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

Syntactic

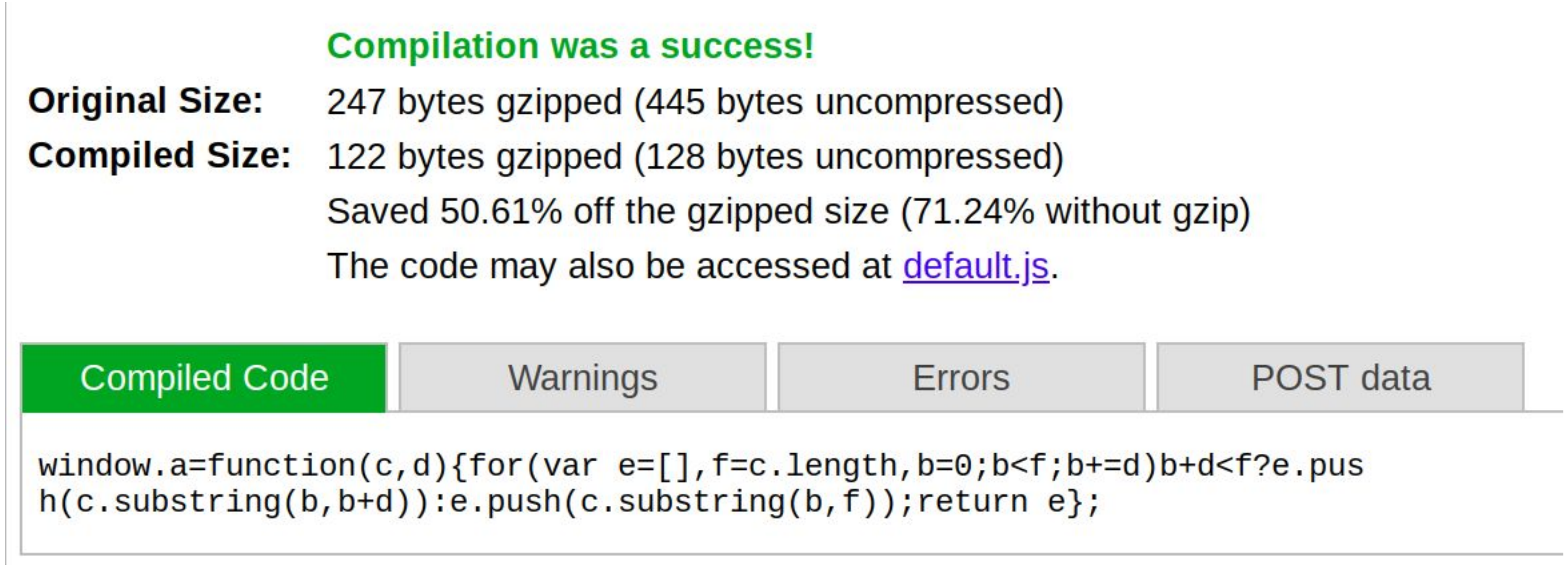
Type annotation

Types specified in JSDoc comments

Compare Closure compiler, Flow, TypeScript, etc.

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.

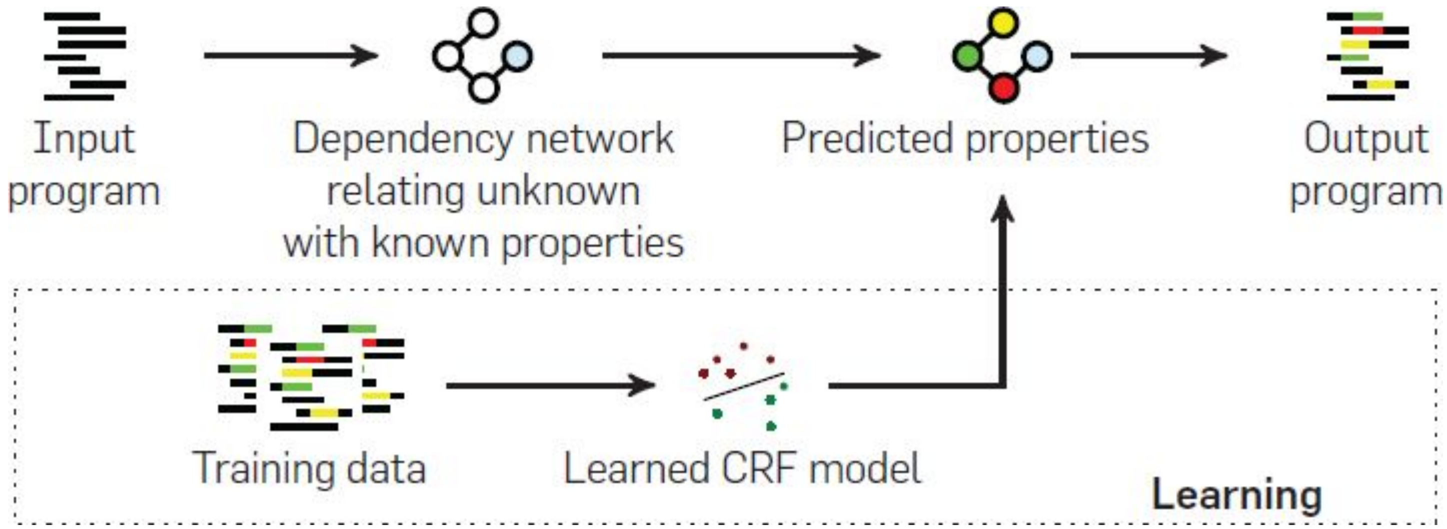
```
window.a=function(c,d){for(var e=[],f=c.length,b=0;b<f;b+=d)b+d<f?e.pus
h(c.substring(b,b+d)):e.push(c.substring(b,f));return e};
```

