2017 IEEE/ACM 39th International Conference on Software Engineering

Stochastic Optimization of Program Obfuscation

Han Liu† Chengnian Sun* Zhendong Su* Yu Jiang†‡ Ming Gu† Jiaguang Sun†
†School of Software, Tsinghua University, Beijing, China
*University of California, Davis, USA
liuhan12@mails.tsinghua.edu.cn, {cnsun, su}@ucdavis.edu,
{jy1989, guming, sunjg}@mail.tsinghua.edu.cn

***Abstract*—Program obfuscation is a common practice in software development to obscure source code or binary code, in order to prevent humans from understanding the purpose or logic of software. It protects intellectual property and deters malicious attacks. While tremendous efforts have been devoted to the development of various obfuscation techniques, we have relatively little knowledge on how to most effectively use them together. The biggest challenge lies in identifying the most effective combination of obfuscation techniques.**

This paper presents a unified framework to optimize program obfuscation. Given an input program P and a set $\mathcal{T}$ of obfuscation transformations, our technique can automatically identify a sequence $seq = \langle t_1, t_2, \cdots, t_n \rangle$ ($\forall i \in [1, n].\ t_i \in \mathcal{T}$), such that applying t_i in order on P yields the optimal obfuscation performance. We model the process of searching for seq as a mathematical optimization problem. The key technical contributions of this paper are: (1) an *obscurity language model* to assess obfuscation effectiveness/optimality, and (2) a guided stochastic algorithm based on Markov chain Monte Carlo methods to search for the optimal solution seq.

We have realized the framework in a tool Closure* for JavaScript, and evaluated it on 25 most starred JavaScript projects on GitHub (19K lines of code). Our machinery study shows that Closure* outperforms the well-known Google Closure Compiler by defending 26% of the attacks initiated by JSNice. Our human study also reveals that Closure* is practical and can reduce the human attack success rate by 30%.

***Keywords*-program obfuscation; obscurity language model; markov chain monte carlo methods;**

I. Introduction

Software obfuscation is a deliberate act to hide the intention and logic of programs by obscuring source or executable code with semantics-preserving program transformations. It is a common approach against reverse engineering, and serves multiple purposes in practice, *e.g.*, protecting intellectual property, deterring malicious attacks. In this paper, we refer to any attempt to reverse engineer obfuscated code as an attack.

To defend against potential adversaries, decades of research has been devoted to developing various obfuscation techniques [1]–[6]. Some of these techniques manipulate the syntactical representation of programs (*e.g.*, renaming variables, changing format), while other advanced techniques complicate the control and data flow of programs under obfuscation.

These obfuscation techniques can be effective in deterring human adversaries who attempt to *manually* crack the obscurity by reading the code directly or with the help of static/dynamic analyzers. However, they might not be sufficient for deterring learning-based computer adversaries, a new and promising class of deobfuscators. These adversaries [7], [8] leverage coding features mined from a large corpus of source code to recover useful information (*e.g.*, identifier names, types) from obfuscated programs. Their evaluation results have demonstrated the potential of this class of deobfuscators at attacking obfuscated code.

The threat posed by learning-based adversaries motivates us to revisit the research of obfuscation. Although already having a large number of obfuscation techniques, we have little knowledge on how to coordinate them to produce better obfuscation result. Therefore, in this paper, we propose an automatic approach to optimize the obfuscation performance for a program. Specifically, given an input program P and a set $\mathcal{T}$ of obfuscation transformations, our technique identifies a sequence $seq = \langle t_1, t_2, \cdots, t_n \rangle$ ($\forall i \in [1, n].\ t_i \in \mathcal{T}$), such that applying t_i in order on P yields the optimal obfuscation performance. (We refer to this sequence as a *configuration* of obfuscation transformations.)

We model the process of searching for seq as a mathematical optimization problem, *i.e.*, finding an optimal configuration from all available configurations. Specifically we face the following challenges.

Challenge 1. We need an objective function to measure the obscurity of obfuscated programs. It is used to compare configurations in terms of obfuscation performance, and navigate the search process towards the optimal configurations. However there is no explicit, precise definition of such a function yet, since a number of factors can affect program obscurity, such as syntax, semantics and structure of programs, and even experience of adversaries.

Challenge 2. The search space for the optimal configuration is unbounded. It is infeasible to enumerate every configuration during the optimization process. The state-of-the-art obfuscators, such as Google Closure Compiler [9], specify a fixed order of obfuscation transformations for all programs. However, our evaluation in §VI shows that such a *statically specified configuration* cannot always yield good obfuscation results for various programs.

Challenge 3. Different obfuscation transformations might conflict with each other on the same source code, causing the obscurity to degrade (*e.g.*, §V Listing 5a, 5b and 5c). Given an obfuscation transformation t, instead of applying t to the whole program P, it is ideal to apply t to the rightful code regions in P, so that multiple obfuscation transformations can

‡Yu Jiang is the corresponding author.

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.28

be coordinated to achieve the optimal obscurity.

Guided Obfuscation Optimization. To overcome the three challenges, we propose a novel framework which *automatically* optimizes obfuscation for an input program.

(1) Obscurity Language Model (OLM) First, we propose an *obscurity language model* to assess the obfuscation. In particular, it measures obscurity based on the code perplexity of the obfuscated program against a large corpus of source code (*e.g.*, all the unobfuscated source code available on the web).

(2) MCMC-Based Search Second, we realize the optimization process by using *Markov chain Monte Carlo* (MCMC) [10] methods to search for the optimal *configuration* of obfuscation transformations. Our search strategy can efficiently sample the huge search space, so that within a bounded period of time we can find an optimal *configuration*.

(3) Program Decomposition Lastly, we stochastically decompose the large input program into fine-granularity units (*e.g.* functions), optimize the obfuscation for each unit individually, and then compose the obfuscated units back into a whole program. This decomposition process can decrease the likelihood of conflicts between obfuscation transformations.

We have realized the framework and instantiated it for JavaScript programs into a tool Closure*. The evaluation on real-world popular open-source projects (over 19K lines of code) demonstrates the ability of Closure* at combating the state-of-the-art deobfuscator JSNice [7]. In particular, Closure* outperforms Google Closure Compiler (Closure) by 26% in obfuscation performance, namely, Closure* protects 26% more information from being recovered by JSNice than Closure. In cases where Closure can already obfuscate the source code well, Closure* can still achieve 22% improvement. In order to ensure reproducibility, we have open-sourced Closure* at `https://bitbucket.org/njaliu/closure-star-tool`.

Contribution. We summarize our contributions as follows.

- We propose an *obscurity language model*, the first practical metric to assess the obscurity of obfuscated programs.
- We propose an effective and efficient MCMC-based algorithm to optimize program obfuscations.
- The comprehensive evaluation results demonstrate the effectiveness of our proposed technique. Compared to Google Closure Compiler, our realization for JavaScript programs exhibits 26% and 30% improvement on deterring learning-based and human attacks respectively.

Paper Organization. The remainder of the paper is organized as follows. §II introduces necessary background knowledge. §III presents the overall framework. §IV details the obscurity language model and §V elaborates the guided obfuscation optimization. We present our evaluation of Closure* in §VI and survey related work in §VII. §VIII concludes the paper.

