Big Code: New Opportunities for Improving Software Construction

Francisco Ortin[1*], Javier Escalada[1], Oscar Rodriguez-Prieto[1]
[1] University of Oviedo, Computer Science Department, c/Calvo Sotelo s/n, 33007, Oviedo, Spain

* Corresponding author. Tel.: +34 985 10 3172; email: francisco.ortin@gmail.com
Manuscript submitted June 30, 2016; accepted August 3, 2016.
doi: 10.17706/jsw.11.11.1083-1088

Abstract: An emerging research topic called big code has recently appeared. Big code is based on the idea that open source code repositories can be used to create new kind of programming tools and services to improve software reliability and construction. We discuss different fields of application of big code, and the key issues to implement tools aimed at improving software construction following this approach. We describe the existing works that have already used this idea to build tools for vulnerability detection, software deobfuscation, automatic code completion for API usage, and efficient querying using detailed source-code information. Then, we propose different fields of application and the key issues found. We identify eight different fields where big code may be applied, and describe different examples for each field. We also detect seven different issues that must be tackled when creating tools based on the big code approach.

Key words: Big code, graph database, machine learning, probabilistic reasoning, program analysis

1. Introduction

In the last years, there has been an increasing interest in big data, namely processing and analyzing large datasets. The objective of big data is to extract value from large datasets, such as the creation of predictive models, generation of reports, data visualization, finding relationships between variables, and object grouping (clustering). Big data is being used in many different fields such as medicine, finance, healthcare, education, social networks and genomics.

At the same time, the use of open source code repositories (GitHub, SourceForge, BitBucket and CodePlex) has significantly risen in the last decade. One example of this trend is the publication of the 10 millionth GitHub repository. The first million repositories were created just under 4 years. The 10 millionth just took 48 days [1]. These massive codebases imply a unique opportunity for creating new kind of programming tools and services to improve software reliability and construction, using machine learning and probabilistic reasoning. Due to its similarity with big data, this research field has been termed "big code" [2].

The main contributions of this paper are the description of the emerging topic of big code, and the new opportunities it creates for improving software construction. To this end, we identify the existing work in this field (Section 2), the expected fields of application of big code (Section 3), and different issues to be considered in the usage of massive codebases (Section 4).

2. Existing Works

The Software Reliability Lab in the Department of Computer Science at ETH Zurich has produced several notable results in big code using statistical reasoning. They created JSNice, a kind of deobfuscator that predicts names of variables and built-in type annotations of JavaScript programs, using conditional random fields (CRFs) [3]. They trained the CRF model with 10,517 projects from GitHub, predicting the correct names for 63.4% of program identifiers and 81.6% for type annotations. Slang is another tool to perform code completion for programs using APIs, reducing the problem of code completion to a natural-language processing problem of predicting probabilities of sentences. They used both n-gram and recurrent neural network models, obtaining the desired completion in the top 3 candidates in 90% of the cases. They also investigated statistical machine translation from C# to Java, achieving 41% of the translated methods to be semantically equivalent to the original ones [4].

The Machine Learning and Computer Security Research (MLSEC) at the University of Göttingen has also developed different tools that process program representations to improve software security. In particular, the Joern tool represents programs as code property graphs, a combination of abstract syntax trees (AST), control flow graphs (CFG) and program dependency graphs (PDG), and stores them in a graph database [5]. That representation allows modeling templates of common vulnerabilities with graph traversals to identify buffer and integer overflows, format string vulnerabilities, and memory disclosures. With that infrastructure, they managed to identify 18 previously unknown vulnerabilities in the source code of the Linux kernel.

The MLSEC has also used supervised learning algorithms for discovering vulnerabilities in source code [6]. They take the ASTs of code with known vulnerabilities to extrapolate them by determining structural patterns. Then, functions potentially suffering from the same flaw can be suggested to the analyst. They applied this method to discover 10 zero-day vulnerabilities in 4 open-source projects [6]. Chucky is another tool that exposes missing checks in vulnerable code, learning from the analysis of programs that do not have those vulnerabilities [7]. Chucky found 12 previously unknown vulnerabilities in 2 popular software projects.

Regarding the representation of programs, Urma and Mycroft argue that graph data models and their associated query languages provide a unifying conceptual model, and an efficient scalable implementation to store full source-code detail [8]. Based on this idea, Wiggle is a tool that stores overlapping graphs as a mixture of AST, type information, CFG and data-flow levels. The experiments showed that Wiggle scales to multi-million-line programs, and it is more expressive than source-code query languages [8].

3. Potential Fields of Application

We now discuss some fields of application of big code that we believe might be active topics of research in the following years. We classify them regarding the main techniques to be used, and describe for each classification some example problems worth solving.

– *Statistical modelling methods*. Statistical methods such as conditional random fields can be used to extract useful information from existing applications, and provide statistically likely solutions to problems that are difficult to solve with traditional rule based techniques [3]. These methods can be used to provide code completion after analyzing recurrent idioms, perform automatic type annotation of gradually typed languages [3], obtain semantic information of programs that use contracts (Eiffel, Code Contracts and D) and improve deobfuscation –not only for variable names.