https://closure-compiler.appspot.com/

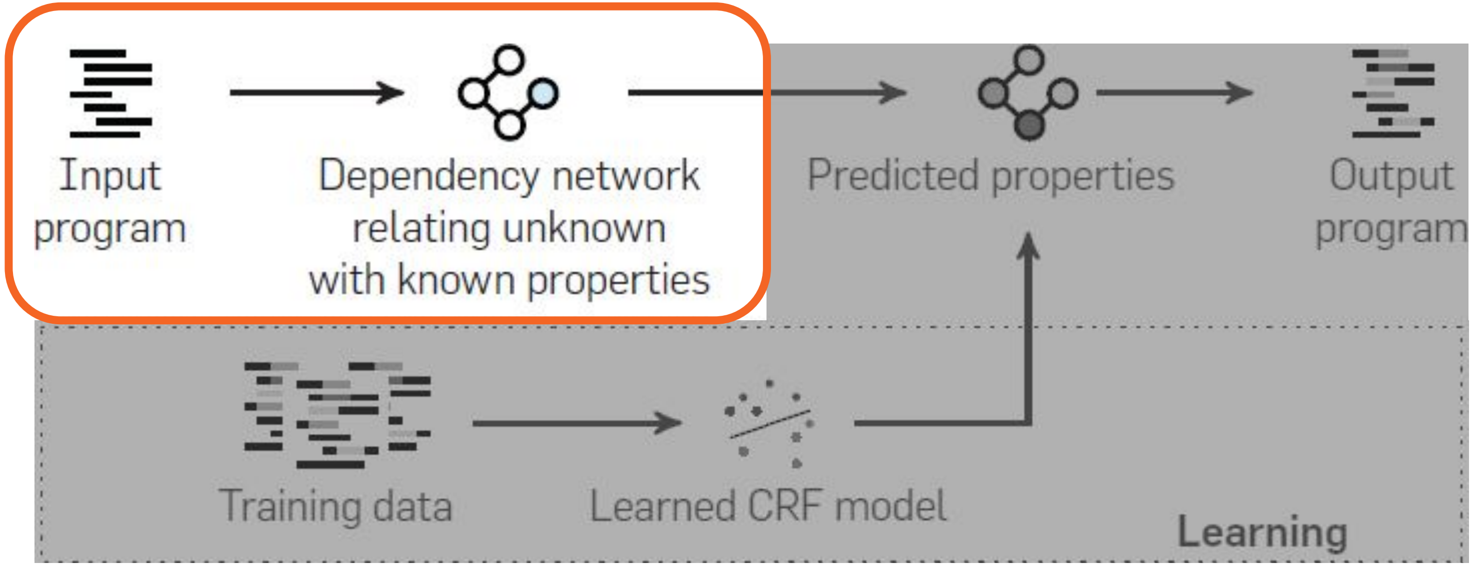
Original: commented, annotated

```
'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



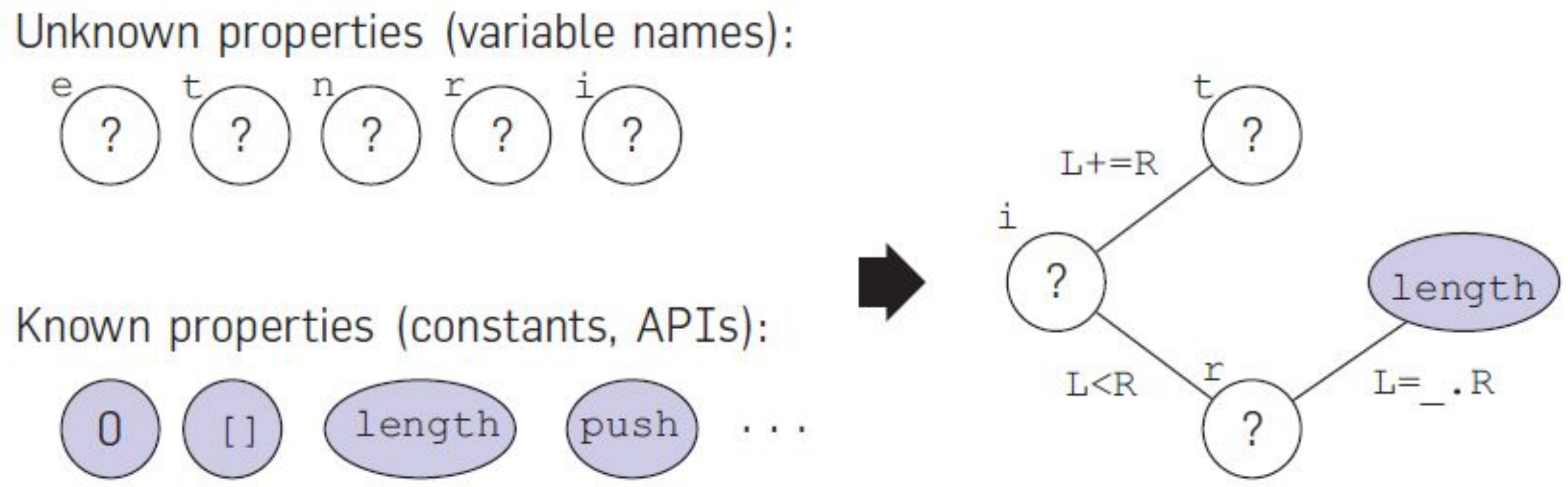
Structured Prediction for Programs



Dependency network

Input: Obfuscated Program

