Automating Dependability Analysis for Software Systems

John Heaps
Computer Science Department

Dependability

- Dependability in Software Engineering
  - Availability
  - Reliability
  - Safety
  - Security

- Dependability defined for Software Systems
  - Privacy Policies
  - Software Bugs

- Analysis of Dependability
  - Privacy policy compliance
  - Software bug detection

The Cost of Poor Dependability

- Financial Loss
  - In 2018 alone, software bugs cost the world economy over $1.7 trillion and impacted over 3.7 billion people

- Loss of Privacy
  - Facebook scandals have leaked private information for about 500 million users

- Loss of Life
  - The U.S. Patriot missile defense system did not detect an incoming missile due to inaccurate tracking calculation causing the loss of 28 U.S. military troops

- By verifying policy requirements and detecting bugs in code, many of the costs and damages caused by poor software Dependability can be eliminated

The Need For Automation

- The manual process to verify policy compliance and perform bug detection of code takes too long and requires too much information
  - Any system could have dozens or hundreds of privacy constraints
    - Medical systems must be in compliance with HIPAA
  - Any system could exhibit any number of different types of bugs
    - Currently, CWE defines 808 different types of bugs
    - A system could be tens of thousands to millions of lines of code in size

State of the Art: Static Analysis and Model Checking Techniques

- Model Checking
  - Construct specifications (usually in temporal logic) that define the constraints of the system
  - If any state of the system does not satisfy the specifications, it is reported as a bug

- Many of the most popular static analysis bug detection tools and techniques use Bug Patterns
  - Each pattern defines a bug or policy violation
  - Usually defined in the form of *if-then*

Model Checking Problems

- Model checking is not scalable
    - Subject to the state explosion problem

- Still need to manually generate specifications for policy and system or manually create a model of the system to be checked

Static Analysis Problems

- Static analysis requires patterns/specifications to be provided for it
  - Patterns and specifications must be manually defined for each bug that wants to be detected
  - Cannot detect bugs that do not have a pattern or specification defined for it

- Static analysis is conservative
  - Most static analysis techniques have a high false positive rate

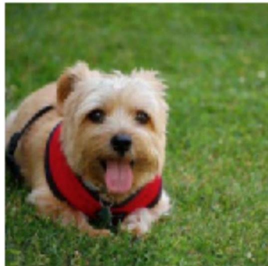
- Static analysis is not always scalable
  - Some static analysis techniques require an exploration of all possible paths/states/etc. which is subject to the state explosion problem

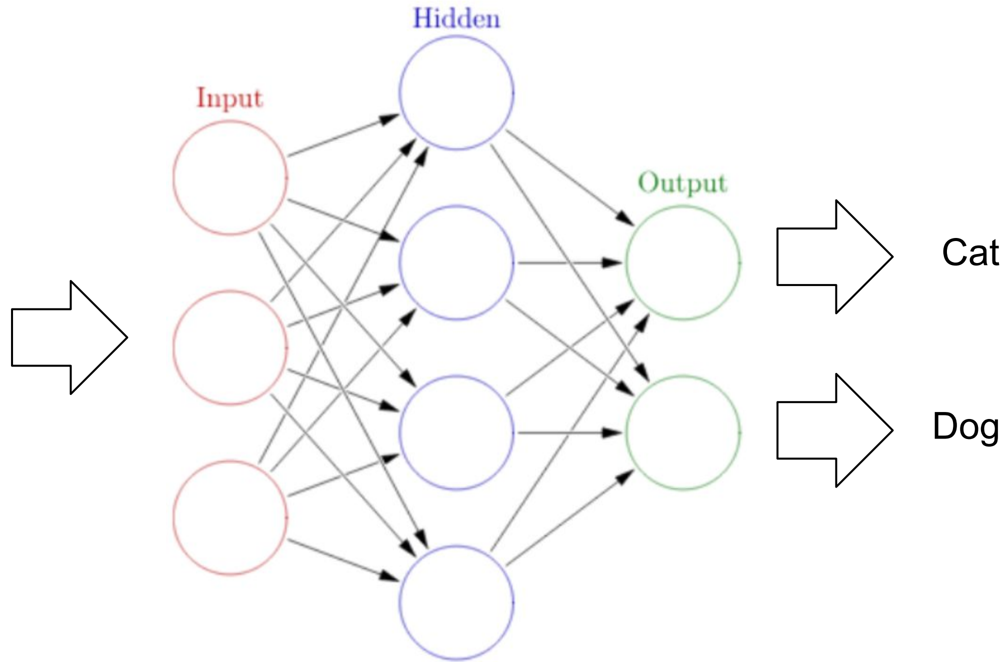
Why Perform Deep Learning

- Many of the limitations of static analysis can likely be mitigated by deep learning:
    - Will learn features of code and bugs, so no patterns or specifications need to be defined
    - There is a good chance that new or different versions of learned bugs can still be detected
    - Will not be as conservative
    - Is not subject to the state explosion problem

- We should be able to utilize many existing deep learning techniques from natural language processing (NLP)

Background: Feed-Forward Neural Networks (FFNNs)