II. Background

A. Program Obfuscation

Program obfuscation is a set of semantics-preserving program transformations to conceal programming intentions. It makes programs difficult to be understood by manual and automatic analyses. In this subsection, we describe the most common obfuscation transformations in the literature.

Name Obfuscation. This category of obfuscation transformations replaces identifier names (*e.g.*, variables, functions, classes) with meaningless or misleading ones. Take a declaration `var len` as an example. It is straightforward to conjecture that this variable is related to *length*. However, if the variable name is obfuscated to `a`, it will take more efforts to make the same conjecture.

Data Obfuscation. This category obfuscates data flows in programs via reusing variables, inlining variables, value encoding, *etc.*. For example, with `var max = f(arr); display(max);` we can easily figure out that the code is to select and display the maximal item in an array. But with the variable `max` inlined as `display(f(arr))`, we can hardly speculate the intention.

Control Flow Obfuscation. This category obfuscates the control flows of programs. Typical operations include inserting opaque predicates [1] whose value is hard to infer, flattening [11] and function inlining which can complicate the control flow and delay human understanding.

Layout Obfuscation. This category removes formatting (*e.g.* indentations, line breaks), and compresses the source code to reduce the readability and size of the code.

B. Learning-Based Adversary

Traditional adversaries of obfuscation aim at manually cracking the program obscurity, but they are limited by human experience. With the rapid development of machine learning techniques and accessibility of high-quality open-source projects, adversaries resort to learning-based attacks. Generally, such attacks try to recover information from obfuscated program elements, *e.g.*, giving a variable a meaningful name to expose its functionality.

Usually, the attack involves two phases. First, a knowledge model is built from a large corpus using machine learning algorithms. The model is able to estimate the probability for a program element to occur in the corpus. For example, the model can tell that `file = open('log','w')` is more likely to occur in a Python program than `a = open('log','w')`. Next, the adversary deobfuscates a program by querying the model, annotating program elements (*e.g.*, assigning meaningful names), and optimizing the annotations so that the deobfuscated program is most "similar" to the corpus.

We take JSNice [7] as a powerful instance. For name obfuscation as in §II-A, JSNice can correctly predict 63.4% of the obfuscated names [7], making the obfuscation greatly compromised. Figure 1a shows an obfuscated code with very short argument and variable names, which is difficult to understand. However, after JSNice recovers the names as shown in Figure 1b, we can easily know the function is copying an input string by iteratively retrieving its substrings. Even for complex large programs, JSNice can recover sensitive code elements within an acceptable time bound [7].

```
function chunkData(e, t) {
  var n = [], i = 0;
  var r = e.length;
  for (; i < r; i += t) {
    if (i + t < r) {
      n.push(e.substring(i, i + t));
    } else {
      n.push(e.substring(i, r));
    }
  }
  return n;
}
```

(a) Before attack

```
function chunkData(str, step) {
  var colNames = [], i = 0;
  var len = str.length;
  for (;i < len;i += step) {
    if (i + step < len) {
      colNames.push(str.substring(i, i + step));
    } else {
      colNames.push(str.substring(i, len));
    }
  }
  return colNames;
}
```

(b) After attack

Fig. 1: A code snippet deobfuscated by JSNice.

C. Language Model

Language models (LM) assign probabilities to different sequences of *words*. The probabilities, in turn, indicate how likely the sequence is to occur. In the context of programming languages, researchers have investigated such probabilistic nature (called *naturalness*) of programs [12] and highlighted its promising potential in handling traditional software engineering tasks [13]–[15], *e.g.*, for code completion, given code snippet `for{`, the LM can predict `for{int i=0;i<` is the most possible code to follow. For a token sequence $s = t_1 t_2 \cdots t_n$, its LM probability is $P(s) = P(t_1) \cdot \prod_{i=2}^{n} P(t_i | t_1, \cdots, t_{i-1})$. Each conditional probability determines how likely a subsequence is to follow its prefix. In practice, estimating the conditional probability is usually difficult or even infeasible due to the huge number of prefixes. A practical approximation is the *n-gram* model, which assumes that the occurrence of a token is dependent on a limited prefix with length n. This way, $P(t_i | t_1, \cdots, t_{i-1})$ is approximated to $P(t_i | t_{i-n+1} \cdots t_{i-1})$, which is computed by counting the occurrences below

$$P(t_i | t_1, \cdots, t_{i-1}) = \frac{count(t_{i-(n-1)} \cdots t_{i-1}, t_i)}{count(t_{i-(n-1)} \cdots t_{i-1})} \quad \text{(N-gram)}$$

Given a program $s = t_1 t_2 \cdots t_n$, to better interpret its *naturalness* based on an LM $\mathcal{M}$, we use the measurement *perplexity* or its log-transformed version *cross-entropy* [16], defined as $H_{\mathcal{M}}(s) = -\frac{1}{n} \log p_{\mathcal{M}}(t_1 \cdots t_n)$. Based on the *n-gram* ($n = k$), the formulation accordingly amounts to

$$H_{\mathcal{M}}(s) = -\frac{1}{n} \sum_{1}^{n} \log p_{\mathcal{M}}(t_i \mid t_{i-k+1} \cdots t_{i-1}) \quad \text{(Perplexity)}$$

Commonly, code with high *perplexity* is "oddly" written. In our setting, such oddness is likely a result of obfuscation. A well-designed LM should be able to identify to what degree a program is obfuscated via the *perplexity* measurement.

III. Problem Formulation & Overall Framework

In this section, we formulate the problem of obfuscation optimization, and briefly introduce the workflow of our proposed approach.

A. Problem Formulation

Generally, obfuscation refers to program transformations without changing the program's behavior. Although already having a large number of obfuscation techniques, we have little knowledge on how to coordinate them to produce better obfuscation result. Therefore, in this paper, we propose an automatic approach to optimize the obfuscation performance for a program.

*Definition 3.1 (**Obfuscation Optimization**):* Given an input program P and a set $\mathcal{T}$ of obfuscation transformations, the problem of obfuscation optimization identifies a sequence $seq = \langle t_1, t_2, \cdots, t_n \rangle$ ($\forall i \in [1, n].\ t_i \in \mathcal{T}$), such that applying t_i in order on P yields the optimal obfuscation performance.

Configuration. We refer to the sequence seq as a *configuration* of obfuscation transformations. The state-of-the-art obfuscators enforce a statically determined configuration for every input program. Their belief behind is *works for one, works for all*. Our technique distinguishes itself from those by applying input-dependent obfuscation transformations. That is, for each input program, we aim to find the most suitable configuration.

Measuring Optimality. The optimality of a configuration measures the degree of difficulty for an adversary to deobfuscate an obfuscated program. As stated in Challenge 1 in §I, there is no explicit, precise definition of such a measure yet. Therefore, in this paper, we propose an obscurity language model (OLM) (detailed in §IV) to measure the optimality by analyzing the structural similarity between an obfuscated program and a large corpus of programs that are available online. Specifically, this is done by computing the *perplexity* (Equation Perplexity). The higher *perplexity* an obfuscated program has, the odder it is *w.r.t.* the corpus, and the harder it is for an adversary to deobfuscate the program.

B. Workflow

As aforementioned, we model the search process for the optimal configuration of obfuscation transformations as a mathematical optimization problem. That is, we iteratively explore the obfuscation space to find better *configurations* such that the obscurity of programs is improved. In particular, we designed a search algorithm based on Markov chain Monte Carlo methods, which steers the search process towards the optimal configuration with the guidance of the obscurity language model.

