2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

Towards Better Program Obfuscation: Optimization via Language Models

Han Liu
Institute of Software System and Engineering
Tsinghua University
Beijing, China
liuhan12@mails.tsinghua.edu.cn

ABSTRACT

As a common practice in software development, program obfuscation aims at deterring reverse engineering and malicious attacks on released source or binary code. Owning ample obfuscation techniques, we have relatively little knowledge on how to most effectively use them. The biggest challenge lies in identifying the most useful combination of these techniques. We propose a unified framework to automatically generate and optimize obfuscation based on an obscurity language model and a Monte Carlo Markov Chain (MCMC) based search algorithm. We further instantiate it for JavaScript programs and developed the **Closure*** tool. Compared to the well-known Google Closure Compiler, **Closure*** outperforms its default setting by 26%. For programs which have already been well obfuscated, **Closure*** can still outperform by 22%.

CCS Concepts

•**Security and privacy** → *Software security engineering;*

Keywords

obfuscation; obscurity language model; MCMC random search

1. PROBLEM AND MOTIVATION

Program obfuscation is a common practice in software development, making source or executable code difficult for human to understand. In practice, obfuscation serves multiple purposes such as deterring reverse engineering and preventing malicious attacks. However, even with obfuscation, software is still vulnerable to various unexpected attacks. To combat potential adversaries, a large group of obfuscation techniques have been developed [8, 10, 21]. Generally, these techniques enforce specific transformations to complicate both the syntactic and semantic structure of programs. For human adversaries who attempt to *manually* crack the obscurity, by either reading the code or using static analyzers, such obfuscation is effective. Yet, facing the new and promising learning-based adversaries [3, 18], it is insufficient. Based on their evaluation, these adversaries show great potential in eliminating obfuscation.

ICSE '16 Companion, May 14 - 22, 2016, Austin, TX,USA
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4205-6/16/05.
DOI: http://dx.doi.org/10.1145/2889160.2891040

The emerging threat motivates us to revisit program obfuscation. Although given a large number of techniques, we have relatively little knowledge about how to coordinate and maximize their power to improve obfuscation, especially *w.r.t.* learning-based adversaries. In this work, let S be the set of obfuscation transformations and C be the original code, our goal is to identify an optimal transformation set $\langle o_1, o_2, \cdots, o_n \rangle$, where $\forall i \in [1, n]$. $o_i \in S$. Specifically, employing this set on C yields the optimal obfuscation[1]. This task, which is essentially an optimization process, poses following challenges. First, determining an objective function to navigate the search is non-straightforward since program obscurity is affected by multiple factors. Second, the search space for the optimal *configuration* is commonly quite huge, which means that full space enumeration is impractical. The state-of-the-art obfuscators, like Google Closure Compiler [1], adopt a fixed *configuration*. However, this strategy cannot necessarily guarantee an optimal obfuscation in practice. Lastly, the *double-edged sword* effect applies here. That said, an obfuscation o can improve the obscurity of some code c while degrade others, dependent on whether o finds target patterns in c. Normally, identifying the most matched code is non-trivial.

We proposed a unified framework for optimizing obfuscation, combining an obscurity language model and a fine-grained random search algorithm. We instantiated the framework for JavaScript programs, and verified that the model is capable of measuring obscurity and the algorithm shows efficacy and feasibility in obfuscation optimization.

2. BACKGROUND AND RELATED WORK

Program obfuscation has been intensively studied [7, 8, 21, 20]. Although no obfuscation is perfect [6, 13] , we can leverage it to deter reverse engineering and make software analyses (such as abstract interpretation [9]) imprecise [10, 12, 11]. Our framework offers a complementary vision to maximize the power of these works by identifying the most useful techniques on diverse code.

Additionally, our obscurity language model bridges obfuscation and natural language processing (NLP) techniques, which witnessed recent success in the context of programs [4, 15, 17]. Since software is *natural* [14], people resort to NLP methods for software engineering tasks, such as code completion [14, 16, 22], API prediction [19] and name suggestion [2, 3, 18]. We demonstrated the feasibility to relate obscurity with *naturalness* [14] and guide optimization tasks.