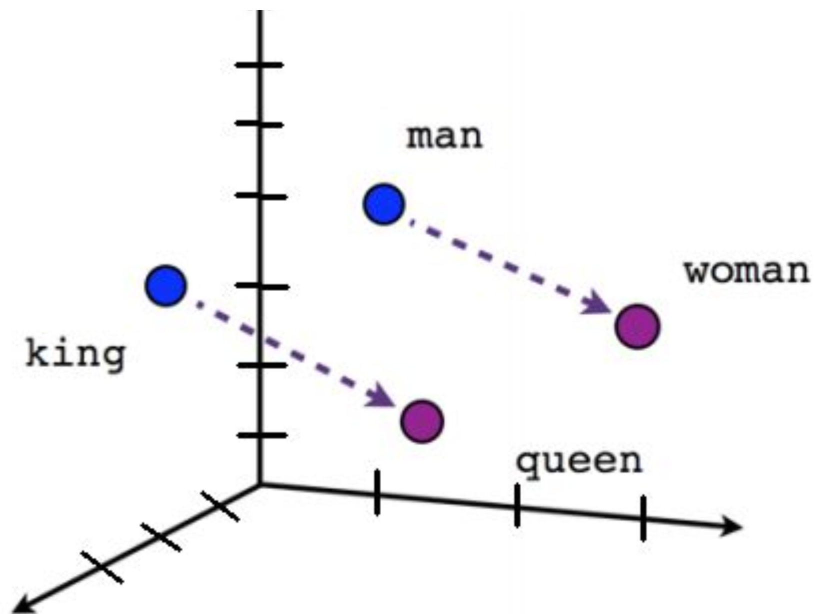
Background: Vector Representations

king = {2,0,3}

queen = {2,2,2}

man = {0,1,4}

woman = {0,3,3}



How do we learn these vector representations?

Source Text

Training Samples

The quick brown fox jumps over the lazy dog. ⟹
(the, quick)
(the, brown)

The quick brown fox jumps over the lazy dog. ⟹
(quick, the)
(quick, brown)
(quick, fox)

The quick brown fox jumps over the lazy dog. ⟹
(brown, the)
(brown, quick)
(brown, fox)
(brown, jumps)

The quick brown fox jumps over the lazy dog. ⟹
(fox, quick)
(fox, brown)
(fox, jumps)
(fox, over)

Major Obstacles for Deep Learning on Software Code

- Syntactic structures are more complex
- The data sparsity problem is more severe
- The number of new code elements encountered is usually far greater than normal NLP

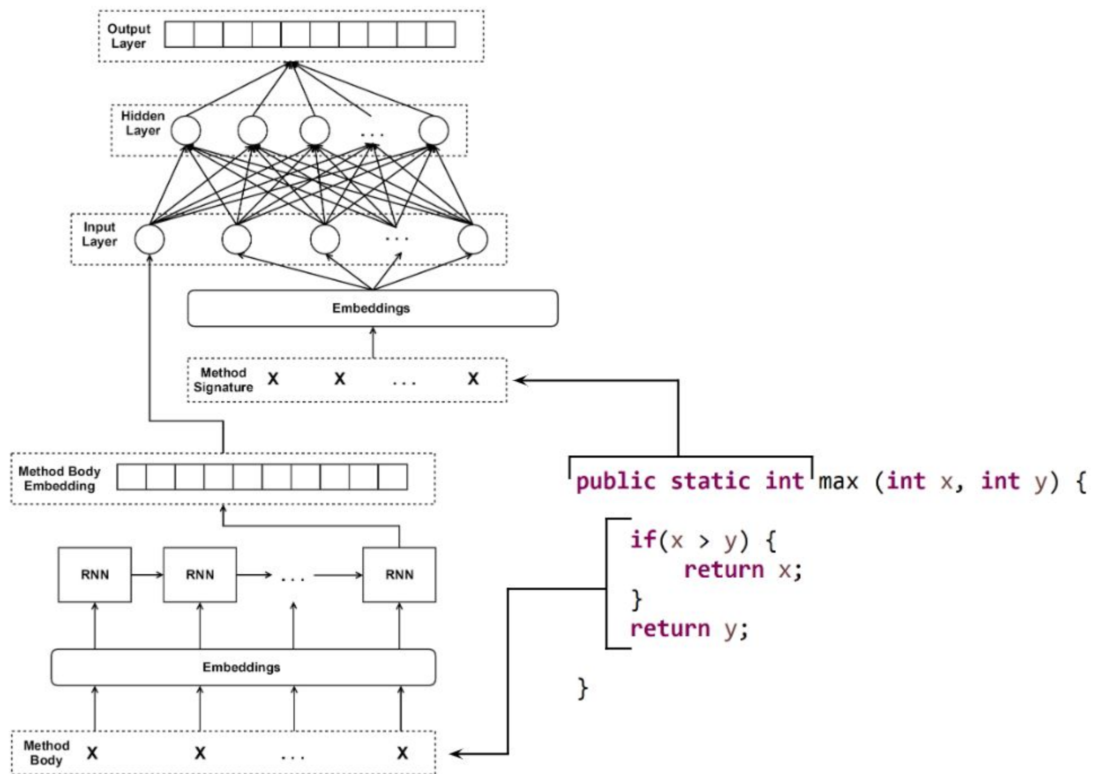
Toward a Semantic-Oriented Model

- Previous word embeddings based on definition of a language model was based on statistical frequencies of co-occurrences of code elements (a ***contextual-oriented model***)

- To learn a new ***semantic-oriented model***, a new type of relationship between words must be defined
    - An equivalence relation can be defined between one word and a sequence of words by using the dictionary

- By using this semantic model we can solve many previously mentioned limitations:
    - No longer affected by data sparsity (only need a word's definition)
    - New token can be handled easily (only a definition needs to be provided for it)

Toward a Semantic-Oriented Model



Conclusion

- Automation of dependability analysis for software systems is important to prevent loss and feasibly perform analysis at large scale

- Model checking and static analysis are state of the art, but have many limitations

- Many of these limitations may be solved through the use of deep learning

? Questions ?