```
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;
}
```


Unknown properties (variable names):
e
t
n
r
i
?
Known properties (constants, APIs):
0
[]
length
push
...
t
?
L+=R
i
?
length
L<R
r
?
L=_.R

Extract properties and build dependency network

Extract properties

Extracting Names

Known

Constants

Object Properties

Methods and API Calls

Global Variables

Mostly treated as string constants

Unknown

Local variables

Different scopes -> different properties

Keywords and naming conflicts not allowed in the prediction space Ω_x

Extracting Types

Known

Any **expression** with known type or any constant

Manually provided or built with program analysis

Unknown

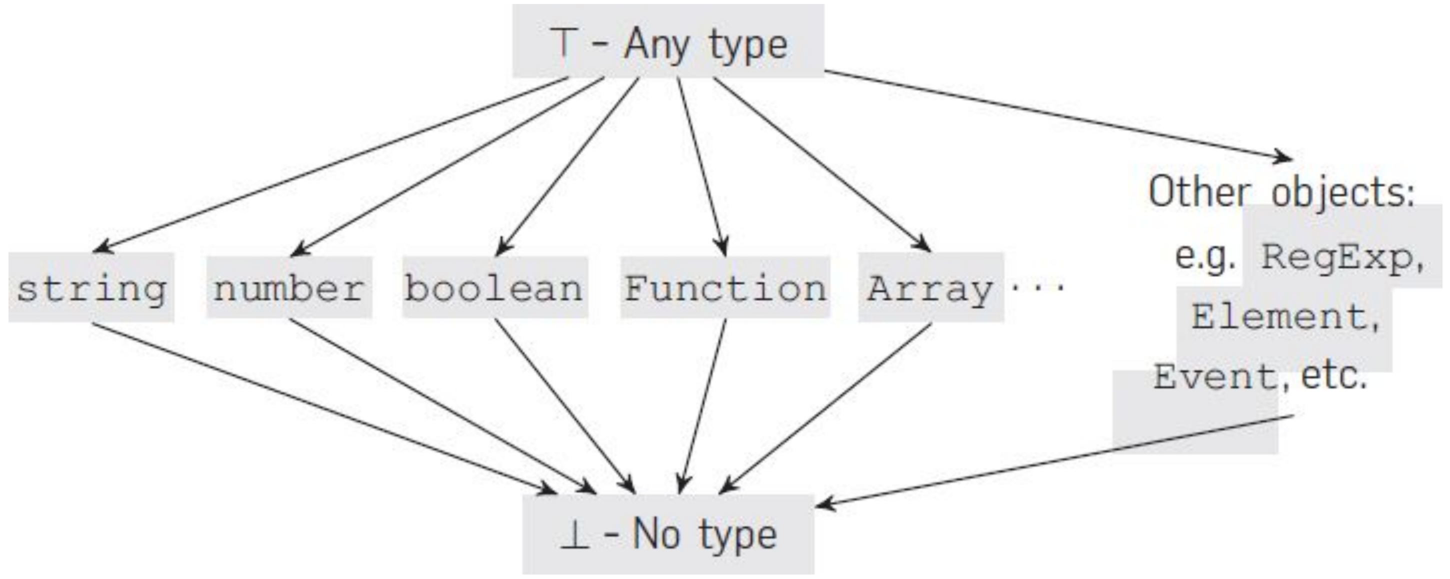
Variables with unknown type

$\Omega_x = (\mathit{JSTypes})^n$: no constraints on predictions.

$$expr ::= val \mid var \mid expr_1(expr_2) \mid expr_1 \circledast expr_2 \qquad \mathit{Expression}$$

$$val ::= \lambda var : \tau.expr \mid n \qquad \mathit{Value}$$

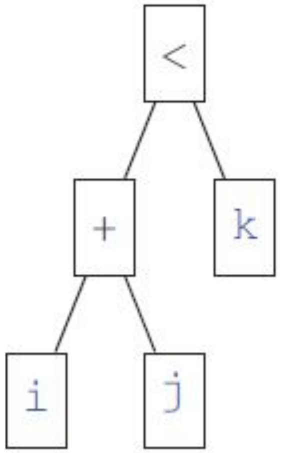
Type Lattice



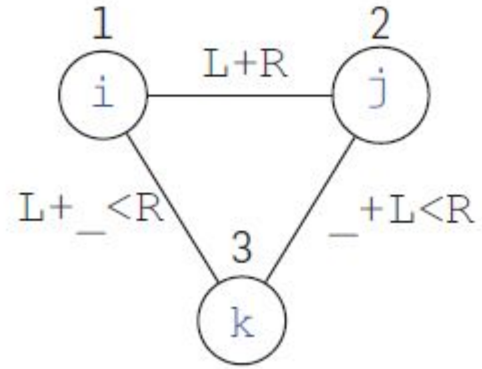
JSTypes is the powerset of all types over this lattice

Build dependency graph

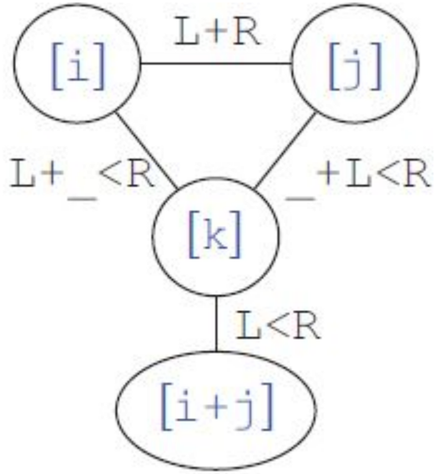
i + j < k



AST



Names



Types

Grammar

$$
\begin{aligned}
rel_{ast} &::= rel_L(rel_R) \mid rel_L \circledast rel_R \\
rel_L &::= \mathtt{\textbf{L}} \mid rel_L(\_) \mid \_(rel_L) \mid rel_L \circledast \_ \mid \_ \circledast rel_L \\
rel_R &::= \mathtt{\textbf{R}} \mid rel_R(\_) \mid \_(rel_R) \mid rel_R \circledast \_ \mid \_ \circledast rel_R
\end{aligned}
$$

Additional relations

ARG_TO_PM

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

ALIAS

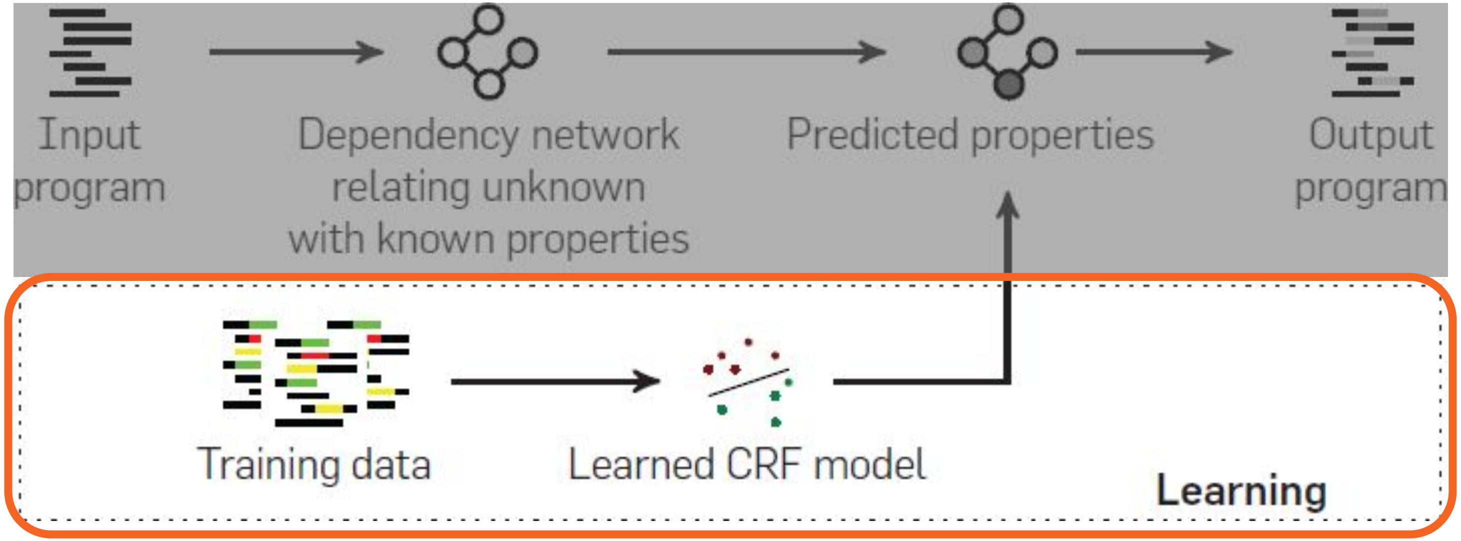
Types only. If a and b are related with r, and c is a variable that aliases b, we add the edge (a, c, (r, ALIAS))

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)


Input program
Dependency network relating unknown with known properties
Predicted properties