The general workflow of our framework is shown in Figure 2. It takes as input the source file for obfuscation and the set $\mathcal{T}$ of available obfuscation transformations, and outputs an obfuscated program by an optimal configuration. The obfuscation process can be divided into the following four components.

Parse At first, the input source file is parsed into an abstract syntax tree (AST). The other components all work on ASTs.

Search We decompose the original program at function level. The obfuscation engine then randomly selects a set of functions to form a partition and generate a *configuration* to obfuscate it.

Assess We enclose the obfuscated code by the search component into a query to the OLM. Then OLM computes the

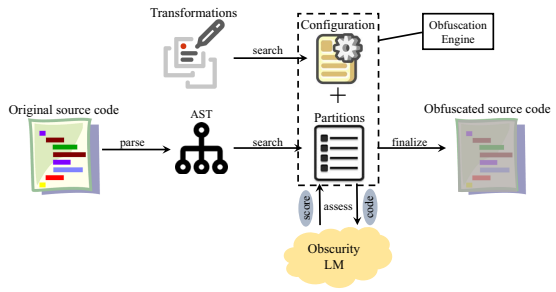


Fig. 2: The workflow of the proposed obfuscation framework.

perplexity of the code against a corpus of programs, and returns it as a "score" to the obfuscation engine.

Finalize After a parameterized number of iterations of *search* and *assess* (*e.g.*, 10K iterations), we choose the iteration with the highest perplexity score and output the obfuscated program in that iteration as the final result.

In a nutshell, our framework iteratively searches for more effective *configuration* that produces better obfuscation result. The search process is guided by the OLM so that it can eventually converge to an optimal *configuration*.

IV. Obscurity Language Model

This section describes the general process to build a language model for software engineering tasks, difference and challenge of building an obscurity language model, and how we address the challenges.

A. Building a General Source Code Language Model

As stated in [12], building a general language model for source code aims at capturing the statistical regularities of code. This model is then used for software engineering tasks at the source code level, such as code completion. Therefore, in order to build such a model, a program is first tokenized and represented as a sequence of lexemes. Finally, an n-gram model is built by computing the conditional probability of a lexeme t_i given its prefix $\langle t_{i-n+1}, \cdots, t_{i-1} \rangle$ with the formula in Equation N-gram.

Recent techniques [14], [15] can build better language models to capture the regularities of programs, by either associating the lexemes with semantic information or taking the localness of lexemes into account during the model training phase. However, both still work on the lexeme level.

B. Challenges of Building Obscurity Language Models

Different from a traditional language model for software engineering tasks at *unobfuscated* source code level (*e.g.*, code completion) [12], [14], [15], our OLM aims to capture the remaining regularities of software after obfuscation. In other words, the OLM measures the perplexity between the obfuscated program and a large corpus of programs.

However, if we use the traditional language model, an obfuscated program is inherently perplex as its variables are renamed to short, meaningless names, and data/control flows are altered. Our OLM should be resilient to the perplexity induced by obfuscators, and should be able to measure the *remaining regularities* induced by the source code itself. The fewer regularities of the original source code remain, (*i.e.*, higher perplexity) the better the configuration that obfuscates the program is. Therefore in this case, representing programs with sequences of *lexemes* is insufficient and even impractical. We detail the challenges in the following.

1) Inherent Perplexity of Obfuscated Programs: An obfuscated program has short and meaningless variable names, which results in an inherently high *perplexity* if it is measured by the traditional software engineering language model over the lexeme-based program representations. An illustrative example, which contains two obfuscated versions of JQuery[1] is shown in Figure 3.

```
function cloneCopyEvent(e, a) {
  var t, s, d, n, r, v, i, c;
  if (1 === a.nodeType) {
    if (predicate(e)) {
      n = G0.access(e), r = G0.set(a, n);
      c = n.events;
      delete r.handle,
      r.events = {};
      for (bca(t,s,c,d))
        jQuery.event.add(a, d, c[d][t]);
    }
    G1.hasData(e) && (v = G1.access(e),
    i = jQuery.extend({}, v), G1.set(a, i));
  }
}
```

(a) Variable renaming

```
function cloneCopyEvent(d, e) {
  var b, f, a, c;
  if (1 === e.nodeType) {
    if (predicate(d)) {
    c = G0.access(d), b = G0.set(e, c);
    c = c.events;
    delete b.handle,
    b.events = {};
    for (bca(a,b,c,f))
      jQuery.event.add(e, a, c[a][b]);
    }
    G1.hasData(d) && (a = G1.access(d),
    a = jQuery.extend({}, a), G1.set(e, a));
  }
}
```

(b) Variable renaming and reusing

Fig. 3: Inherent *perplexity* due to obfuscated names

Figure 3a is obfuscated with variable renaming, and Figure 3b is obfuscated with variable renaming and reusing. Both code snippets have high perplexity when they are measured with the original source code, as their lexeme-based representations are very different from that of the original one (namely, all local variables and function arguments have different names/lexemes). Thus, the traditional language model cannot differentiate which version is better obfuscated.

However, Figure 3b is better obfuscated, as its data flows are also obfuscated by reusing 4 of the 8 local variables in Figure 3a. Using JSNice [7] as an automatic adversary to attack the two code snippets also confirms this. Specifically, 1 variable is obfuscated in Figure 3a and 5 variables are obfuscated in Figure 3b. Figure 3b outperforms Figure 3a by protecting more information. Our OLM is expected to differentiate the superiority of Figure 3b in terms of obfuscation quality.

C. Obeme-Based Obscurity Language Model

To address the challenges above, we use *obemes* to build an OLM instead of lexemes. Generally, an obeme is an enhanced representation of a token by considering its lexeme, type and variable ordering. Its advantage is the capability of handling inherent obscurity of obfuscated source code. Given a token $tk = (l, t)$ (where l is the lexeme and t is the token type), then

[1] https://github.com/jquery/jquery/blob/master/src/manipulation.js

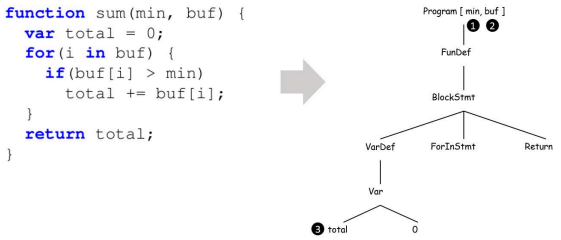


Fig. 4: A simple function and its corresponding AST.

its obeme is defined as follows.

$$obeme(tk) = \begin{cases} v + order, & tk \text{ is a variable or an argument} \\ t, & tk \text{ is a literal, } e.g., \text{ 1, "string"} \\ l, & \text{otherwise} \end{cases}$$

Variables and Arguments. If the token tk is a local variable or a function argument, we use its variable ordering in its scope as its obeme. Given the abstract syntax tree (AST) of a program, we define the *order* $\in \mathbb{N}$ of a local variable to be the relative position in the sequence generated by an AST traversal. Specially, the *order* of a function argument is set to be the top within the function scope according to its relative position in the argument list.

An example is shown in Figure 4. The function `sum` has two arguments `min` and `buf`, and one local variable `total`. Under pre-order traversal, `min`, `buf` and `total` are visited in turn. Thus, their orders are 1, 2 and 3 respectively. To avoid name clashing with integer literals, we prefix the *order* with a marker string 'v' to form an *obeme*. In this way, variables are evaluated for *perplexity* based on the program structure rather than their names.