Overview

- Dependability Using Model Checking and Static Analysis
- Background: Deep Learning
- Toward Deep Learning on Software Code
- Toward a Semantic-Oriented Model

Policy Verification Obstacles

Policy

- Written in natural language
- Natural language is ambiguous

Verification of Compliance

- Want a similar representation for comparison/analysis
- Want formalized specifications

Software System

- Written in code
- Code can have bugs
- Code can be written in different programming languages

Improper Authorization Code Example

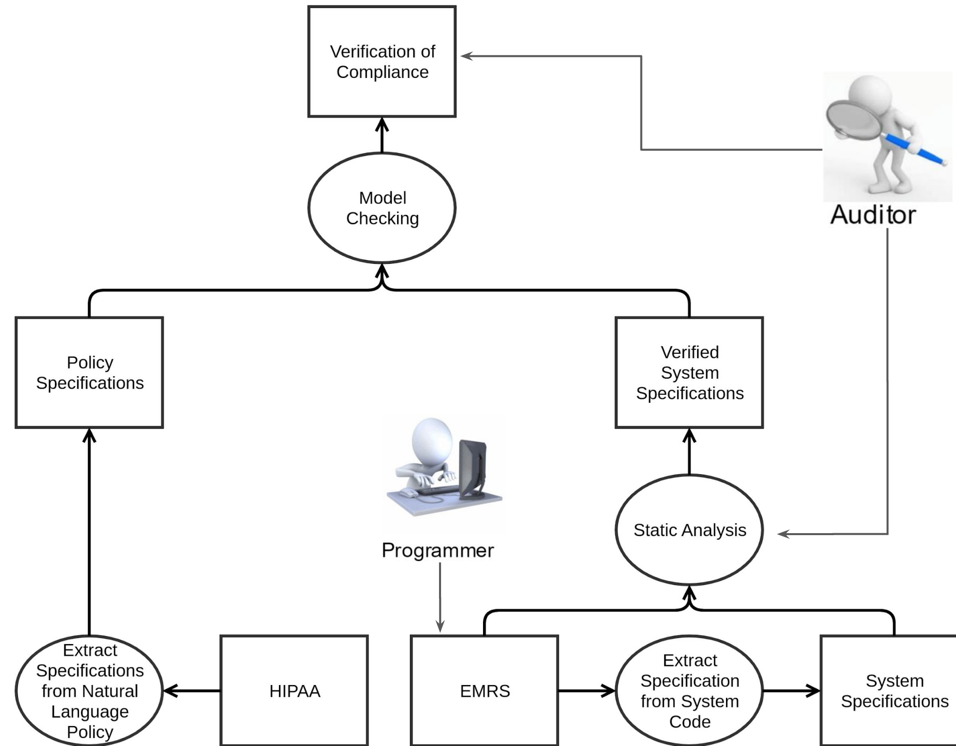
```
public ResultSet runEmployeeQuery(Connection conn, String name){
     PreparedStatement stmt = conn.prepareStatement("SELECT * " +
          "FROM employees WHERE name = ?");
     stmt.setString(1, name);
     ResultSet ts = stmt.executeQuery()
     return rs;
}
...

// "canQueryEmployee()" returns true if current user is authorized to
// query the employees table.
if(AuthCheck.canQueryEmployee()){
     ResultSet employeeRecord = runEmployeeQuery(dbConn, employeeName);
}
```

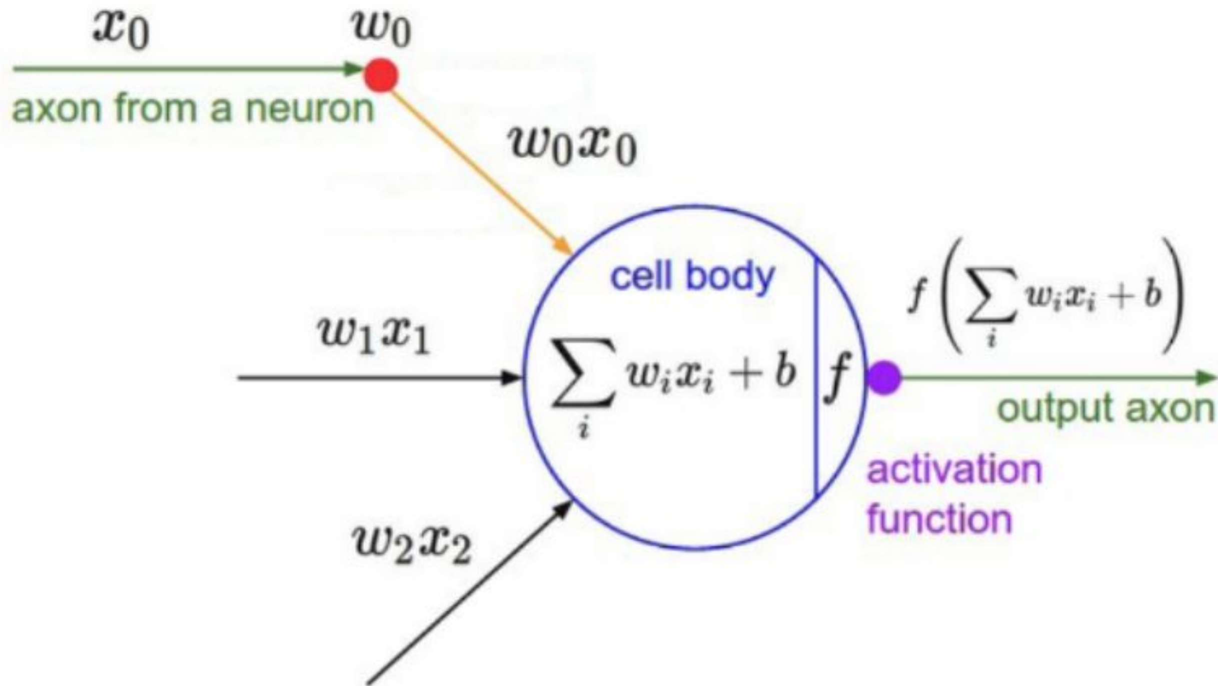
Improper Authorization Pattern Detection Example

```
public void sawOpcode(int seen) {
     if ("AuthCheck".equals(classConstant) &&
               seen == INVOKESTATIC &&
               "canQueryEmployee".equals(nameConstant) && "()Z".equals(sigConstant)) {
          seenGuardClauseAt = PC;
          return;
     }
     if (seen == IFEQ && (PC >= seenGuardClauseAt + 3 && PC < seenGuardClauseAt + 7)) {
          logBlockStart = branchFallThrough;
          logBlockEnd = branchTarget;
     }
     if (seen == INVOKEVIRTUAL && "runEmployeeQuery".equals(nameConstant)) {
          if (PC < logBlockStart || PC >= logBlockEnd) {
               bugReporter.reportBug(
                    new BugInstance("IMPROPER_AUTHORIZATION", HIGH_PRIORITY)
                    .addClassAndMethod(this).addSourceLine(this));
          }
     }
}
```

