. Communications of the ACM **65**(5), 70–78 (April 2022)
10. Dai, J., Guha, R.K., Lee, J.: Efficient virus detection using dynamic instruction sequences. Journal of Computing **4**(5), 405–414 (2009)
11. Edelson, D.R.: A mark-and-sweep collector c++. In: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 51–58. ACM (1992)
12. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E.: discovre: Efficient cross-architecture identification of bugs in binary code. In: NDSS (2016)
13. Gaynow, A.: What science can tell us about c and c++'s security. https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/ (May 27, 2020)
14. Go Team: "Download and Install Go". https://go.dev/doc/install (2022)
15. Griffith, A.: GCC: The Complete Reference. McGraw-Hill, Inc., USA, 1 edn. (2002)
16. Hex-rays: Ida f.l.i.r.t. technology: In-depth. https://hex-rays.com/products/ida/tech/flirt/in_depth/ (2022)
17. IDA Pro: Hex-rays ida pro disassembler and debugger. https://hex-rays.com/ (2017)
18. Islam, R., Tian, R., Batten, L., Versteeg, S.: Classification of malware based on string and function feature selection. In: 2010 Second Cybercrime and Trustworthy Computing Workshop. pp. 9–17 (2010)

19. ISO/IEC JTC 1/SC22: Fortran language vulnerabilities. http://www.open-std.org/jtc1/sc22/wg23/docs/documents (2022)
20. Kalgutkar, V., Kaur, R., Gonzalez, H., Stakhanova, N., Matyukhina, A.: Code authorship attribution: Methods and challenges. ACM Computing Surv. **52**(1) (feb 2019)
21. Layton, R., Azab, A.: Authorship analysis of the zeus botnet source code. In: 2014 Fifth Cybercrime and Trustworthy Computing Conference. pp. 38–43. IEEE (2014)
22. Lupsan, H.: Security vulnerability classes in popular programming languages. https://versprite.com/blog/security-research/security-vulnerability-classes-in-popular-programming-languages/ (2021)
23. Myers, N.C.: Memory Management in C++, p. 363–374. SIGS Publications, Inc., USA (1996)
24. Pizzolotto, D., Inoue, K.: Identifying compiler and optimization level in binary code from multiple architectures. IEEE Access **9**, 163461–163475 (2021)
25. Rosenblum, N., Miller, B.P., Zhu, X.: Recovering the toolchain provenance of binary code. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. p. 100–110. ACM (2011)
26. Rosenblum, N., Zhu, X., Miller, B.P.: Who wrote this code? identifying the authors of program binaries. In: European Symposium on Research in Computer Security. pp. 172–189. Springer (2011)
27. Rosenblum, N.E., Miller, B.P., Zhu, X.: Extracting compiler provenance from program binaries. In: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. p. 21–28. ACM (2010)
28. Rust Team: Guide to rustc development. https://www.swift.org/swift-compiler/ (2022)
29. Sarda, S., Pandey, M.: LLVM Essentials. Packt Publishing (2015)
30. Swift Team: Swift compiler. https://www.swift.org/swift-compiler/ (2022)
31. Tesauro, G.J., Kephart, J.O., Sorkin, G.B.: Neural networks for computer virus recognition. IEEE expert **11**(4), 5–6 (1996)
32. Tian, R., Batten, L., Islam, M.R., Versteeg, S.: An automated classification system based on the strings of trojan and virus families. pp. 23 – 30 (11 2009)
33. Tian, Z., Huang, Y., Xie, B., Chen, Y., Chen, L., Wu, D.: Fine-grained compiler identification with sequence-oriented neural modeling. IEEE Access **9**, 49160–49175 (2021)
34. Tung, L.: Programming language security: These are the worst bugs for each top language. https://www.zdnet.com/article/programming-language-security-these-are-the-worst-bugs-for-each-top-language/ (2020)
35. Wang, S., Wang, P., Wu, D.: Semantics-aware machine learning for function recognition in binary code. 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME) (2017)
36. WhiteSource: What are the most secure programming languages? https://www.whitesourcesoftware.com/most-secure-programming-languages/ (2021)
37. Wikibooks: C programming. https://en.wikibooks.org/wiki/C_Programming (2017)
38. Xue, H., Sun, S., Venkataramani, G., Lan, T.: Machine learning-based analysis of program binaries: A comprehensive study. IEEE Access **7**, 65889–65912 (2019)
39. Zeng, J., Fu, Y., Miller, K.A., Lin, Z., Zhang, X., Xu, D.: Obfuscation resilient binary code reuse through trace-oriented programming. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 487–498 (2013)