Output program
Training data
Learned CRF model
Learning

Learning

Discriminative not Generative

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

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*

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

Z(x) is just a normalization factor:

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

score

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

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

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

z = assignments for known properties

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

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.

$$\forall \quad \langle l_i^1, l_i^2, rel_i \rangle \in all\_features(D):$$

$$\psi_i(l^1, l^2, rel) = \begin{cases} 1 & \textbf{if } l^1 = l_i^1 \textbf{ and } l^2 = l_i^2 \textbf{ and } rel = rel_i \\ 0 & \textbf{otherwise} \end{cases}$$

score

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.

$$score(\mathbf{y}, x) = \sum_{(a,b,rel)\in E^x} \sum_{i=1}^{k} w_i \psi_i((\mathbf{y},\mathbf{z})_a, (\mathbf{y},\mathbf{z})_b, rel)$$

Learning w

Structured Support Vector Machine (SSVM)[*]

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

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

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

[*] TSOCHANTARIDIS, I., JOACHIMS, T., HOFMANN, T., AND ALTUN, Y. Large margin methods for structured and interdependent output variables. Journal of Machine Learning Research 6, 2005, 1453–1484

Structured Hinge Loss

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{j=1}^{t} \ell(\mathbf{w}; x^{(j)}, \mathbf{y}^{(j)}) \text{ s.t. } \mathbf{w} \in \mathcal{W}_\lambda \tag{1}$$

where

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

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 α
- Project back to the feasible region

[*] RATLIFF, N. D., BAGNELL, J. A., AND ZINKEVICH, M. (approximate) subgradient methods for structured prediction. In AISTATS (2007), pp. 380–387.