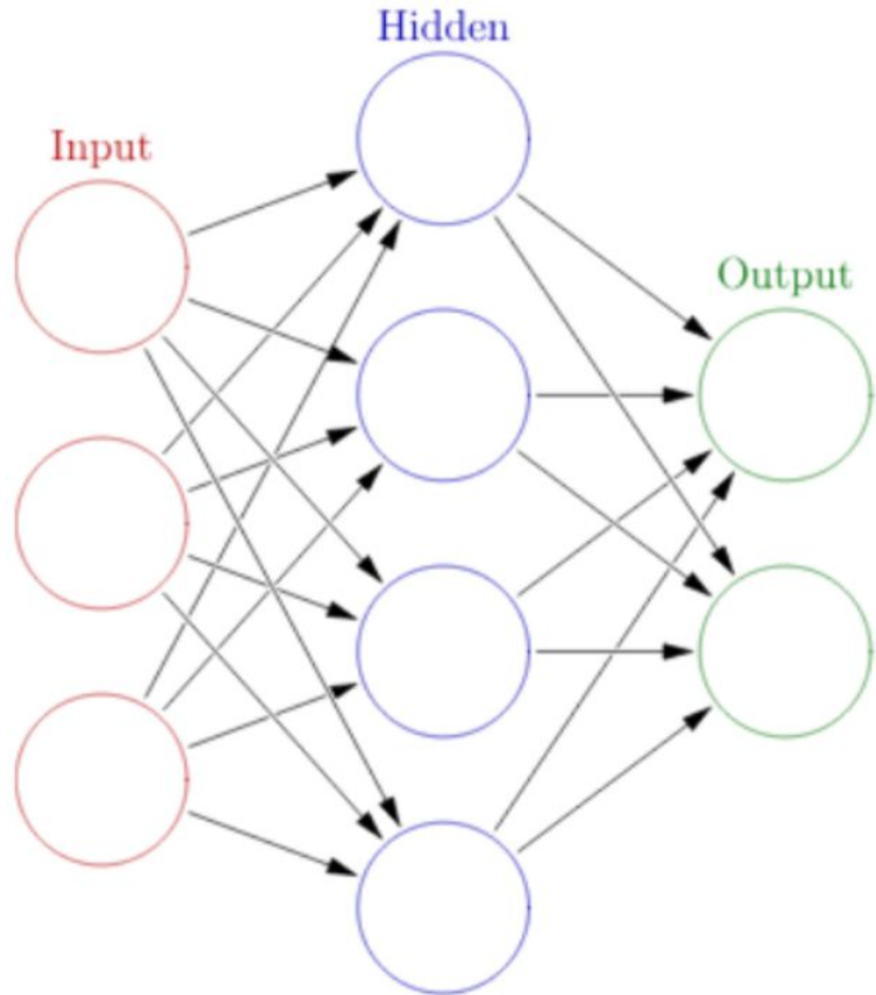
Privacy Policy Verification Using Model Checking