3. GENERAL APPROACH

[1]We call the transformation set *configuration* here

Our framework is built up on an obscurity language model (OLM) and a random search based engine. We take the original source code and a set of transformations as input, iteratively optimize the obfuscation, and output the obfuscated code in the end.

From the global picture, our obscurity language model extends the classical *n-gram* model and assigns a probability to the occurrence of an obfuscated code *w.r.t.* a large scale of corpus (*e.g.* all the available unobfuscated code on the web), in a sense that lower probability leads to higher obscurity. In OLM, the probability is called *perplexity*. Thus, the optimization of obfuscation amounts to producing code with higher *perplexity*. To achieve a high correlation between *perplexity* and obscurity, we train the model using two techniques: *type based tokens* (*TBT*) and *ordered variables* (*OV*). In *TBT*, we treat every token as a pair (*le*,*tt*) where *le* is the *lexeme* [14] and *tt* denotes the token type. When collecting tokens (*grams*) to build the model, we first check whether *tt* is involved in the obfuscation *w.r.t.* a set of transformations. If yes, we extract its *le*. Otherwise, we use *tt* instead. Generally, *TBT* is used to eliminate the noise of tokens which are not related to the obfuscation but may impact the *perplexity*. To further facilitate capturing regularity in obfuscated code, we proposed the *OV* which exploits the structural rather than textual obscurity. To this end, we first label variables and arguments enclosed in a scope (*e.g.* function) with the traversal order. Then, labels are trained to identify obscure code *w.r.t. natural* ones. Strength of *OV* lies in the capability of handling commonly used obfuscation techniques as variable renaming and reusing, which result in inherent high *perplexity* but are still vulnerable to learning-based adversaries. The obtained OLM assesses code in terms of obscurity by calculating *perplexity*, and then navigates the optimization process.

Based on OLM, we enforce a random search over the obfuscation space, which is commonly huge. Our goal is effectively sampling the space to identify optimal *configurations* which can maximize obscurity — *perplexity* as in OLM. We employ the *Monte Carlo Markov Chain* (MCMC) as the search engine. The search is divided into four steps. Firstly, we randomly propose a candidate *configuration*. The candidate is then applied on original code c to generate an obfuscated code c'. Thirdly, c' is delivered to OLM for obscurity assessing. Lastly, MCMC determines whether to accept the proposed candidate based on the *Metropolis-Hastings* algorithm [5]. Specifically, if *perplexity* given by OLM increases, MCMC accepts the candidate; Otherwise, MCMC accepts with a specific probability. Moreover, we proposed the *conflict removal* (*CR*) to further strengthen our framework. The key insight of *CR* is to enforce a fine-grained optimization in case that specific obfuscation optimizes a part of the program but degrade others at the same time. Obfuscators in the literature obfuscate programs as a whole. However, in our framework each granularity unit of a program is optimized independently so that the whole code can achieve the global optimal obscurity. To this end, we randomly apply candidate *configuration* on program units U (functions in this work) to generate a group of obfuscated units U'. Then, we assemble all the units together as the aforementioned obfuscated code c' for *perplexity* assessing. In this manner, *CR* is smoothly integrated in our framework to remove conflicts between program units due to applying universal obfuscation.

4. RESULTS AND CONTRIBUTION

We implemented our framework as an obfuscator called **Closure*** for JavaScript. To evaluate its effectiveness, we applied **Closure*** on popular open-source projects. As a representative of the emerging family of learning-based adversaries, we set JSNice [18] to be the attacker. Since JSNice shines on the capability of inferring variable names, we use the number of correctly recovered variables as the metric to assess obscurity[2]. First, we evaluated whether OLM can truly capture obscurity by ranking obfuscated code on both *perplexity* and obscurity as in Figure 1. Figre 1 displayed a tight correlation between *perplexity* and obscurity since the two ranks overlapped much (0.9743 coefficient), which verified the availability of OLM in assessing program obfuscation. Moreover, we compared **Closure*** with Google Closure Compiler (Closure) to exploit its optimization efficacy as in Table 1.