Literals. For literals in programs (*e.g.*, strings, numbers, characters), we use their token types as their obemes, as we focus on structural similarity between obfuscated programs and program corpus and any difference between literals has little impact on the structural similarity.

Others. For other types of tokens, (*e.g.*, key words, parentheses, brackets), we use their lexemes as the obemes.

D. Obfuscation Assessment

After training an OLM from the obeme-based representations of a program corpus, we can quantitatively measure how obscure a program is with its perplexity. The measurement serves as the guidance in each optimization iteration described in §III, steering the search process towards better obfuscation configurations that can produce more perplex obfuscated programs. A key property of the obscurity LM is the highly positive correlation between perplexity and program obscurity, which we have validated empirically in §VI-D.

V. Guided Obfuscation Optimization

This section details the MCMC-based obfuscation engine, and our main strategy to eliminate conflicts between obfuscation transformations at the function level.

A. MCMC-Based Obfuscation Optimization

Given a program P to obfuscate, an OLM $\mathcal{M}$ built from a corpus of programs, a set $\mathcal{T}$ of available obfuscation transformations and a configuration seq, the perplexity of P' that is obfuscated from P by seq (*i.e.*, $P' = seq(P)$) is denoted as

$$\Delta(P; seq; \mathcal{M}) = H_{\mathcal{M}}(seq(P)) \qquad \text{(Objective Function)}$$

As aforementioned, the perplexity measures the obscurity and serves as the objective function for optimizing obfuscation. Therefore, the goal of the obfuscation optimization can be formally expressed as

$$seq^* = \operatorname{argmax} \Delta(P; seq; \mathcal{M})$$

That is, find an optimal configuration seq^* such that maximizes the perplexity of the obfuscated version of P.

In this paper, we employ MCMC sampling [10] to find seq^*. MCMC has been proved and demonstrated to be effective at estimating a *target* probability distribution for which the direct sampling is difficult. For example, in the setting of obfuscation, the space of *configurations* is unbounded if we allow duplications of obfuscation transformations in a configuration. In such cases, MCMC can mitigate the complexity by sampling more often in the region of configurations which yields better perplexity.

Specifically, we use the *Metropolis-Hastings* algorithm to sample a sequence of configurations $\langle seq_0, seq_1, \ldots, seq_n \rangle$ in the configuration space. The *target* probability distribution from which we draw samples is defined as

$$\eta(P; seq; \mathcal{M}) = \frac{1}{Z} \exp(\sigma \cdot \Delta(P; seq; \mathcal{M})) \qquad (1)$$

As described in [17], σ is a constant and Z is a partition function that normalizes the target distribution. A significant property of η is that *higher* perplexity leads to *higher* probability. Suppose seq' is the proposed candidate sample, the sampling process accepts the candidate and moves to seq' with a probability as below:

$$\mathcal{A}(seq \to seq'; P; \mathcal{M}) = \min(1, \frac{\eta(P; seq'; \mathcal{M})}{\eta(P; seq; \mathcal{M})}) \qquad (2)$$

Particularly, **`Metropolis-Hastings`** algorithm enables us to accept a new sample without computing the partition function (Z in Equation 1) since Z is canceled out by the division.

The overall MCMC based optimization process is shown in Algorithm 1. On line 1, the initial sample is a configuration obtained by shuffling all the transformations in $\mathcal{T}$. Then at each iteration of the optimization, we propose a new sample based on the current sample. This is realized by the function `mutate` on line 3. Specifically, the mutation is done by updating a random small number of transformations in the current *configuration*. The update can be removal, insertion and substitution. Next, we evaluate the new sample by querying the acceptance function and choose to replace the current sample with the new one (line 4-5). After the optimization iteration, we reproduce the recorded best obfuscation and output the obfuscated program.

Algorithm 1: MCMC-Based Obfuscation Optimization.

Input : P is the input program.
$\mathcal{T}$ is the set of obfuscation transformations.
$\mathcal{M}$ is the language model trained as in §IV.
N is the number of iterations.
Output: the obfuscated version produced by the optimal configuration.

$seq \leftarrow \texttt{shuffle}(\mathcal{T})$, $seq^* \leftarrow seq$
// Generate MCMC samples.
for $i \leftarrow 1$ *to* N **do**
  $seq' \leftarrow \texttt{mutate}(seq)$
  // rand() simulates the uniform distribution.
  if $\texttt{rand}() < \mathcal{A}(seq \rightarrow seq'; P; \mathcal{M})$ **then**
    $seq \leftarrow seq'$
    if $\Delta(P; seq; \mathcal{M}) > \Delta(P; seq^*; \mathcal{M})$ **then**
      $seq^* \leftarrow seq$
// Reproduce the best obfuscation.
return $seq^*(P)$

Algorithm 2: Obfuscation Optimization with Partitioning.

Input : P is the input program.
$\mathcal{T}$ is the set of obfuscation transformations.
$\mathcal{M}$ is the language model trained as in §IV.
N is the number of iterations.
N_{part} is the number to partition P
Output: the obfuscated version produced by the optimal configuration.

$F_{obf} \leftarrow \varnothing$, $F_{orig} \leftarrow P$
for $p \leftarrow 1$ **to** N_{part} **do**
  $seq \leftarrow \texttt{shuffle}(\mathcal{T})$, $seq^* \leftarrow seq$, $F_{test} \leftarrow \varnothing$
  for $i \leftarrow 1$ *to* N **do**
    if $p = N_{part}$ **then**
      $partition \leftarrow F_{orig}$
    else
      $partition \leftarrow \texttt{random\_sample}(F_{orig})$
    $P \leftarrow (F_{obf}, partition, F_{orig} \setminus partition)$
    $seq' \leftarrow \texttt{mutate}(seq)$
    // rand() simulates the uniform distribution.
    if $\texttt{rand}() < \mathcal{A}(seq \rightarrow seq'; P'; \mathcal{M})$ **then**
      $seq \leftarrow seq'$
      if $\Delta(P'; seq; \mathcal{M}) > \Delta(P'; seq^*; \mathcal{M})$ **then**
        $seq^* \leftarrow seq$, $F_{test} \leftarrow partition$
  $F_{obf} \leftarrow F_{obf} \cup seq^*(F_{test})$, $F_{orig} \leftarrow F_{orig} \setminus F_{test}$
return F_{obf}

B. Conflicting Function Optimization

As introduced in §I, our framework is able to identify the rightful code regions to apply the suitable obfuscation transformations. This is motivated by an observation we found in the very beginning of this project, that a configuration sometimes fails to improve the obscurity of multiple functions at the same time.

1) An Example: Figure 5 shows a real-world example. The code snippet is extracted from the popular open-source JavaScript project jade[2], which is a widely-used (9,637 stars on GitHub) templating language for producing XML like documents.

In the example, we consider the obfuscation transformation — folding constants `foldConstants`. When enabled, `foldConstants` recognizes and evaluates constant expressions, and uses the values to replace the expressions. Figure 5a shows the original source code (two functions `handleTemplateCache` and `bracketExpression`). Existing obfuscators apply transformations on the whole program, *i.e.*, for this example both functions. By disabling and enabling `foldConstants`, two obfuscated programs are generated in Figure 5b and 5c respectively. For example, `undefined` is folded to `0` in Figure 5c.

