Genetic Improvement for Code Obfuscation

Justyna Petke
University College London
London, UK
j.petke@ucl.ac.uk

ABSTRACT

Genetic improvement (GI) is a relatively new area of software engineering and thus the extent of its applicability is yet to be explored. Although a growing interest in GI in recent years started with the work on automatic bug fixing, the area flourished when results on optimisation of non-functional software properties, such as efficiency and energy consumption, were published. Further success of GI in transplanting functionality from one program to another leads to a question: *what other software engineering areas can benefit from the use of genetic improvement techniques?* We propose to utilise GI for code obfuscation.

CCS Concepts

•Software and its engineering → Search-based software engineering;

Keywords

Genetic Improvement; Code Obfuscation; Software Optimisation

1. GENETIC IMPROVEMENT

The name 'genetic improvement' was only coined in 2012 in the keynote paper by Harman et al. [7]. However, beginnings of GI can be traced back to 1995 with the work by Ryan and Walsh [12], where genetic programming was used to transform an existing sequential program into a parallel one. The next important step arose from the bug fixing work by Arcuri et al. [2]. The proposed approach of using genetic programming for software repair has developed as a field in its own right and led to several awards, including a 'Gold Humie' [14]. Langdon et al. [9] and White et al. [15] used a similar method to optimise non-functional properties of software, such as efficiency and energy consumnption. Petke et al. [11] adopted this approach for software specialisation. More recent applications of GI involve reduction of memory consumption [16] and software slimming [17]. Furthermore, research on automated software transplantation using genetic improvement led to a best paper award [4] and wide media coverage [13].

GECCO Workshop on Genetic Improvement '16 Denver, Colorado USA

DOI:

2. CODE OBFUSCATION

Code obfuscation arose from the need for secure software systems. Obfuscation aims to prevent unauthorised persona from tampering with software and hinder its reverse-engineering. However, it can also be used for malicious purposes by software virus creators. Collberg et al. [5] first defined an obfuscator in terms of semantics-preserving program transformations applied in a systematic fashion to the selected code. Collberg et al. [5] also provided three main categories of obfuscation: layout (aimed at making software unreadable), data (affecting data structures, e.g., splitting variables) and control obfuscation (altering program's control flow). Barak et al. [3] define an obfuscator O as an (efficient, probabilistic) "compiler" that takes as input a program (or circuit) P and produces a new program $O(P)$ that has the same functionality as P yet is "unintelligible" in some sense. The field's popularity can be seen in 24 editions of the 'International Obfuscated C Code Contest' so far [1].

3. GI FOR CODE OBFUSCATION

Traditionally, code obfuscators apply a set of program transformations to the whole of the selected code. These program transformations are semantics-preserving. Therefore, we propose to loosen the semantics criterion by introducing test cases as a proxy for correct program behaviour and new operators in the context of code obfuscation. This way code changes that were previously infeasible are allowed to take place and thus the space of possible obfuscated programs can increase drastically. Genetic programming has already been used for the purpose of building obfuscated software [10], showing feasibility of the approach.

In the GI approach we propose to start with an existing program. Standard obfuscating operations can be applied, such as, deletion, renaming, replacement and loop condition changes. However, we propose to use metaheuristic search to apply these changes. Furthermore, we do not impose semantics-preserving conditions on the mutation operators. For example, we allow for *arbitrary* changes to the loop conditions. These could be achieved by replacing one loop condition with another in a different part of the code. Such an operator has already been used in genetic improvement work [9]. We use a given code obfuscation metric in the fitness function (Drape [6], for instance, provides several examples of such metrics). We also use the number of

passed test cases as a proxy for correct program behaviour in fitness evaluation. We can apply the genetic programming approach to select the top programs for mutation and crossover in the next generation. The whole process can then be repeated with fixed population and generation size as is standard in genetic programming [8]. The proposed process is summarised in Figure 1.

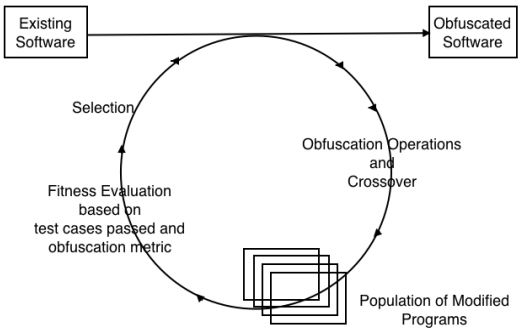

Figure 1: Proposed approach for the use of GI for code obfuscation.

Similarly to code obfuscation, the proposed approach is not limited to changes at the source code level. The same method can be applied to bytecode with appropriate changes to the fitness function and mutation operators. Furthermore, the search technique does not need to be genetic programming (even though it had a lot of success in the GI area). Other search-based methods, such as hill climbing, could be explored. We propose a general new method for code obfuscation. Further research and experiments are needed to establish the best components of the genetic improvement approach in the context of code obfuscation.

4. CONCLUSIONS

We propose to use genetic improvement techniques for the purpose of code obfuscation. By allowing unsystematic, non-semantics-preserving changes to the code, the space of obfuscated programs is significantly increased, thus allowing greater security of the modified code. Further research is needed to verify if code correctness of the resultant program is sufficient when test cases are used as a proxy in the proposed approach. However, this work shows that the space of applications of genetic improvement techniques is yet to be explored.

5. REFERENCES

[1] The International Obfuscated C Code Contest. http://www.ioccc.org/index.html.

[2] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In J. Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, pages 162–168, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, pages 1–18, 2001.

[4] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 257–269, New York, NY, USA, 2015. ACM. ACM SIGSOFT Distinguished Paper Award winner.

[5] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations, 1997.

[6] S. Drape. Intellectual property protection using obfuscation. Technical Report RR-10-02, March 2010.

[7] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs. In *The 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 12)*, pages 1–14, Essen, Germany, Sept. 3-7 2012. ACM.

[8] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[9] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, Feb. 2015.

[10] P. LaRoche, N. Zincir-Heywood, and M. Heywood. Using code bloat to obfuscate evolved network traffic. In C. D. et al., editor, *EvoCOMNET*, volume 6025 of *LNCS*, pages 101–110, Istanbul, 7-9 Apr. 2010. Springer.

[11] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In M. N. et al., editor, *17th European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 137–149, Granada, Spain, 23-25 Apr. 2014. Springer.

[12] C. Ryan and P. Walsh. Automatic conversion of programs from serial to parallel using genetic programming - the paragen system. In *Proceedings of ParCo'95*. North-Holland, 1995.

[13] J. Temperton. Code 'transplant' could revolutionise programming. *Wired.co.uk*, 30 July 2015. Online.

[14] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 364–374, 2009. 'Gold Humie' 2009 award winner.

[15] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, Aug. 2011.

[16] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke. Deep parameter optimisation. In S. S. et al., editor, *GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, pages 1375–1382, Madrid, 11-15 July 2015. ACM.

[17] K. Yeboah-Antwi and B. Baudry. Embedding adaptivity in software systems using the ECSELR framework. In W. B. L. et al., editor, *Genetic Improvement 2015 Workshop*, pages 839–844, Madrid, 11-15 July 2015. ACM.