Computing the gradient

$$\mathbf{y}_{best} \leftarrow \operatorname*{argmax}_{\mathbf{y}' \in \Omega_{x^{(j)}}} \left(\, score(\mathbf{y}', x^{(j)}) + \Delta(\mathbf{y}^{(j)}, \mathbf{y}') \, \right), \qquad (2)$$

resulting in the gradient $\mathbf{g}$

$$\mathbf{g} \leftarrow \mathbf{f}(\mathbf{y}_{best}, x^{(j)}) - \mathbf{f}(\mathbf{y}^{(j)}, x^{(j)})$$

Regularization

Enforces non-negativity

Minimizes overfitting

Operates on vector components independently

λ = 2.0 for names, 5.0 for types

$$\mathrm{Proj}_{\mathcal{W}_\lambda}(\mathbf{w}) = \mathbf{w}' \quad \text{such that} \quad w_i' = \max(0, \min(1/\lambda, w_i))$$

Training

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

If the number of wrong labels does not decrease, halve α

Up to 24 iterations over the data

Parallelized across multiple threads*

[*] ZINKEVICH, M., WEIMER, M., LI, L., AND SMOLA, A. J. Parallelized stochastic gradient descent. In NIPS (2010), pp. 2595–2603.

Training runtime

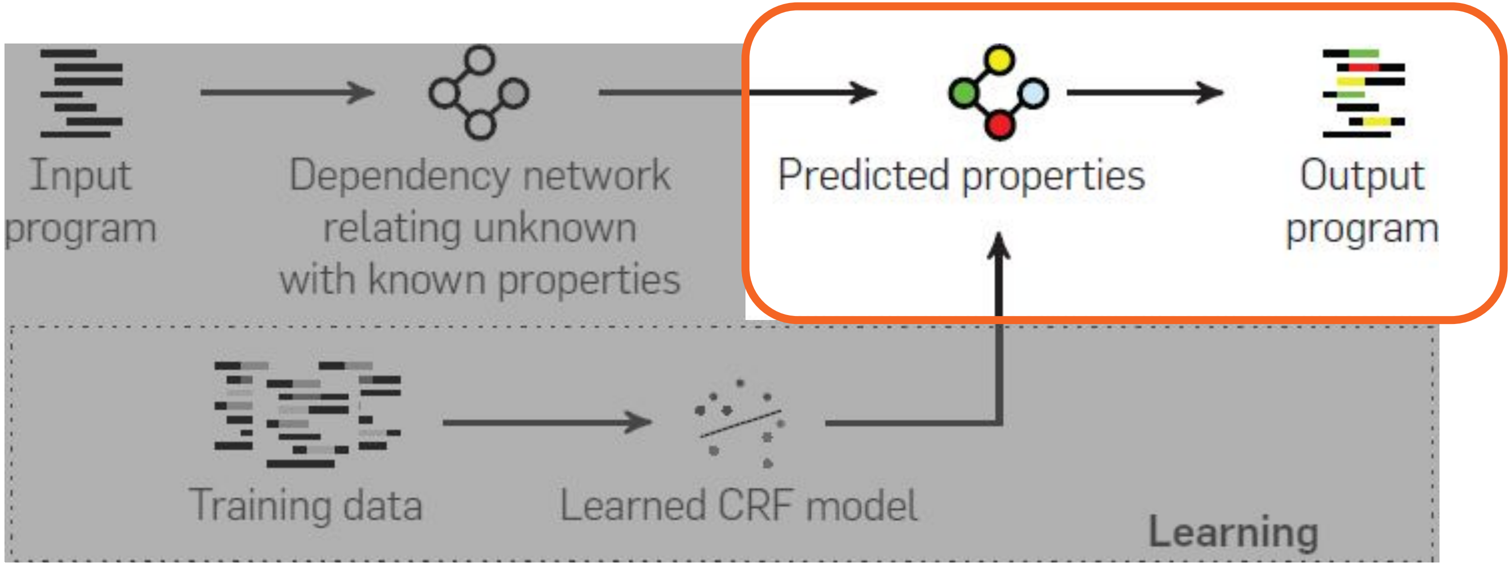
32-core machine with four 2.13GHz Xeon processors, running Ubuntu 12.04 with 64-Bit OpenJDK Java 1.7.0_51.

- 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

Model Sizes

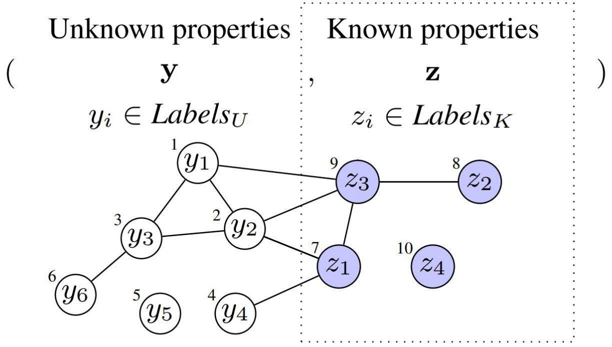
Names	**Types**
7,627,484 features	70,052 features
145.5MB	1.3MB

Dictionary of all names and types
16.8MB

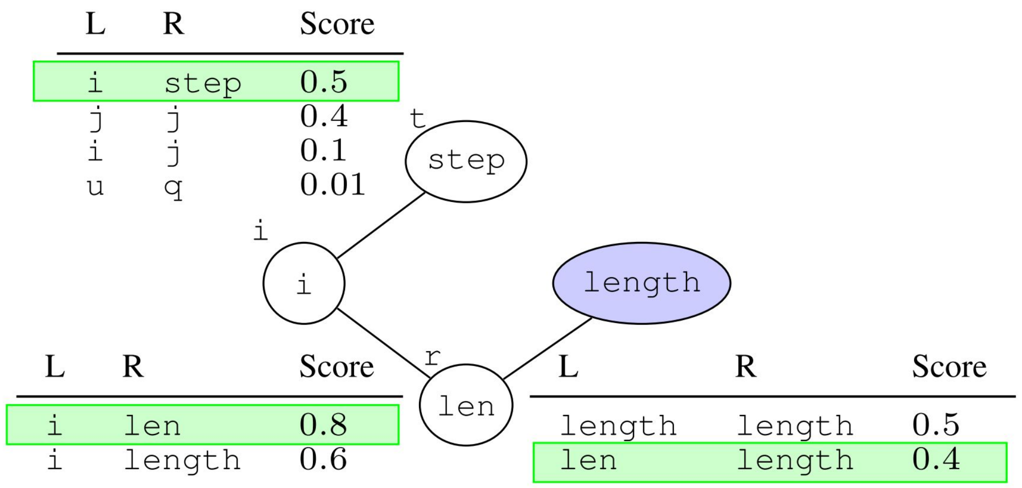


MAP inference

Prediction



Prediction: $\mathbf{y} = \text{argmax}_{\mathbf{y}' \in \Omega_x} Pr(\mathbf{y}' \mid x)$



L	R	Score
i	step	0.5
j	j	0.4
i	j	0.1
u	q	0.01

L	R	Score
i	len	0.8
i	length	0.6

L	R	Score
length	length	0.5
len	length	0.4

$$\mathbf{y} = \underset{\mathbf{y}' \in \Omega_x}{\text{argmax}}\, Pr(\mathbf{y}' | x) = \underset{\mathbf{y}' \in \Omega_x}{\text{argmax}}\, score(\mathbf{y}', x) = \underset{\mathbf{y}' \in \Omega_x}{\text{argmax}}\, \mathbf{w}^T \mathbf{f}(\mathbf{y}', x)$$

Algorithm 1: Greedy Inference Algorithm