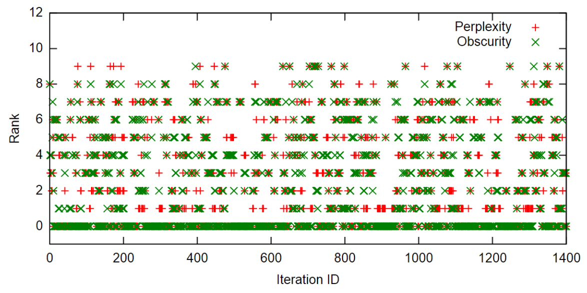

Figure 1: Ranks on *perplexity* and obscurity.

We chose projects from *trained* and *untrained* sets of JSNice [3]. For *trained* projects (20,000 GitHub stars on average), **Closure*** managed to achieve a promising obfuscation optimization by 26% compared to Closure. In terms of *untrained* projects where the code has already been well obfuscated by Closure, **Closure*** still exhibited a similar 22% optimization. Furthermore, either replacing MCMC algorithm with *Greedy* strategy or removal of *CR* decreased the optimization, which confirmed the superiority of MCMC and *CR*. We also asked 20 experienced programmers to attack code via recognizing core function related variables. The statistics revealed that **Closure*** blocked 20% more attacks and delayed the attack by 30% as well. That said, the achieved optimization is of practical benefits.

Table 1: Total — #variables. Closure & Closure* — #recovered variables (smaller is better). Improve — relative improvement (Closure* on Closure).

Project	Total	Closure	Closure*	Improve
Trained	1497	1211	891	26%
angular	248	205	171	17%
meteor	443	351	231	34%
react	134	86	68	21%
Untrained	791	218	171	22%
Greedy	1497	1211	1033	15%
Conflicts	1497	1211	928	23%

In general, the main contributions of this work are: first, we proposed the novel OLM as the first well-defined model to assess program obfuscation, which is sensitive to capture different degrees of obscurity. Second, we proposed an effective MCMC based optimization algorithm to guide the search for optimal *configuration* and also eliminate obfuscation conflicts. Lastly, we evaluated our framework on real-world popular JavaScript projects and achieved a promising optimization compared to a well-known obfuscator.

[2]Original file f_o is obfuscated and recovered into f_r by JSNice. We count how many names are correctly inferred.
[3]Trained projects are involved in building JSNice.

5. REFERENCES

[1] The Google Closure Compiler. https://developers.google.com/closure/compiler/, accessed: 2015-11-28.

[2] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 281–293, 2014.

[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, 2015.

[4] M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei. Bimodal modelling of source code and natural language. In F. R. Bach and D. M. Blei, editors, *ICML*, volume 37 of *JMLR Proceedings*, pages 2123–2132. JMLR.org, 2015.

[5] C. Andrieu, N. de Freitas, A. Doucet, and M. Jordan. An introduction to mcmc for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.

[6] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In J. Kilian, editor, *Advances in Cryptology âĂŤ CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2001.

[7] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. In *Technical report 148*, New Zealand, 1997. Department of computer science, the University of Auckland.

[8] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, 1998.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, 1977.

[10] M. Dalla Preda, I. Mastroeni, and R. Giacobazzi. A formal framework for property-driven obfuscation strategies. In *Fundamentals of Computation Theory*, volume 8070 of *Lecture Notes in Computer Science*, pages 133–144. Springer Berlin Heidelberg, 2013.

[11] R. Giacobazzi, N. D. Jones, and I. Mastroeni. Obfuscation by partial evaluation of distorted interpreters. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, pages 63–72, 2012.

[12] R. Giacobazzi and I. Mastroeni. Making abstract interpretation incomplete: Modeling the potency of obfuscation. In A. MinÃľ and D. Schmidt, editors, *Static Analysis*, volume 7460 of *Lecture Notes in Computer Science*, pages 129–145. Springer Berlin Heidelberg, 2012.

[13] S. Goldwasser and Y. Kalai. On the impossibility of obfuscation with auxiliary input. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 553–562, Oct 2005.

[14] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, 2012.

[15] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 803–813, New York, NY, USA, 2014. ACM.

[16] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 532–542, 2013.

[17] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 522–531, Piscataway, NJ, USA, 2013. IEEE Press.

[18] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, 2015.

[19] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, 2014.

[20] D. S., M. A., and T. C. Slicing aided design of obfuscating transforms. In *Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference on*, pages 1019–1024, 2007.

[21] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[22] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 269–280, 2014.