Nice2Predict and JSNice

Presented by Kyra Busser, kfl2120
Predicting Program Properties from “Big Code”

Veselin Raychev

Martin Vechev

Andreas Krause

Department of Computer Science

ETH Zürich

POPL '15: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages

Communications of the ACM, March 2019, Vol. 62 No. 3
Background
Deobfuscating JavaScript

Variable renaming
Minimize file size
Obscure program intent

Type annotation
Types specified in JSDoc comments
Compare Closure compiler, Flow, TypeScript, etc.

Syntactic
Semantic
Obfuscated

Compilation was a success!

Original Size: 247 bytes gzipped (445 bytes uncompressed)
Compiled Size: 122 bytes gzipped (128 bytes uncompressed)
Saved 50.61% off the gzipped size (71.24% without gzip)
The code may also be accessed at default.js.

window.a=function(c,d){for(var e=[],f=c.length,b=0;b<f;b+=d)b+d<f?e.push(c.substring(b,b+d)):e.push(c.substring(b,f));return e};
''use strict';
/**
 * @param {string} bin
 * @param {number} size
 * @return {?}
 */
window.chunkData = function chunkData(bin, size) {
  /** @type {!Array} */
  var results = [];
  var length = bin.length;
  /** @type {number} */
  var i = 0;
  for (; i < length; i = i + size) {
    if (i + size < length) {
      results.push(bin.substring(i, i + size));
    } else {
      results.push(bin.substring(i, length));
    }
  }
  return results;
};
Approach
Structured Prediction for Programs
Dependency network

Input program → Dependency network relating unknown with known properties → Predicted properties → Output program

Training data → Learned CRF model → Learning
function chunkData(e, t) {
    var n = [];
    var r = e.length;
    var i = 0;
    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;
}
Extract properties and build dependency network
Extract properties
# Extracting Names

<table>
<thead>
<tr>
<th>Known</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>Local variables</td>
</tr>
<tr>
<td>Object Properties</td>
<td>Different scopes -&gt; different properties</td>
</tr>
<tr>
<td>Methods and API Calls</td>
<td>Keywords and naming conflicts not allowed in the prediction space $\Omega_x$</td>
</tr>
<tr>
<td>Global Variables</td>
<td></td>
</tr>
<tr>
<td>Mostly treated as string constants</td>
<td></td>
</tr>
</tbody>
</table>
Extracting Types

**Known**
Any expression with known type or any constant
Manually provided or built with program analysis

**Unknown**
Variables with unknown type
\( \Omega_x = (\text{JSTypes})^n \): no constraints on predictions.

\[
\text{expr ::= val | var | expr_1(expr_2) | expr_1 \circ expr_2} \quad \text{Expression}
\]
\[
\text{val ::= } \lambda \text{var : } \tau. \text{expr} \mid n \quad \text{Value}
\]
**Type Lattice**

\[ \top - \text{Any type} \]

- string
- number
- boolean
- Function
- Array
- \( \bot - \text{No type} \)
- Other objects: e.g. RegExp, Element, Event, etc.

**JSTypes** is the powerset of all types over this lattice
Build dependency graph
i + j < k
Grammar

\[\text{rel}_{\text{ast}} ::= \text{rel}_L(\text{rel}_R) \mid \text{rel}_L \odot \text{rel}_R\]

\[\text{rel}_L ::= \text{L} \mid \text{rel}_L(\_ \_ ) \mid \_ (\text{rel}_L) \mid \text{rel}_L \odot \_ \_ \mid \_ \_ \odot \text{rel}_L\]

\[\text{rel}_R ::= \text{R} \mid \text{rel}_R(\_ \_ ) \mid \_ (\text{rel}_R) \mid \text{rel}_R \odot \_ \_ \mid \_ \_ \odot \text{rel}_R\]
Additional relations

ARG_TO_PM
Relates arguments of a function invocation to parameters in the function declaration.