Then, we employ JSNice to recover names for local variables and function arguments. The recovered names are shown in comments before each function. We found that: when `foldConstants` is disabled, function `handleTemplateCache` is well protected (3 variable obfuscated) but `bracketExpression` is not (0 variables obfuscated); when `foldConstants` is enabled, `bracketExpression` is well protected (2 variable obfuscated) while `handleTemplateCache` is not (2 variables obfuscated). That said, the two functions have different optimal *configurations*. An intuitive fix for this conflict is to enable `foldConstants` for one and disable `foldConstants` for the other. Therefore, it is ideal to identify the suitable obfuscation transformations for the rightful code regions.

2) Obfuscation Optimization with Partitioning: We address this challenge by iteratively decomposing the program P and find the optimal configuration for each partition. Specifically, given a number N_{part} to partition P at the function level, we call Algorithm 1 N_{part} times. One call produces a partition (*i.e.*, a set of functions) with its optimal configuration. The subsequent call produces another partition, containing functions that are not covered by the partitions produced in previous calls.

Let the program P (*i.e.*, a set of functions) be represented as a triple $(F_{obf}, F_{test}, F_{orig})$, where F_{obf} is the set of functions in P that is already optimally obfuscated, F_{test} is the set of functions that is being obfuscated, and F_{orig} is the set that has not been obfuscated. Given an obfuscation configuration seq, then the semantics of $seq(P)$ is refined as follows,

$$seq(P) = seq(F_{obf}, F_{test}, F_{orig}) = F_{obf} \cup seq(F_{test}) \cup F_{orig}$$

which can take as input either a set of functions or a triple of sets.

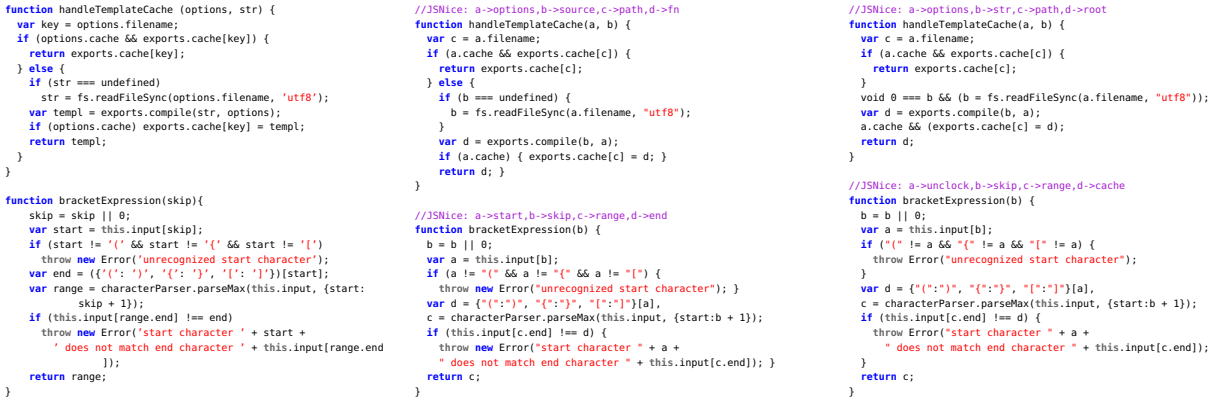
Algorithm 2 details the partitioning process with MCMC search for optimal configurations. The general idea is that we gradually obfuscate the program P in at most N_{part} steps, in each of which a partition of P is obfuscated.

C. Runtime Performance Optimization

Generally, the *automatic* optimization is a tradeoff between obscurity and execution performance. In order to balance the tradeoff, we design several heuristics to reduce unnecessary search/optimization.

First, we exclude *small functions* which have a very small number of local variables and arguments, and functions whose bodies mainly call APIs, because there is little improvement room for obfuscation in such functions. Second, we dynamically record how *perplexity* changes over optimization iterations. If the change is tiny, we terminate further search to save time.

[2]https://github.com/jadejs/jade/blob/1.9.0/jade.js



```
function handleTemplateCache (options, str) {
  var key = options.filename;
  if (options.cache && exports.cache[key]) {
    return exports.cache[key];
  } else {
    if (str === undefined)
      str = fs.readFileSync(options.filename, 'utf8');
    var templ = exports.compile(str, options);
    if (options.cache) exports.cache[key] = templ;
    return templ;
  }
}

function bracketExpression(skip){
    skip = skip || 0;
    var start = this.input[skip];
    if (start != '(' && start != '{' && start != '[')
      throw new Error('unrecognized start character');
    var end = ({'(': ')', '{': '}', '[': ']'})[start];
    var range = characterParser.parseMax(this.input, {start:
            skip + 1});
    if (this.input[range.end] !== end)
      throw new Error('start character ' + start +
        ' does not match end character ' + this.input[range.end
                ]);
    return range;
}
```

(a) Original Source Code

```
//JSNice: a->options,b->source,c->path,d->fn
function handleTemplateCache(a, b) {
  var c = a.filename;
  if (a.cache && exports.cache[c]) {
    return exports.cache[c];
  } else {
    if (b === undefined) {
      b = fs.readFileSync(a.filename, "utf8");
    }
    var d = exports.compile(b, a);
    if (a.cache) { exports.cache[c] = d; }
    return d; }
}

//JSNice: a->start,b->skip,c->range,d->end
function bracketExpression(b) {
  b = b || 0;
  var a = this.input[b];
  if (a != "(" && a != "{" && a != "[") {
    throw new Error("unrecognized start character"); }
  var d = {"(":")", "{":"}", "[":"]"}[a],
  c = characterParser.parseMax(this.input, {start:b + 1});
  if (this.input[c.end] !== d) {
    throw new Error("start character " + a +
    " does not match end character " + this.input[c.end]); }
  return c;
}
```

(b) `foldConstants` is *disabled*.

```
//JSNice: a->options,b->str,c->path,d->root
function handleTemplateCache(a, b) {
  var c = a.filename;
  if (a.cache && exports.cache[c]) {
    return exports.cache[c];
  }
  void 0 === b && (b = fs.readFileSync(a.filename, "utf8"));
  var d = exports.compile(b, a);
  a.cache && (exports.cache[c] = d);
  return d;
}

//JSNice: a->unclock,b->skip,c->range,d->cache
function bracketExpression(b) {
  b = b || 0;
  var a = this.input[b];
  if ("(" != a && "{" != a && "[" != a) {
    throw Error("unrecognized start character");
  }
  var d = {"(":")", "{":"}", "[":"]"}[a],
  c = characterParser.parseMax(this.input, {start:b + 1});
  if (this.input[c.end] !== d) {
    throw Error("start character " + a +
      " does not match end character " + this.input[c.end]);
  }
  return c;
}
```

(c) `foldConstants` is *enabled*

Fig. 5: A real-world example of conflicting functions

VI. Implementation and Evaluation

A. Instantiation for JavaScript

We instantiated the proposed framework for JavaScript (JS) and developed the Closure* tool. The implementation includes 3,342 LOC NodeJS, 365 LOC Python and 51 LOC Java. We made it publicly available at `https://bitbucket.org/njaliu/closure-star-tool`.