– *Machine learning for the extrapolation of known patterns*. Yamaguchi *et al.* [6] showed how ASTs of programs showing a similar pattern can be embedded in a vector space to identify those patterns, and then extrapolate them to a code base. This technique can be used, apart from vulnerability detection, for predicting errors not detected by compilers, such as lack of exception handling in languages without the `throws` descriptor, `null` reference exceptions, and race conditions. In general, we think this technique may be used to extrapolate patterns of undesirable code.

– *Statistical analysis of co-occurrence and anomaly detection*. By analyzing high-quality code, a probabilistic model can be built by computing the probability of co-occurrence of different elements in the same source code pattern (e.g., in Java overriding `hashCode` also commonly requires overriding `equals` [9]). Similarly, it can also be used for anomaly detection compared to the codebase of high-quality programs (e.g., using a Java `AutoCloseable` object with neither a try-with-resources statement nor a `finally` block). This technique was used by MLSEC to determine missing checks in vulnerable source code [7], but it can be generalized to many different scenarios.

– *Natural language processing* (NLP). NLP techniques such as n-gram and recurrent neural network models were used to synthesize sequences of calls to some APIs, together with their arguments [10]. The combination of NLP and statistical reasoning may be used for other tasks such as the automatic creation of program test cases, default implementation of methods and functions, automatic classification of programs behaviors by using latent semantic analysis [6], and sophisticated code completion from program structures and source comments.
– *Supervised learning*. The program representation may be extended with other labeled information for a set of training codebases, to later predict that labeled information. The inferred function may be valuable to improve the software construction process. One example is improving the decompilation process, learning from recurrent patterns that specific compilers use to generate binary code [11]. Another example is predicting the performance benefit of annotating the type of a variable in a gradually typed language, using regression techniques [12].
– *Knowledge-based systems*. Both Joern and Wiggle (Section 2) represent programs as different kinds of graphs in a graph database, and later perform queries using a domain-specific language (DSL) provided by that database. They used this approach to detect vulnerabilities, covariant array assignment, and the use of Java generic wildcards [8]. The valuable information extracted by performing static program analysis (e.g., methods overriding another method, methods calling another method, classes implementing one interface, and types using another type) can be extended by a rule-based knowledge system, which in turn uses a DSL to gather that valuable information stored in the graph database. With this approach, experts may create systems for numerous purposes, such detecting non-refactored code, measuring software metrics, ensuring high-level guidelines like the well-known Oracle and Carnegie Mellon Java guidelines [13], detecting typical semantic errors not checked by the compiler [9], and identifying typical performance problems.
– *Web-based collaborative systems*. The previous rule-based knowledge system could be provided as a website as JSNice [3], so that programmers may check the limitations of their code online. They could also include specific knowledge as experts to be used in their own programs. In a collaborative environment, these rules may even be promoted to the global system. The system may also be used as a service provider, following the programming copilot metaphor proposed by Kite [14]. Besides, the programs used in the collaborative system could also be incorporated in the system codebase.
– *Unsupervised learning*. It can be analyzed how different programming elements such as methods, classes, packages and interfaces are classified, considering all the different features that can be extracted from its representation. The groups created can be used for many different purposes: discovering unknown software patterns, finding relationships between program elements, helping to reform functionality that has become dispersed because of software evolution, identifying software with related functionality, recommending source code patterns based on the program structures in the same group, and program anomaly detection.

4. Issues to be Considered

To achieve the identified fields of application of big code, the following issues need to be addressed.

1) *Static program analysis*. To gather detailed information about programs, static program analysis must be performed. To this end, the information gathering process may be included as a plugin of an existing compiler, extracting the analysis information. This process commonly requires considerable computing resources.
2) *Program representation*. Programs can be represented with multiple different structures such as ASTs, CFGs, PDGs, type graphs, and data-flow graphs. Besides, these graphs also overlap, since some nodes may belong to different structures at the same time. A structured representation of programs representing this information must be carefully defined, since it will be the knowledge base of the system.
3) *Efficient storage*. The program representation described above should be stored in a database to perform efficient queries. Both Wiggle and Joern use the Neo4j graph database. In Wiggle, storing 12 Java well-known projects took between 2 and 10 times longer than compiling it with javac.
4) *Knowledge representation*. The knowledge representation is based on queries against the program representation. Therefore, a powerful and efficient query language must be used to retrieve this knowledge. Besides, that language should also support knowledge inference and graph transformations. Wiggle and Joern use, respectively, the Cypher and Gremlin query languages.
5) *Embedding in vector space*. The graph structures identified in the second point of this enumeration represent a rich source of information. However, most of the machine learning algorithms cannot be applied directly to this type of data, since they commonly use numerical vectors. A key issue is how these graph structures are transformed into vectors, since they need to capture the structure of the programs. The MLSEC group has experimented with different transformations for generalizing vulnerability extrapolation [6].
6) *Algorithm selection and runtime performance*. Another issue to tackle is the algorithm to be used for each problem. Besides the problem of selecting the appropriate algorithm, we also have to deal with the computation requirements of each algorithm. Similar to big data, the information gathered by statically analyzing massive codebases can produce impressive large datasets for training predictive models.
7) *Identifying high-quality software*. In the previous section, we propose different fields of application where the learning algorithm should be trained with high-quality software. A difficult task to undertake is identifying those programs with the expected level of quality.