ALIAS
Types only. If a and b are related with r, and c is a variable that aliases b, we add the edge (a, c, (r, ALIAS))

MAY_CALL
Names only. If a function variable f may call function g, we add the edge (f, g, MAY_CALL)

MAY_ACCESS
Names only. If in a function variable f, there is an access to an object field named foo, we add the edge (f, foo, MAY_ACCESS)
Learning
Discriminative not Generative

Since predictions are all made given a specific observed program $x$, we are only concerned with the conditional probability $Pr(y \mid x)$, not the joint probability $Pr(y, x)$.

This means we don’t need to make any assumptions about the prior probabilities of the observed properties.
CRF
CRF

Conditional Random Field

A model for the conditional probability of labels $y$ given observations $x$

$$P_r(y \mid x) = \frac{1}{Z(x)} \exp(score(y, x))$$

$Z(x)$ is just a normalization factor:

$$Z(x) = \sum_{y \in \Omega} \exp(score(y', x))$$
score

Sum of feature functions $f$ associated with weights $w$

$$score(y, x) = \sum_{i=1}^{k} w_i f_i(y, x) = w^T f(y, x)$$
Feature Functions

Can be anything that controls the likelihood of a given assignment $y$. In practice, this paper just uses the sum of pairwise indicator feature functions over the edges of the dependence network

$$f_i(y, x) = \sum_{(a, b, rel) \in E^x} \psi_i((y, z)_a, (y, z)_b, rel)$$

$z =$ assignments for known properties
Pairwise Indicator Feature Functions

Independent of the program being queried; defined once for all predictions of a given type (variable names or types).

Preprocessed from all features in the training set (plus an extra feature for equality).

Predicted outputs will be chosen from this same set of possible features.

\[ \forall (l_1^i, l_2^i, rel_i) \in \text{all\_features}(D): \]

\[ \psi_i(l_1^i, l_2^i, rel) = \begin{cases} 
1 & \text{if } l_1^i = l_1^i \text{ and } l_2^i = l_2^i \text{ and } rel = rel_i \\
0 & \text{otherwise} 
\end{cases} \]
Substituting allows us to simplify the score function. Since \( Z(x) \) does not depend on \( y' \), this has the same maximum as the full CRF probability.

\[
\text{score}(y, x) = \sum_{(a,b,rel) \in E^x} \sum_{i=1}^k w_i \psi_i((y, z)_a, (y, z)_b, rel)
\]
Learning w
Structured Support Vector Machine (SSVM)*

Generalization of classical SVMs to predict many interdependent labels at once.

Maximize $\Delta$, the margin between $y^{(j)}$ and every other $y'$

$$\forall j, \forall y' \in \Omega_{x^{(j)}} \quad \text{score}(y^{(j)}, x^{(j)}) \geq \text{score}(y', x^{(j)}) + \Delta(y^{(j)}, y')$$

Structured Hinge Loss

\[ w^* = \arg \min_w \sum_{j=1}^{t} \ell(w; x^{(j)}, y^{(j)}) \text{ s.t. } w \in \mathcal{W}_\lambda \quad (1) \]

where

\[ \ell(w; x^{(j)}, y^{(j)}) = \max_{y' \in \Omega_{x^{(j)}}} w^T [f(y', x^{(j)}) - f(y^{(j)}, x^{(j)})] + \Delta(y^{(j)}, y') \]
Stochastic Gradient Descent*

On every iteration:

- Pick a random program from the training set
- Compute the gradient of the loss function for this program
- Take a step in the negative gradient direction, with step size determined by the learning rate $\alpha$
- Project back to the feasible region

Computing the gradient

\[ y_{\text{best}} \leftarrow \arg\max_{y' \in \Omega_{x^{(j)}}} \left( \text{score}(y', x^{(j)}) + \Delta(y^{(j)}, y') \right) \], \quad (2) \]

resulting in the gradient \( g \)

\[ g \leftarrow f(y_{\text{best}}, x^{(j)}) - f(y^{(j)}, x^{(j)}) \]
Regularization

Enforces non-negativity

Minimizes overfitting

Operates on vector components independently

\( \lambda = 2.0 \) for names, 5.0 for types

\[
\text{Proj}_{W_\lambda}(w) = w' \quad \text{such that} \quad w'_i = \max(0, \min(1/\lambda, w_i))
\]
Training

Initialize: $w_i = 1/2\lambda$, $\alpha = 0.1$

If the number of wrong labels does not decrease, halve $\alpha$

Up to 24 iterations over the data

Parallelized across multiple threads*

Training runtime

32-core machine with four 2.13GHz Xeon processors, running Ubuntu 12.04 with 64-Bit OpenJDK Java 1.7.0_51.