B. Experimental Setup

All the experiments are performed on a Ubuntu 14.04 virtual machine with dual Intel Core i5 processors, 10GB RAM and 128GB SSD. In evaluation, the obscurity LM is configured to be *5-gram* using KenLM [18]. The number of partitions $N_{part} = 2$. As for the adversary to Closure*, we use the UnuglifyJS front-end [19] and the Nice2Predict underlying machine learning engine [20], which are both from JSNice [7]. We use 25 top active open source JavaScript projects from GitHub. Most of the projects are selected from the most-stars list. From the perspective of the adversary JSNice, the experimental projects can be classified into two categories. We use the term "*normal*" to refer to projects which are included in the training data of JSNice, while "*obfuscated*" is used for others. Inherently, JSNice should perform better attack on "*normal*" input than "*obfuscated*" ones. We check how our approach responds to attack on both types. One of the major features of JSNice is to assign meaningful names to variables. Therefore, after the MCMC search iteration, we apply JSNice to attack the obfuscated file and quantify the obfuscation by counting the number of variables which are correctly recovered. We believe this makes sense for two reasons. First, the quality of JSNice prediction is dependent on connecting the program to its training corpus, which means that obscurity of the program can be reflected by how variables names are predicted. Second, people leverage much on names to understand code. More variables are predicted with meaningful names, less difficult the program is to be well understood.

Baseline Obfuscation. We selected Google Closure Compiler as a contrast for Closure*. The main contribution of this work is to optimize obfuscation (or transformations in general) by searching for an effective *configuration*, which is orthogonal to the underlying obfuscators. While featured as an optimizer for JavaScript [9], such tools are characterized as a form of obfuscators by JSNice [7] (They used UglifyJS, which is similar to Closure). Moreover, Closure excels at limiting the code size and runtime overhead, making it suitable for real-world usage. We have conducted a preliminary study to compare Closure with commercial obfuscators (available at `https://bitbucket.org/njaliu/closure-star-tool`), showing that Closure generates 56% smaller but equally obfuscated code compared to commercial tools. Lastly, Closure is open-sourced, enabling us to flexibly generate random *configurations*.

C. Research Questions

In this paper, we present the insight of input-dependent program obfuscation and its optimization can be modeled as an optimization task to search for highest *perplexity*. We intend to address the following research questions:

RQ 1. Can the obscurity language model capture the obscurity?
RQ 2. Can the MCMC random search optimize obfuscation?
RQ 3. Does the optimization give us practical benefits?

D. Results and Discussions

Now, we present the evaluation results and multi-dimensional in-depth analysis as well. To begin with, we try to investigate the quality of the obscurity language model, figuring out whether the measurement *perplexity* is consistent with the obscurity of programs. Towards this goal, we monitor the obfuscation optimization process to record the intermediate output at each iteration. Specifically, we rank those files on both *perplexity* and obscurity (which is inversely proportional to the number of correctly recovered variable names), and calculate their *rank distance*. Given an input program and the number of iterations N, the ranks on *perplexity* x_i and obscurity y_i at iteration i ($1 \leq i \leq \mathsf{N}$) satisfy that $x_i, y_i \in [0, \mathsf{N}-1]$. The

rank distance D_i is defined as $D_i = |x_i - y_i|$. Accordingly, $D_i \in [0, \mathsf{N}-1]$. Overall, the monitoring leads to 3,417 records with a distribution shown in Figure 6.

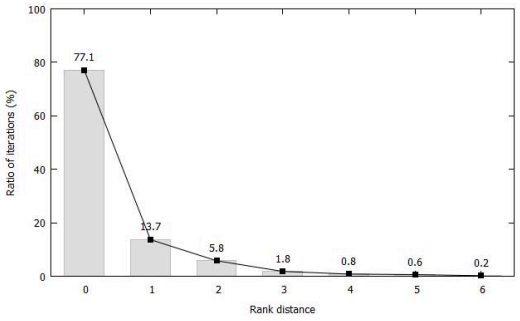


Fig. 6: Rank distance between *perplexity* and obscurity. There are no iterations with rank distance larger than 6 observed within the overall monitored records.

TABLE I: On the different order of the LM. **ARD**: Average Rank Distance. **COEF**: the correlation coefficient.

n-gram	ARD	COEF
3	0.38	0.9596
4	0.37	0.9683
5	0.38	0.9743

The definition of *rank distance* delivers the message that smaller distance indicates better consistency between *perplexity* and obscurity, vise versa. In our setting, *perplexity* calculated by the obscurity LM is used to identify the obfuscation improvement and drive the optimization. That said, we rely on the rank of *perplexity* rather than its absolute values. In that sense, *rank distance* is sufficient for characterizing the efficacy and sensitivity of the obscurity LM. As Figure 6 shows, most iterations (close to 80%) generate a zero distance, which means that *perplexity* is perfectly consistent with obscurity. Furthermore, 1-distance and 2-distance cumulatively increase the ratio to almost 100%. Larger distances (*e.g.*, from 3 to 6), which expose non-negligible inconsistency, rarely occur in the records. From the global picture, Figure 6 gives us an intuitive confidence that our LM is consistent. To be more precise, we further calculate two support indicators in Table I: (1) the average *rank distance* and (2) the correlation coefficient which illustrates a quantitative measure of correlation and dependence valued from -1 to 1. While both indicators describe the quality of an obscurity LM, coefficient is of the most practical use via mirroring the capability of the LM to locate a powerful obfuscation. In the setting of Figure 6 which uses *5-gram*, the obscurity LM shows strong consistency in all the indicators. The 0.9743 coefficient further confirms that *perplexity* and obscurity are close to a total positive correlation. If we modify the order of *n-gram* ($n = 3, 4$), the consistency decreases slightly in terms of coefficient, which suggests that the *5-gram* LM fits our setting the best.

Based on the observation above, for **RQ1** we believe that the language model we built is quite capable of capturing obscurity during the optimization. In another word, it is reasonable to rely on it to guide the obfuscation process.

TABLE II: Obfuscation efficacy of Closure* compared to Closure on *normal* and *obfuscated* projects. **Total** column refers to the number of all variables considered (including local variables and function arguments). **Closure** and **Closure*** columns list the number of variables which are attacked (correctly recovered) by JSNice, which means that smaller value amounts to better obfuscation. **Improve** column calculates the relative improvement of Closure* over Closure.

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

Next, we compare the obfuscation between Closure* and Closure `SIMPLE` mode (which is the default setting). The detailed statistics is displayed in Figure 7 and Table II. To be specific, 54 files with core functionalities from the *normal* projects are selected as in left of Figure 7. Clearly, Closure* outperforms Closure for the major portion of the files. In terms of the rest, Closure* is able to achieve the same obfuscation as Closure. The red arrow demonstrated the maximal improvement, where the optimized obfuscation protected 102 more variables (65%) from attack. Considering all the *normal* projects, we calculated the relative improvement over Closure in Table II. Under the attack of JSNice, Closure* optimized the obfuscation of Closure by protecting 26% more variables. We also listed results for top three popular projects: *angular* [21], *meteor* [22] and *react* [23], which have 20,000 GitHub stars on average. Likewise, Closure* exhibits an average 24% optimization. Similarly, we carried the experiments on *obfuscated* projects[3]. From right part of Figure 7, we find that the maximal improvement involves 17 variables (29%). According to Table II, Closure* managed to reduce the number of recovered variables from 28% (in Closure) to 22%. Although the absolute improvement is not as big as in *normal* projects, the relative improvement tends to be close (22%). The achieved optimization can be seen as effective transformations to confuse the attacking process, making adversaries to infer in a wrong direction. In addition, we run our sampling with greedy strategy and without conflict removal (last two rows in Table II). The optimization decreases to 15% and 23% respectively, which embodies the necessity of our MCMC random search with conflicting function elimination. We also conduct the Wilcoxon Signed-Rank Test [24] over obfuscation results from Closure and Closure* to validate the statistical significance. With the 1.783 *z-score* which does not exceed the critical value 1.960 according to [25], the optimization is statistically significant. In general, our findings

[3]*obfuscated* projects are those on which JSNice can only recover small number of variables.