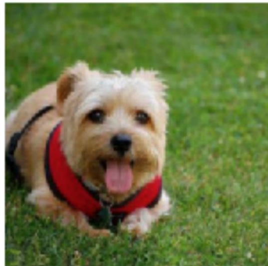
Background: Artificial Neuron

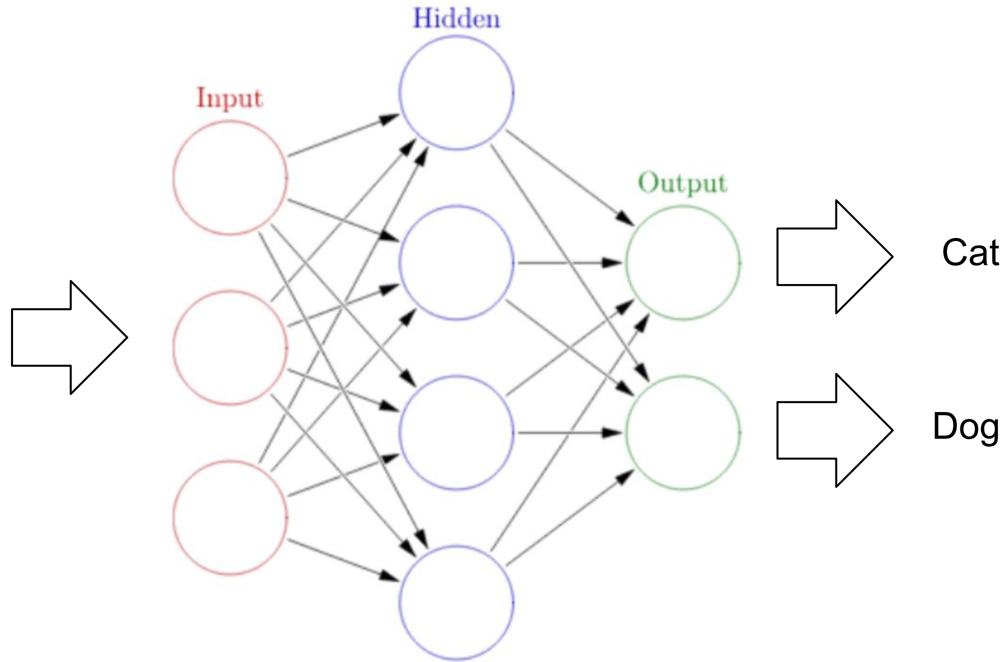


Background: Feed-Forward Neural Networks (FFNNs)



Background: Feed-Forward Neural Networks (FFNNs)





Cat

Dog

Background: Vector Representations

- Neural networks and deep learning work great for numerical data, but can't perform calculations on code because it is text

- Need to convert code to some numerical representation

- A vector representation is an *m*-dimensional real-valued vector representing the relative meaning of a word (compared to other words in the vocabulary)

  Learns vector representations based on the language model

king = {2,0,3}

queen = {2,2,2}

man = {0,1,4}

woman = {0,3,3}

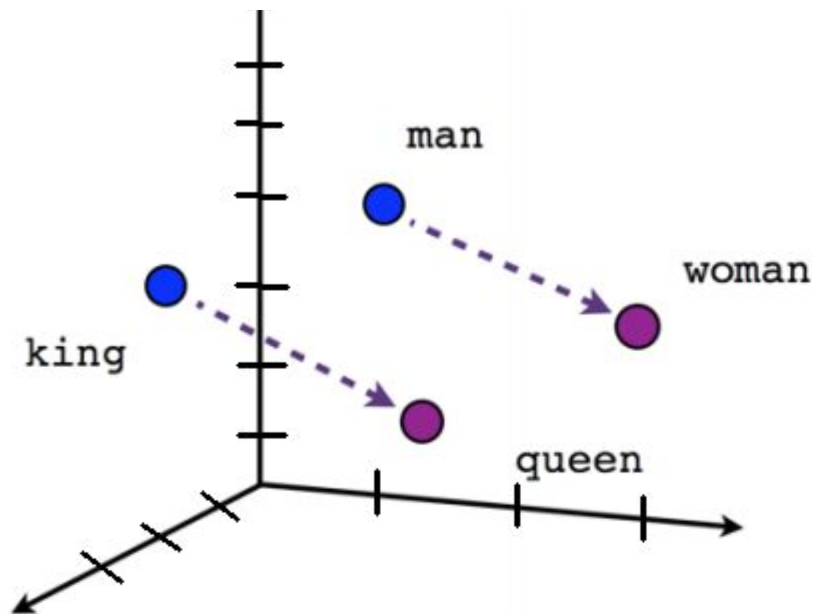
Background: Vector Representations

king = {2,0,3}

queen = {2,2,2}

man = {0,1,4}

woman = {0,3,3}



How do we learn these vector representations?

Background: Language Model

- A language model is a probability distribution of occurrence of a sentence (or sequence of words) or the next word in a sequence

$$P(W) = P(w_1, w_2, w_3, ..., w_n)$$

P("I ran to the store for groceries")

- Vector representations in deep learning attempt to model the meaning of words

Background: Language Model

- Predict the next word given a previous sequence of words

  “I” -> “ran”

  “I ran” -> “to”

  …

  “I ran to the store for” -> “groceries”

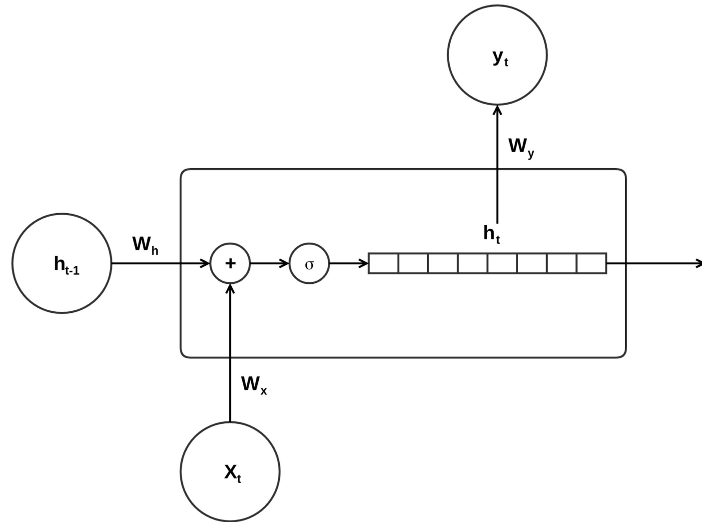
- Limitations of the language model:
  - The data sparsity problem (need a huge corpus to learn on)
  - If new words are encountered after training they cannot be handled