- Training for name prediction: ~10 hours.
  - 57 minutes to compile the input code and generate networks for the input programs
  - 23 minutes per SSVM (sub-) gradient descent optimization pass

- Training for type prediction
  - 57 minutes for compilation and network construction
  - 2 minutes and 16 seconds per SSVM (sub-) gradient descent optimization pass
<table>
<thead>
<tr>
<th></th>
<th>Names</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Features</td>
<td>7,627,484</td>
<td>70,052</td>
</tr>
<tr>
<td>Size</td>
<td>145.5MB</td>
<td>1.3MB</td>
</tr>
<tr>
<td>Dictionary of all names and types</td>
<td>16.8MB</td>
<td></td>
</tr>
</tbody>
</table>
MAP inference
Prediction

\[ y = \arg \max_{y' \in \Omega_x} P_r(y' | x) = \arg \max_{y' \in \Omega_x} \text{score}(y', x) = \arg \max_{y' \in \Omega_x} w^T f(y', x) \]
Algorithm 1: Greedy Inference Algorithm

Input: network $G^x = \langle V^x, E^x \rangle$ of program $x$,
initial assignment of $n$ unknown properties $y_0 \in \Omega_x$,
known properties $z$
pairwise feature functions $\psi_i$ and their learned weights $w_i$

Output: $y \approx \arg\max_{y' \in \Omega_x} \left( \text{score}(y', x) \right)$

1 begin
2 $Y \leftarrow Y_0$
3 for $pass \in [1..\text{num\_passes}]$ do
4     // for each node with unknown property in the graph $G^x$
5     for $v \in [1..n]$ do
6         $E_v \leftarrow \{(v, -), (\_, -)\} \in E^x \cup \{(-, v, \_) \in E^x\}$
7         score$_v \leftarrow \text{scoreEdges}(E_v, (y, z))$
8         for $l' \in \text{candidates}(v, (y, z), E_v)$ do
9             $l \leftarrow y_v$  // get current label of $v$
10            $y_v \leftarrow l'$  // change label of $v$ in $y$
11            score$_{v}' \leftarrow \text{scoreEdges}(E_v, (y, z))$
12            if $y \in \Omega_x \land \text{score}_v' > \text{score}_v$ then
13                score$_v \leftarrow \text{score}_v'$
14            else
15                $y_v \leftarrow l$  // no score improvement: revert label.
16 return $y$
scoreEdges()

Score the subset of the network adjacent to the current node

\[ \text{scoreEdges}(E, A) = \sum_{(a, b, rel) \in E} \sum_{i=1}^{k} w_i \psi_i(A_a, A_b, \text{rel}) \]
candidates()

Takes the $s$ labels with the highest corresponding weights for each edge

Beam size $s$ controls precision vs. running time

$s = 64$, experimentally determined (see Results section)

Decrease the beam size by a factor of 16 if a node has more than 32 adjacent nodes
Optimize by edges (pairs) instead of single nodes

“At almost no computation cost, we also perform optimizations on pairs of nodes in addition to individual nodes. In this case, for each edge in \( G^x \), we use the \( s \) best scoring features on the same type of edge in the training set and attempt to set the labels of the two elements connected by the edge to the values in each triple.”

Not clear how this fits with the algorithm pseudocode
Results
# Datasets

<table>
<thead>
<tr>
<th>Training</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,517 JavaScript projects from GitHub</td>
<td>50 JavaScript projects with the highest number of commits from BitBucket</td>
</tr>
<tr>
<td>No overlap with eval set</td>
<td>2,710 files</td>
</tr>
<tr>
<td>324,501 files</td>
<td>381,243 LOC, Largest file 3,055</td>
</tr>
<tr>
<td>Filtered minified and obfuscated files</td>
<td>383.5 (109.5) arcs and 29.2 (12.6) random variables for names (types) on average in the eval set</td>
</tr>
</tbody>
</table>
Parameter selection