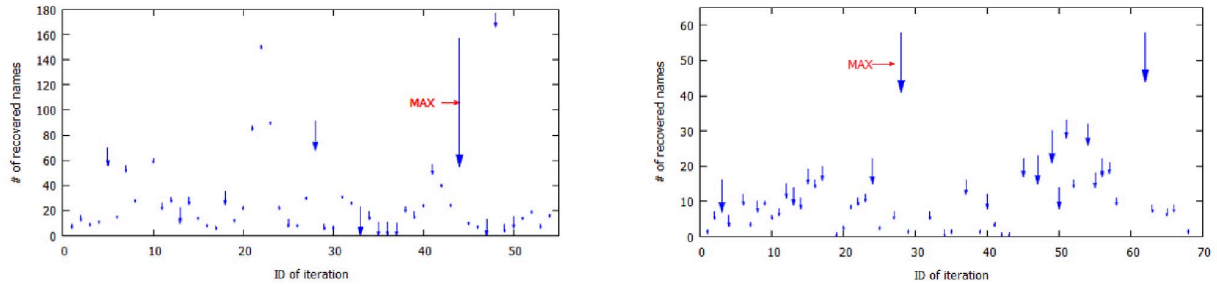


Fig. 7: Obfuscation optimization on part of *normal* projects (**Left**) and *obfuscated* projects (**Right**). Arrow pointing down means Closure* is better, up means Closure is better. The longer an arrow is, the greater the optimization is.

answer **RQ2** in the affirmative, and also convince us that the efficacy of Closure* is not limited in the training corpus of the obscurity LM. On the other hand, the optimized obfuscation transformations is not always the same *configuration* as Closure, particularly some use fewer while some use more. This also hints us that the assumption *Adding a transformation is always good* may not hold.

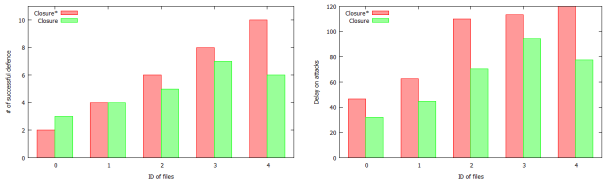


Fig. 8: For attacks on Closure* and Closure, **Left** shows the number of successful defence. For those failed defence, **Right** summarizes how much extra delay Closure* and Closure can incur.

In real-world applications, a major strength of obfuscation is the incurred delay on possible malicious use. To further exploit the practical benefits of the optimized obfuscation, we investigate how much extra difficulty is added compared to Closure in terms of manual attacks. We deliver 10 obfuscated files (17-42 LOC, from Closure* or Closure) of *meteor* project to 20 programmers (10 PhD and 10 master, both have at least 2-year JavaScript programming experience), asking them to identify the core functionality related variables by assigning given meaningful names. The accuracy and delay on the tasks (which interpret the practical benefits) are summarized in Figure 8.

Here, we count (1) how many attacks are successfully defended using Closure* and Closure, respectively (2) how long is the delay on effective attacks. Clearly, Closure* shows better defense by blocking 5 more (20% for Closure) attacks in total. With respect to other misses, Closure* is also able to increase the attack delay by 30% on average. With the growth on file size, this advantage can be amplified, disabling more potential attacks. In sum, we believe Closure* offers complementary power to defend practical attacks on obfuscation, which is probably relied on human understanding and learning-based tools as well. Therefore, we can reply to **RQ3** with a positive answer. In the future, intensive experiments are planned on more professional attackers.

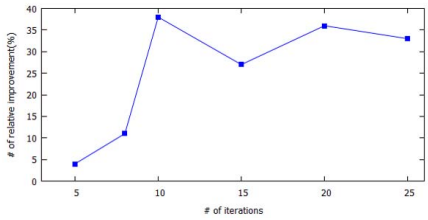


Fig. 9: Execute different numbers of iterations to evaluate the achieved obfuscation improvement.

Furthermore, we analyzed how the number of iterations affected obfuscation improvement. We randomly select 20 files and employ different numbers of iterations for obfuscation. The tuning process is shown in Figure 9. In principle, obfuscating larger programs should be more difficult than small ones since it is harder to locate a good obfuscation within bounded search. Generally, the obfuscation improvement grows fast with the increase on number of iterations when the total number is small (N = 5,8,10). However, for large numbers (N = 20,25), the observed improvement remains at a similar level. The trend can be explained as: the obfuscation is optimized when the obfuscator is able to locate suitable transformations. Commonly, the effective *configuration* combines specific set of obfuscation transformations. Thus, if the sampling is inefficient, there is little possibility to hit a good solution. Conversely, if the sampling already offers enough chances to the obfuscator, increasing iterations displays little impact. Yet, there are also cases which achieve a high improvement with a small number of iterations. That is because those files match to a single transformation to the most. In that case, even an inadequate sampling can uncover the best. This observation opens the possibility for Closure* to infer a good *configuration* of specific files so as to enforce an efficient iteration.

E. Threats to Validity

Construct validity. This threat concerns the relation between theory and experimental observations. In our case, we focus on

optimizing the obfuscation to improve the obscurity. To measure obscurity, we turn to calculate *perplexity*, count the number of recovered variables and efforts cost in manual attack. The most possible threat is that the measurement cannot truly capture obscurity. As a response, we clarify that quantifying obscurity is not our goal, instead we try to identify good obfuscations from poor ones. From this point, our measurement is useful, which is confirmed in the evaluation where the guided optimization weakened the adversary and deterred practical attacks. On the other side, we also agree on the significance of more advanced connections between obscurity and program information, to which the proposed framework can flexibly interface.

External validity. This threat focuses on to what degree our approach can generalize to applications outside the scope in this paper. There indeed exists the potential threat that the proposed obfuscation framework may not adapt to most or all types of programs, since our instantiation is for JavaScript and experimented on open-source projects. However, the threat is mitigated by the fact that (1) our framework is not language dependent and can be instantiated in other language domains, and (2) open-source projects share a considerable group of common elements with other non-public ones. Thus, we believe the found trend should be general.

VII. Related Work

In this section, we discuss on the related works. To optimize program obfuscation, we take a novel position to effectively employing well-designed transformations. The key idea is a combination of stochastic search techniques and language model of source code.