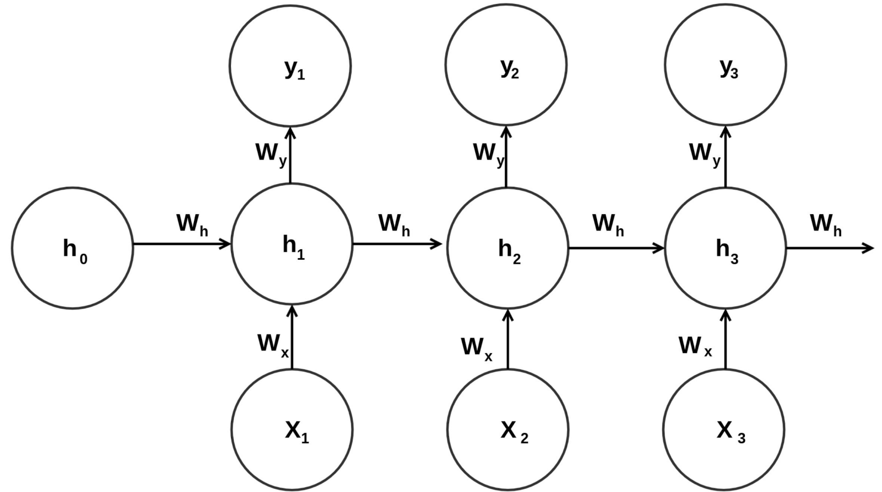
Previous Research in the Literature

- On the Naturalness of Software (Hindel et al.)
  - Software exhibits behavior like a natural language, and can therefore be treated like a natural language
  - Can have natural language processing techniques and language model be applied to software
  - Can learn software using deep learning similar to natural language

- Previous deep learning on code can be split into two main categories:
  - Models that apply word prediction exactly like NLP
  - Models that use an abstract syntax tree (AST) and perform prediction using AST nodes
    - Perform better on average

Background: Recurrent Neural Networks (RNNs)





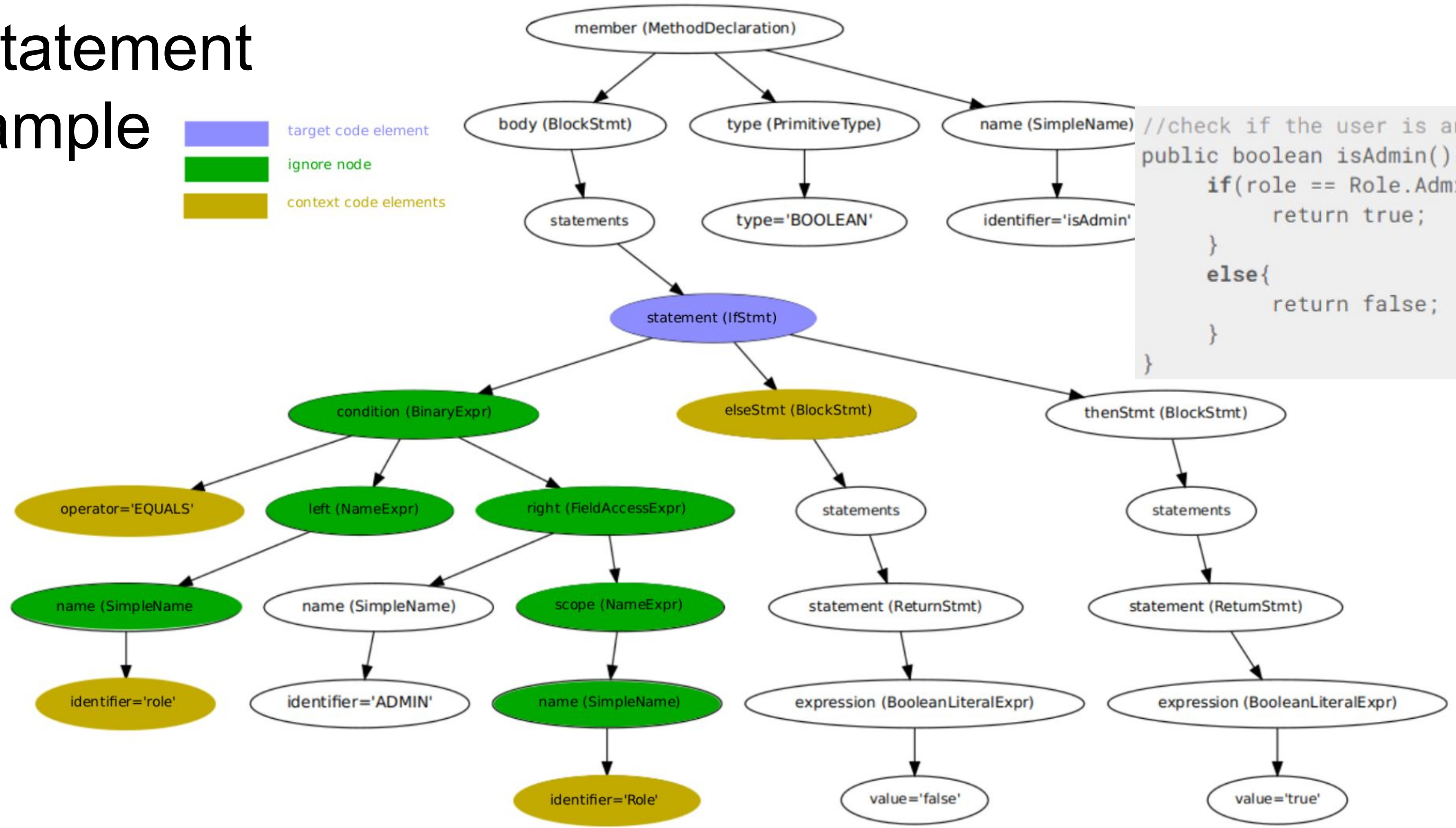
Need to Learn Word Embeddings Differently

- Previous word embeddings based on definition of a language model was based on statistical frequencies of co-occurrences of code elements (a ***contextual-oriented model***)