5. Conclusions

We have analyzed how the emerging big code research field can create new opportunities for building programming tools and services to improve software reliability and construction. We think this research line will be active in the following years, and we identify different fields of applications based on the works that have recently emerged.

We are currently working on improving decompilation using machine learning [11], and predicting the performance gain of adding type annotations to variables in a gradually typed program [12]. We also plan to build a web-based collaborative system to support a rule-based expert system based on the knowledge extracted from static program analysis, stored in graph databases.

Acknowledgment

This work has been funded by the European Union, through the European Regional Development Funds (ERDF); and the Principality of Asturias, through its Science, Technology and Innovation Plan (grant GRUPIN14-100).

References

[1] Doll, B. (2013). 10 million repositories. *GitHub*. Retrieved August 31, 2016, from https://github.com/blog/1724-10-million-repositories.

[2] Defense Advanced Research Projects Agency. (2014). MUSE envisions mining "big code" to improve software reliability and construction. Retrieved August 31, 2016, from http://www.darpa.mil/news-events/2014-03-06a.

[3] Raychev, V., Vechev, M., & Krause, A. (2015). Predicting program properties from "big code", in: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, ACM, New York, NY, USA, 2015, pp. 111-124.

[4] Karaivanov, S., Raychev, V., & Vechev, M. (2014). Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward*! (pp. 173-184). ACM, New York, NY, USA.

[5] Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP'14* (pp. 590-604). IEEE Computer Society, Washington, DC, USA.

[6] Yamaguchi, F., Lottmann, M., & Rieck, K. (2012). Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC'12* (pp. 359-368). ACM, New York, NY, USA.

[7] Yamaguchi, F., Wressnegger, C., Gascon, H., & Rieck, K. (2013). Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS'13* (pp. 499-510). ACM, New York, NY, USA.

[8] Urma, R., & Mycroft, A. (2015). Source-code queries with graph databases - with application to programming language usage and evolution. Science of Computer Programming, 97, pp. 127-134.

[9] Bloch, J. (2008). *Effective Java (The Java Series)* (2nd Edition). Prentice Hall PTR, Upper Saddle River, NJ, USA.

[10] Raychev, V., Vechev, M., & Yahav, E. (2014). Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14* (pp. 419-428). ACM, New York, NY, USA.

[11] Escalada, J., & Ortin, F. (2014). An Adaptable Infrastructure to Generate Training Datasets for Decompilation Issues. In *New Perspectives in Information Systems and Technologies*, Volume 2. Springer International Publishing, pp. 85-94.

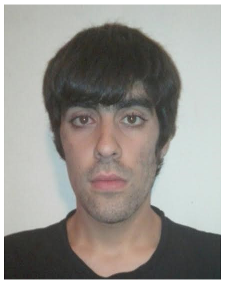
[12] Takikawa, A., Feltey, D., Greenman, B., New, M. S., Vitek, J., & Felleisen, M. (2016). Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'16* (pp. 456-468) ACM, New York, NY, USA.

[13] CERT, Carnegie Mellon University. (2016). Java coding guidelines. Retrieved August 31, 2016, from https://www.securecoding.cert.org/confluence/display/java/Java+Coding+Guidelines.

[14] Kite, Your program copilot. (2016). Retrieved August 31, 2016, from https://kite.com.

Francisco Ortin, 1973, is an Associate Professor of the Computer Science Department at the University of Oviedo, Spain. He is the head of the Computational Reflection research group (http://www.reflection.uniovi.es). He received his BSc in Computer Science in 1994, and his MSc in Computer Engineering in 1996. In 2002 he was awarded his PhD entitled *A Flexible Programming Computational System developed over a Non-Restrictive Reflective Abstract Machine*. He has been the principal investigator of different research projects funded by Microsoft Research and the Spanish Department of Science and Innovation. His main research interests include dynamic languages, type systems, aspect-oriented programming, computational reflection, and runtime adaptable applications. Contact him at http://www.di.uniovi.es/~ortin

Javier Escalada, 1988, full-time PhD student at the Computer Science Department of the University of Oviedo, Spain. He received his BSc degree in Computer Science in 2010. In 2011 he was awarded an MSc in Computer Security from Deusto University, Bilbao. Then, we obtained an MSc in Artificial Intelligence in 2012 from the Polytechnic School of Madrid. His PhD thesis is focused on creating a platform for detecting binary patterns in programs using machine learning. That platform is currently used to improve the information gathered by existing decompilers.



Oscar Rodriguez-Prieto, 1992, full-time PhD student at the Computer Science Department of the University of Oviedo, Spain. He received his BSc degree in Software Engineering in 2014. In 2015 he was awarded an MSc in Artificial University from the Spanish National Distance Education University (UNED). His PhD thesis is aimed at using Big Code to improve software reliability and construction.