Code Obfuscation. Program obfuscation has been extensively studied to make reverse engineering or human understanding harder [26]. Theoretically, it is claimed that no "perfect" obfuscation exists [27], [28]. Despite of the impossibility, Barak suggested *indistinguishability obfuscation* [27], [29]. Garg and Brakerski further presented obfuscators for polynomial-size circuits [30], [31]. Barak described a simplified variant to achieve protection against algebra attacks [32]. Sahai proposed *punctured programs* for cryptographic problems [33]. Goldwasser proposed to identify the *best possible* obfuscation which leaks as little information as other programs with the same functionality [34]. Other works were advanced in pursuit of better efficiency [4], [35]. For practical software use, Collberg proposed the *opaque predicate* to obfuscate the program by inserting boolean valued expressions whose values are known to obfuscators but difficult to analyze for automatic tools [1]. Sharif employed the *conditional code obfuscation* at compilation phase to transform input dependent branch conditions and encrypt the body [2]. Regarding program slicing [36] as the adversary, Drape transformed code so that the *orphaned slices* — code left after the slicing — are minimized [37]. Differently, Linn presented a complement to thwart disassembling process which translates machine code to assembly code [6]. Considering semantics, researchers viewed program analyses as adversaries and identify a set of transformations to make the analyses as much imprecise as possible. Regarding abstract interpretation [38], Giacobazzi leveraged interpreter distortion to generate obfuscated code [5] with the notion of *incompleteness* [39]. On the other side, Preda investigated the concrete program semantics instead of abstract semantics to guide the obfuscation process [3].

Compared to previous works, we focus on the emerging learning-based adversaries. Moreover, we proposed to search for effective obfuscation transformations for a given program, which results in an input sensitive optimization.

Languagem Models of Source Code. Due to the fact that software is repetitive and not unique [40], language models can be built for source code to capture regularities. Based on the classical *n-gram* model, Hindle exploited the *naturalness* of software, which proved code to be predictable and led to a programming suggestion engine [12]. Nguyen extended the model with *sememes* to involve semantic information other than lexemes [14]. Moreover, the *localness* is further enriched by Tu via proposing a *cache language model* to absorb local constructs for predicting programs [15]. In terms of method sequence, Raychev built the language models on call sequences [13]. The language model can synthesize method calls based on context and fill program holes across various objects with arguments. Towards the application of suggesting names, Allamanis [41] and Raychev [7] addressed the naming for variables while Allamanis handled methods and classes as well [8].

In our setting, we built an obscurity language model to capture the remaining regularities of obfuscated programs. To this end, we built the language model with obemes, which is different from *lexemes* based techniques.

VIII. Conclusion

Verifiability. The replication package of Closure* is at `https://bitbucket.org/njaliu/closure-star-tool`.

Conclusion. In this paper, we have proposed a novel language model based obfuscation framework. Two key insights behind the framework are: (1) We have built a language model for obfuscated programs and validated that *perplexity* helps capture obscurity, and (2) we employ stochastic search like MCMC to effectively identify powerful obfuscation *configurations* for diverse source programs. Generally, the framework can be regarded as a dynamic, guided combination of existing obfuscation techniques and techniques from the NLP community. We have realized the framework as Closure* for JavaScript programs. Evaluated on top active GitHub projects, Closure* is shown to outperform state-of-the-art obfuscators and support diverse programs. The achieved optimization can help deter advanced practical attacks. We believe that the presented framework highlights a new perspective on program obfuscation and complements existing work.

Acknowledgment

We thank the anonymous reviewers for the constructive comments. This research is sponsored by NSFC Program (No.91218 302, No.61527812), National Science and Technology Major Project (N0.16ZX010 38101), MIIT IT funds (Research and application of TCN key technologies) of China, and National Key Technology R&D Program (No.2015BAG14B01-02).

REFERENCES

[1] 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*, ser. POPL '98, 1998, pp. 184–196.

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

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

[4] P. Ananth, D. Gupta, Y. Ishai, and A. Sahai, "Optimizing obfuscation: Avoiding barrington's theorem," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 646–658.

[5] 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*, ser. PEPM '12, 2012, pp. 63–72.

[6] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03, 2003, pp. 290–299.

[7] 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*, ser. POPL '15, 2015, pp. 111–124.

[8] 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*, ser. ESEC/FSE 2015, 2015, pp. 38–49.

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

[10] C. Andrieu, N. de Freitas, A. Doucet, and M. Jordan, "An introduction to mcmc for machine learning," *Machine Learning*, vol. 50, no. 1-2, pp. 5–43, 2003.

[11] T. László and Á. Kiss, "Obfuscating c++ programs via control flow flattening," *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, pp. 3–19, 2009.

[12] 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*, ser. ICSE '12, 2012, pp. 837–847.

[13] 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*, ser. PLDI '14, 2014, pp. 419–428.

[14] 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*, ser. ESEC/FSE 2013, 2013, pp. 532–542.

[15] 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*, ser. FSE 2014, 2014, pp. 269–280.

[16] C. D. Manning and H. Schütze, *Foundations of statistical natural language processing*. MIT Press, 1999, vol. 999.

[17] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013, pp. 305–316.

[18] "KenLM," https://github.com/kpu/kenlm.

[19] "UnuglifyJS," https://github.com/eth-srl/UnuglifyJS.

[20] "Nice2Predict," https://github.com/eth-srl/Nice2Predict.

[21] "angular.js," https://github.com/angular/angular.

[22] "meteor," https://github.com/meteor/meteor.

[23] "react," https://github.com/facebook/react.

[24] S. Siegel and N. J. J. Castellan, *Nonparametric statistics for the behavioral sciences*. New York, St. Louis: McGraw-Hill, 1988, includes indexes.

[25] R. Lowry, "Concepts and applications of inferential statistics," http://vassarstats.net/textbook/ch12a.html.

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

[27] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *CRYPTO 2001*, ser. Lecture Notes in Computer Science, J. Kilian, Ed. Springer Berlin Heidelberg, 2001, vol. 2139, pp. 1–18.

[28] 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*, Oct 2005, pp. 553–562.

[29] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," *Journal of the ACM (JACM)*, vol. 59, no. 2, p. 6, 2012.

[30] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," in *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*. IEEE, 2013, pp. 40–49.

[31] Z. Brakerski and G. N. Rothblum, "Virtual black-box obfuscation for all circuits via generic graded encoding," in *Theory of Cryptography*. Springer, 2014, pp. 1–25.

[32] B. Barak, S. Garg, Y. T. Kalai, O. Paneth, and A. Sahai, "Protecting obfuscation against algebraic attacks," in *Advances in Cryptology–EUROCRYPT 2014*. Springer, 2014, pp. 221–238.

[33] A. Sahai and B. Waters, "How to use indistinguishability obfuscation: deniable encryption, and more," in *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*. ACM, 2014, pp. 475–484.

[34] S. Goldwasser and G. N. Rothblum, "On best-possible obfuscation," *Theory of Cryptography*, pp. 194–213, 2007.

[35] J. Zimmerman, "How to obfuscate programs directly," in *Advances in Cryptology-EUROCRYPT 2015*. Springer, 2015, pp. 439–467.

[36] W. Mark, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81, 1981, pp. 439–449.

[37] 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*, 2007, pp. 1019–1024.

[38] 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*, ser. POPL '77, 1977, pp. 238–252.

[39] R. Giacobazzi and I. Mastroeni, "Making abstract interpretation incomplete: Modeling the potency of obfuscation," in *Static Analysis*, vol. 7460. Springer Berlin Heidelberg, 2012, pp. 129–145.

[40] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10, 2010, pp. 147–156.

[41] 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*, ser. FSE 2014, 2014, pp. 281–293.