```
Input: network $G^x = \langle V^x, E^x \rangle$ of program $x$,
       initial assignment of $n$ unknown properties $\mathbf{y}_0 \in \Omega_x$,
       known properties $\mathbf{z}$
       pairwise feature functions $\psi_i$ and their learned weights $w_i$
Output: $\mathbf{y} \approx \text{argmax}_{\mathbf{y}' \in \Omega_x} \big( score(\mathbf{y}', x) \big)$
begin
    $\mathbf{y} \leftarrow \mathbf{y}_0$
    for $pass \in [1..num\_passes]$ do
        // for each node with unknown property in the graph $G^x$
        for $v \in [1..n]$ do
            $E_v \leftarrow \{(v, \_, \_) \in E^x\} \cup \{(\_, v, \_) \in E^x\}$
            $score_v \leftarrow scoreEdges\big(E_v, (\mathbf{y}, \mathbf{z})\big)$
            for $l' \in candidates\big(v, (\mathbf{y}, \mathbf{z}), E_v\big)$ do
                $l \leftarrow y_v$    // get current label of $v$
                $y_v \leftarrow l'$    // change label of $v$ in $\mathbf{y}$
                $score'_v \leftarrow scoreEdges\big(E_v, (\mathbf{y}, \mathbf{z})\big)$
                if $\mathbf{y} \in \Omega_x \wedge score'_v > score_v$ then
                    $score_v \leftarrow score'_v$
                else
                    $y_v \leftarrow l$    // no score improvement: revert label.
    return $\mathbf{y}$
```

scoreEdges()

Score the subset of the network adjacent to the current node

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

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

Training

10,517 JavaScript projects from GitHub

No overlap with eval set

324,501 files

Filtered minified and obfuscated files

Evaluation