10-fold cross-validation

1% sample of the training data

\( \lambda = 2.0 \) for names, 5.0 for types

Margin \( \Delta \) should be applied
## Results

<table>
<thead>
<tr>
<th>System</th>
<th>Names Accuracy</th>
<th>Types Precision</th>
<th>Types Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>all training data</td>
<td>63.4%</td>
<td>81.6%</td>
<td>66.9%</td>
</tr>
<tr>
<td>10% of training data</td>
<td>54.5%</td>
<td>81.4%</td>
<td>64.8%</td>
</tr>
<tr>
<td>1% of training data</td>
<td>41.2%</td>
<td>77.9%</td>
<td>62.8%</td>
</tr>
<tr>
<td>all data, no structure</td>
<td>54.1%</td>
<td>84.0%</td>
<td>56.0%</td>
</tr>
<tr>
<td>baseline - no predictions</td>
<td>25.3%</td>
<td>37.8%</td>
<td>100%</td>
</tr>
</tbody>
</table>
## Beam size and prediction time

<table>
<thead>
<tr>
<th>Beam parameter</th>
<th>Name prediction</th>
<th>Type prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
<td>Precision</td>
</tr>
<tr>
<td>$b$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>57.9%</td>
<td>80.6%</td>
</tr>
<tr>
<td>8</td>
<td>59.2%</td>
<td>80.9%</td>
</tr>
<tr>
<td>16</td>
<td>62.8%</td>
<td>81.6%</td>
</tr>
<tr>
<td>32</td>
<td>63.2%</td>
<td>81.3%</td>
</tr>
<tr>
<td>64 (JSNICE)</td>
<td>63.4%</td>
<td>81.6%</td>
</tr>
<tr>
<td>128</td>
<td>63.5%</td>
<td>82.0%</td>
</tr>
<tr>
<td>256</td>
<td>63.5%</td>
<td>81.6%</td>
</tr>
<tr>
<td>Naïve greedy, no beam</td>
<td>62.8%</td>
<td>81.7%</td>
</tr>
</tbody>
</table>
Typechecking
Discussion
A note on name inference

“We note that our name inference process is independent of what the minified names are. In particular, the process will return the same names regardless of which minifier was used to obfuscate the original program (provided these minifiers always rename the same set of variables).”
However...

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

```javascript
function chunkData(xs, i) {
  var data = [];
  var end = xs.length;
  var offset = 0;
  for (; offset < end; offset += i) {
    if (offset + i < end) {
      data.push(xs.substring(offset, offset + i));
    } else {
      data.push(xs.substring(offset, end));
    }
  }
  return data;
}
```

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

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

Function names treated as givens, might be over-weighted

Different behavior between Proprietary http://jsnice.org/ and Open Source http://www.nice2predict.org/
Future Work
“Similarly to JSNice, DeGuard is based on powerful probabilistic graphical models learned from thousands of open source programs. Using these models, DeGuard recovers important information in Android APKs, including method and class names as well as third-party libraries. DeGuard can reveal string decoders and classes that handle sensitive data in Android malware.”

https://www.sri.inf.ethz.ch/deguard
“DEBIN is a novel system for predicting debug information in stripped binaries. It is able to distinguish register-allocated and memory-allocated variables with decision-tree-based classification. Moreover, it is capable of predicting meaningful names and types for variables and functions through structured prediction with probabilistic graphical models. These models are learned from thousands of non-stripped binary in open source packages. The system can be further used for malware inspection.”

https://github.com/eth-sri/debin
STEP 1: PARSING

This is the only language-specific part of our platform and enables us to add support for any custom language in a matter of weeks.

STEP 2: SOLVERS

Our custom language-independent linear-complexity catalog solvers allow us to analyze huge repositories in a matter of seconds.

STEP 3: ML ALGORITHMS

Our custom Semantic Facts representations allow us to run powerful ML algorithms to understand the structure, function, and intent of the code.

https://www.deepcode.ai/
Beyond layout obfuscation

Can you start with a control flow graph instead of an AST?

Analyze code changes*

Questions?
The End

Thank you very much!