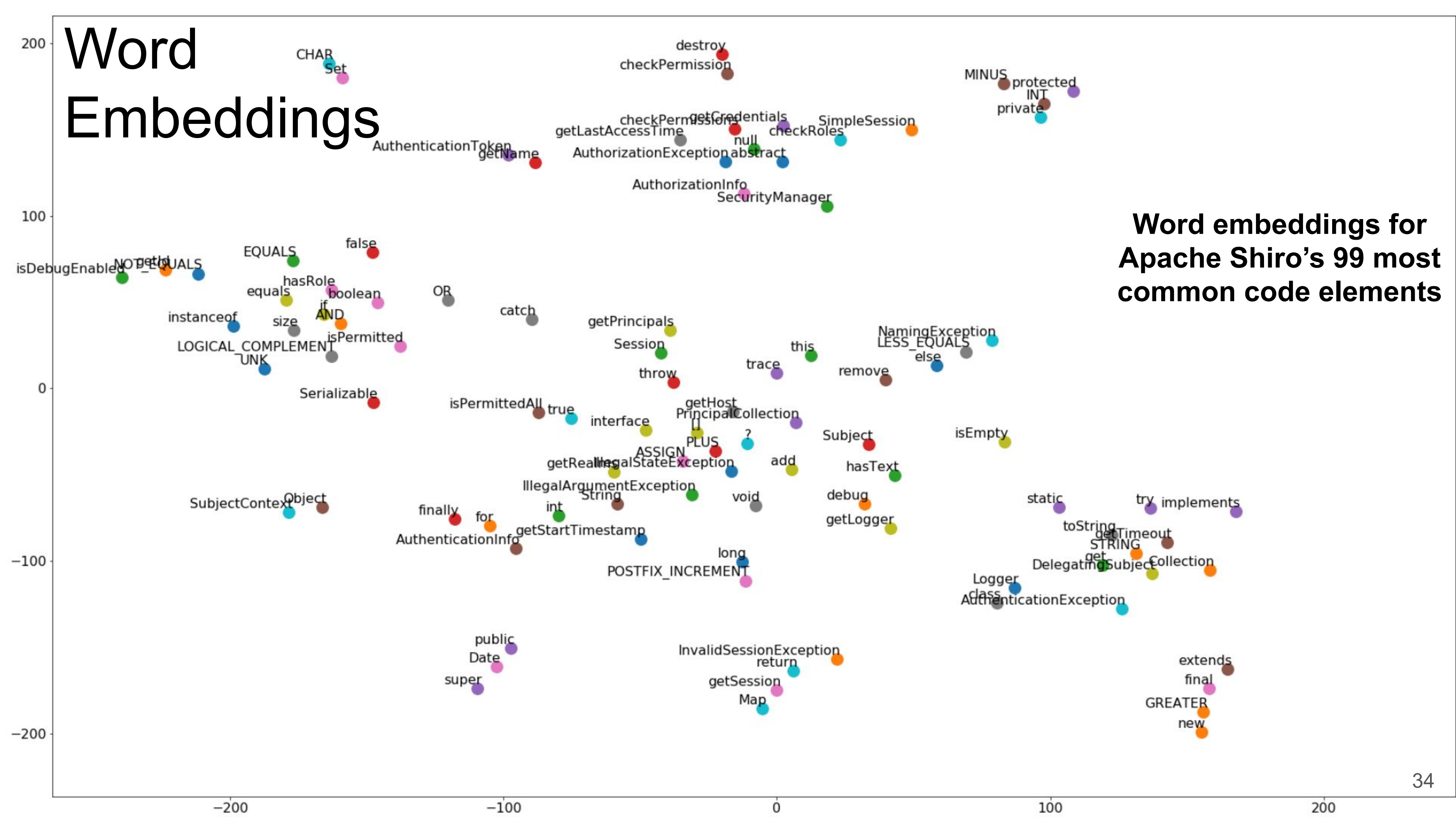
Behavioral Language Model Code Element Collection Rules