50 JavaScript projects with the highest number of commits from BitBucket

2,710 files

381,243 LOC, Largest file 3,055

383.5 (109.5) arcs and 29.2 (12.6) random variables for names (types) on average in the eval set

Parameter selection

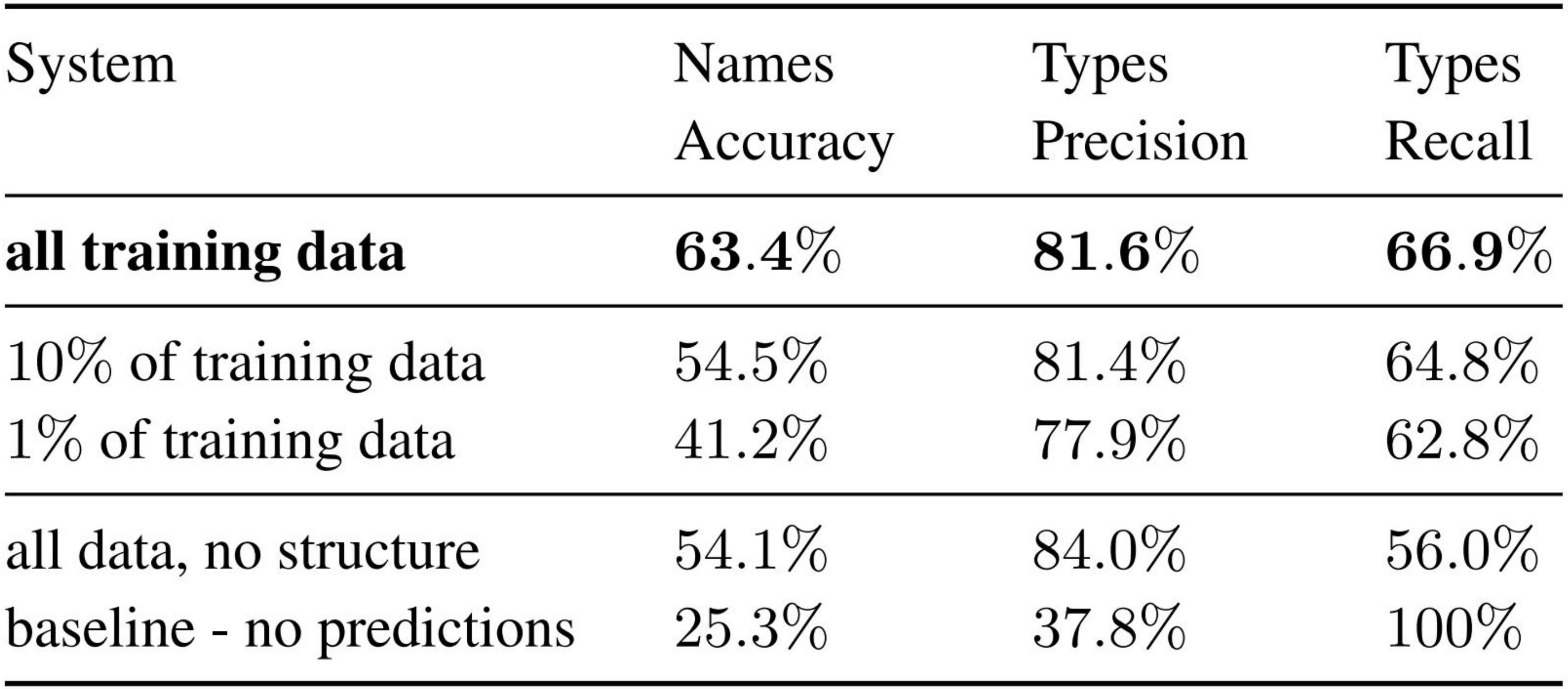
10-fold cross-validation

1% sample of the training data

$\lambda = 2.0$ for names, 5.0 for types

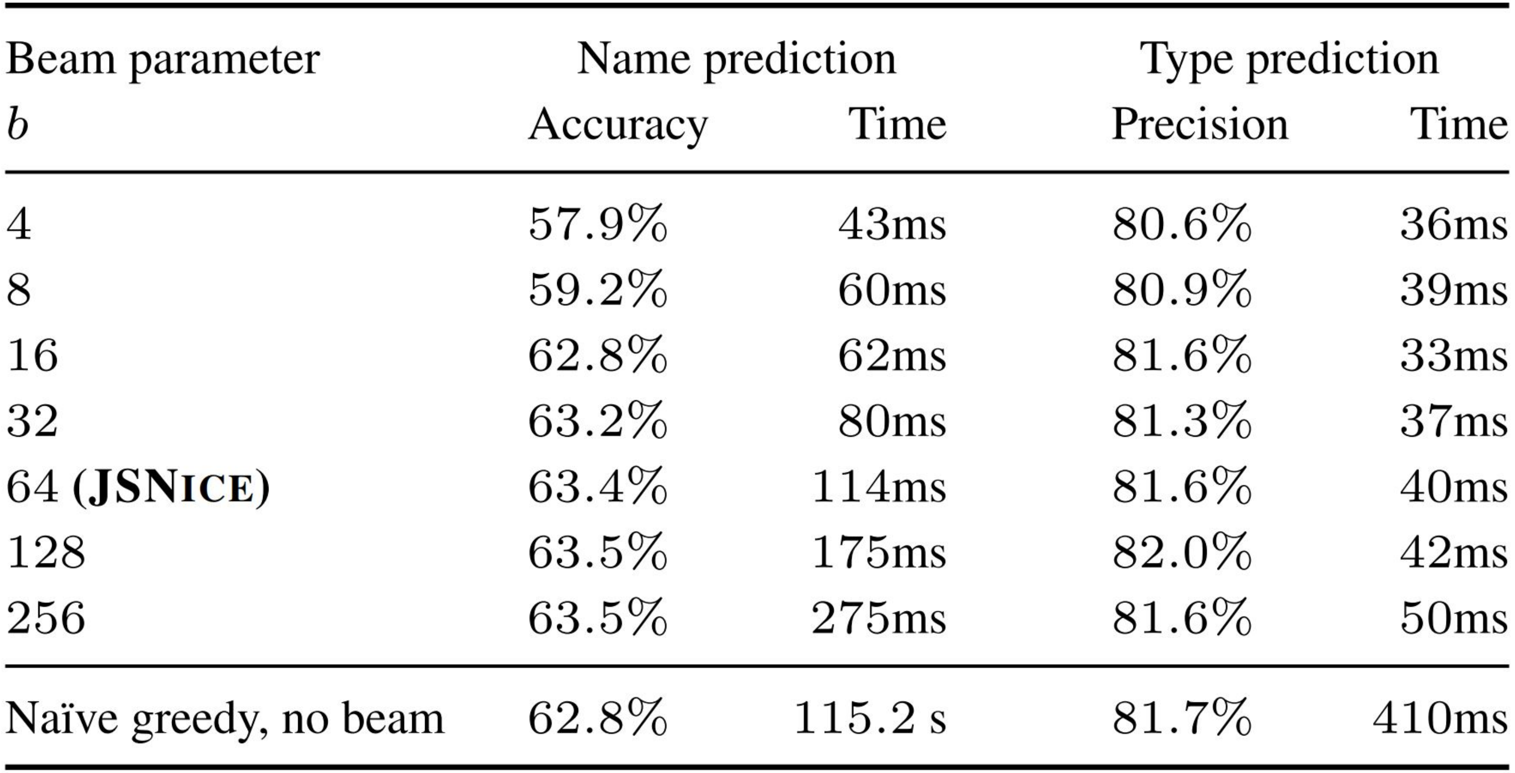
Margin Δ should be applied

Results

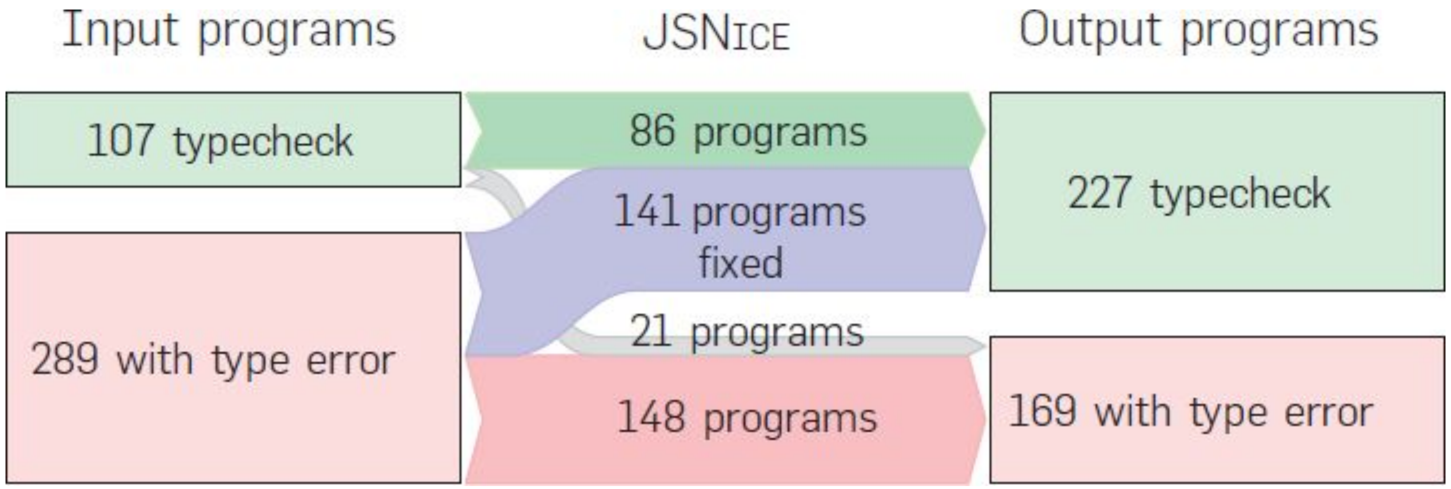


System	Names Accuracy	Types Precision	Types Recall
all training data	**63.4%**	**81.6%**	**66.9%**
10% of training data	54.5%	81.4%	64.8%
1% of training data	41.2%	77.9%	62.8%
all data, no structure	54.1%	84.0%	56.0%
baseline - no predictions	25.3%	37.8%	100%

Beam size and prediction time



Beam parameter	Name prediction		Type prediction	
b	Accuracy	Time	Precision	Time
4	57.9%	43ms	80.6%	36ms
8	59.2%	60ms	80.9%	39ms
16	62.8%	62ms	81.6%	33ms
32	63.2%	80ms	81.3%	37ms
64 (**JSNice**)	63.4%	114ms	81.6%	40ms
128	63.5%	175ms	82.0%	42ms
256	63.5%	275ms	81.6%	50ms
Naïve greedy, no beam	62.8%	115.2 s	81.7%	410ms



Typechecking

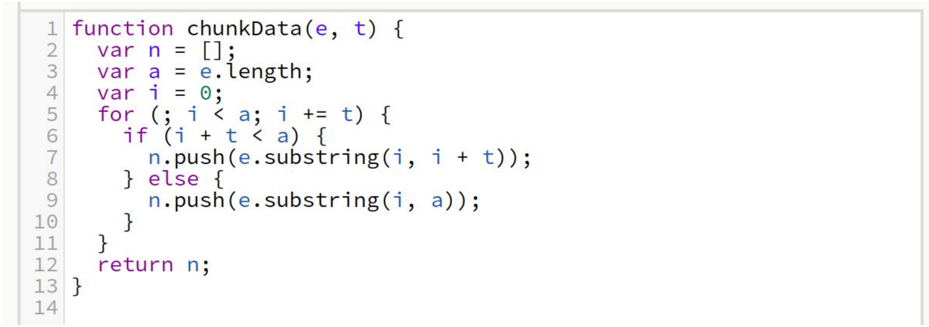
Demo!?

Discussion

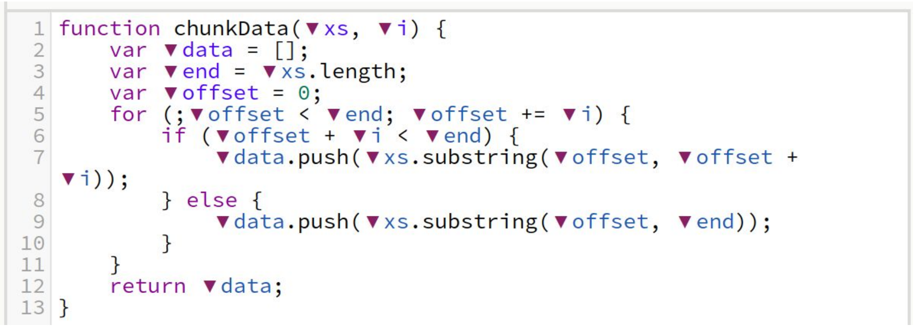
A note on name inference

“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).”

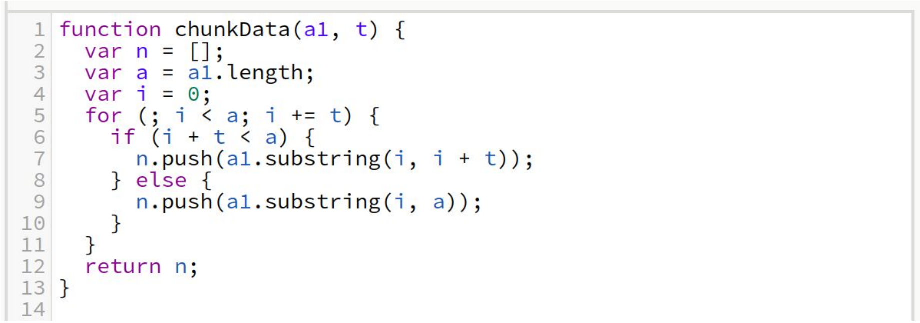
However...



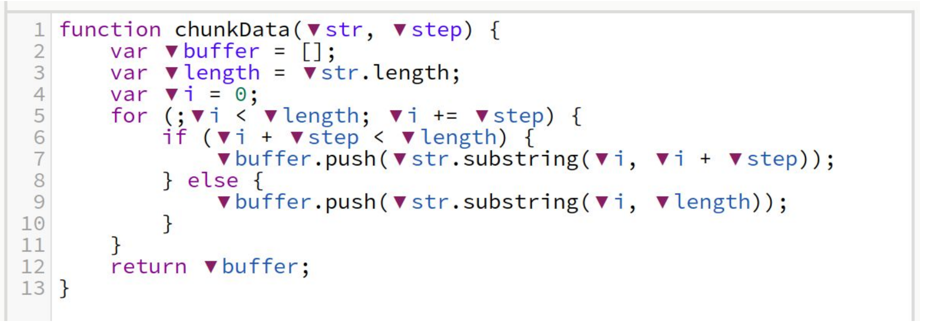
```
function chunkData(e, t) {
  var n = [];
  var a = e.length;
  var i = 0;
  for (; i < a; i += t) {
    if (i + t < a) {
      n.push(e.substring(i, i + t));
    } else {
      n.push(e.substring(i, a));
    }
  }
  return n;
}
```



```
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;
}
```



```
function chunkData(a1, t) {
  var n = [];
  var a = a1.length;
  var i = 0;
  for (; i < a; i += t) {
    if (i + t < a) {
      n.push(a1.substring(i, i + t));
    } else {
      n.push(a1.substring(i, a));
    }
  }
  return n;
}
```



```
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

http://apk-deguard.com/

“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.”

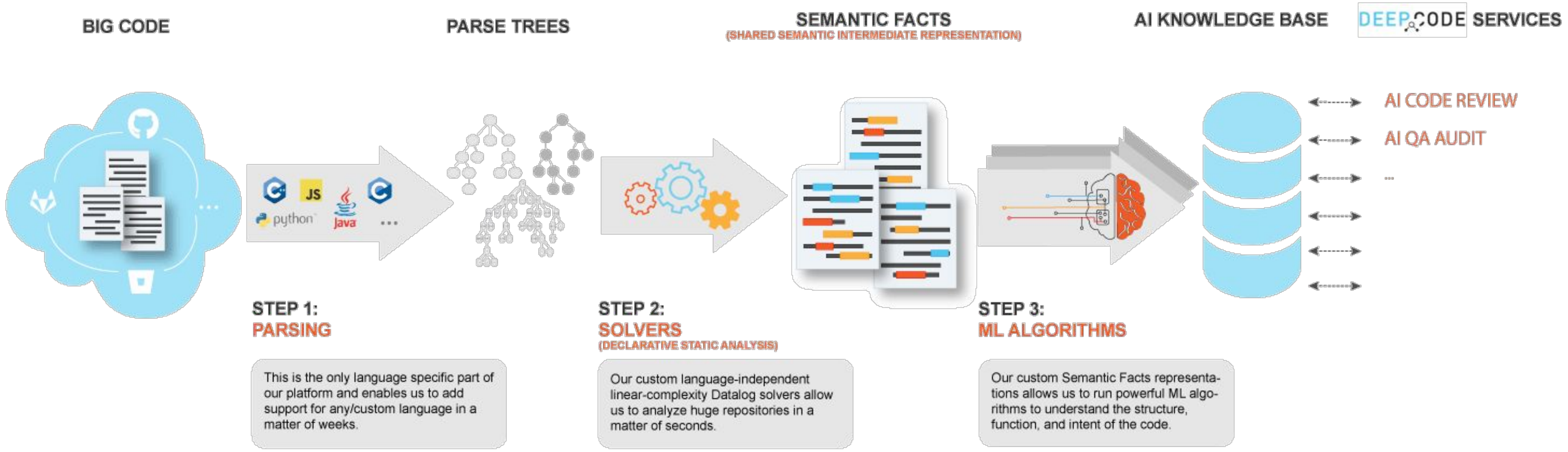
https://www.sri.inf.ethz.ch/deguard

https://debin.ai/

“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

https://www.deepcode.ai/



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.

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

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

Beyond layout obfuscation

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

Analyze code changes*

[*] Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Inferring crypto API rules from code changes. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018). ACM, New York, NY, USA, 450-464. DOI: https://doi.org/10.1145/3192366.3192403

Questions?

The End

Thank you very much!