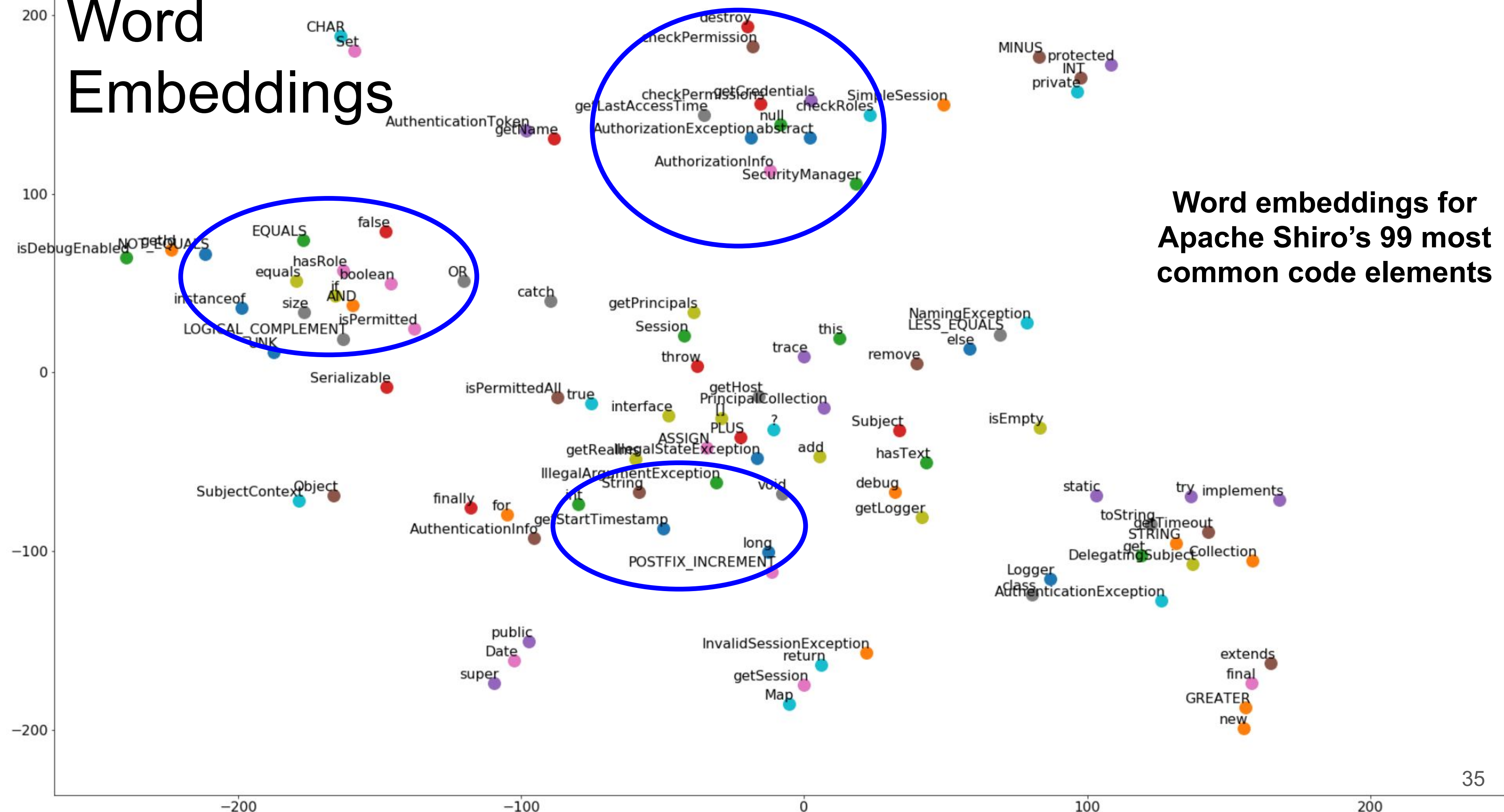
Node Types	Identifier	Code Elements Used As Context
Array Creation Expression	array data type	"new []"
Array Type	"[]"	array data type
Assignment Expression	assign operator	target and value expressions
Binary Expression	binary operator	left and right term expressions
Boolean Literal	"true" or "false"	"boolean"
Break Statement	"break"	associated loop or switch statement
Cast Expression	cast data type	expression being cast
Catch Clause	"catch"	exception types being caught
Char Literal	"CHAR"	"char"
Class Or Interface Declaration	name	"class" or "interface" and modifiers, interfaces, extension, and members
Conditional Expression	"?"	condition expression and "else"
Continue Statement	"continue"	associated loop statement
Do Statement	"do"	"while" and condition expression
Double Literal	"DOUBLE"	"double"
Enum Declaration	name	"enum" and all modifiers
Explicit Constructor Invocation	"this" or "super"	associated arguments and expressions
Field Declaration	field data type	initializations, assignments, and modifiers
For Each Statement	"for"	"break" and "continue" if present and variable and iterable types
For Statment	"for"	"break" and "continue" is present and initialization type, condition expression, and update expression
If Statement	"if"	condition expression and "else" if present
InstanceOf Expression	"instanceof"	expression and instance type
Integer Literal	"INT"	"int"
Long Literal	"LONG"	"long"
Method Call Expression	name	arguments
Method Declaration	name	modifiers, return type, parameters, thrown exceptions, and method body
Null Literal	"null"	"void"
Object Creation Expression	object data type	"new", arguments, and anonymous class body if present
Primitive Type	primitive data type	associated expressions or declarations
Return Statement	"return"	associated expression
String Literal	STRING	"String"
Super Expression	"super"	associated expression
Switch Entry Statement	"case" or "default"	"switch" and associated label expression if present
Switch Statement	"switch"	selector expression and switch entry statements
Synchronized Statement	"synchronized"	associated expression
This Expression	"this"	associated expression
Throw Statement	"throw"	associated expression
Try Statement	"try"	resource expressions, catch clauses if present, and "finally" if present
Unary Expression	unary operator	associated expression
Variable Declaration Expression	variable data type	modifiers and initialization expression if present
Void Type	"void"	associated expressions or declarations
While Statement	"while"	"break" and "continue" if present and condition expression

If-Statement Example



Node Types	Identifier	Code Elements Used As Context
If Statement	"if"	condition expression and "else" if present


Word Embeddings
Word embeddings for Apache Shiro's 99 most common code elements
destroy
checkPermission
CHAR
Set
MINUS
protected
INT
private
getLastAccessTime
SimpleSession
checkRoles
null
AuthenticationToken
AuthorizationException
abstract
AuthorizationInfo
SecurityManager
EQUALS
false
isDebugEnabled
hasRole
equals
boolean
OR
catch
instanceof
size
AND
isPermitted
getPrincipals
Session
NamingException
LESS_EQUALS
else
LOGICAL_COMPLEMENT
UNK
this
trace
remove
throw
Serializable
isPermittedAll
true
interface
getHost
PrincipalCollection
Subject
isEmpty
?
PLUS
ASSIGN
add
hasText
IllegalArgumentException
String
void
debug
static
try
implements
SubjectContext
Object
finally
for
int
getLogger
toString
getTimeout
STRING
getStartTimestamp
AuthenticationInfo
long
POSTFIX_INCREMENT
Collection
Logger
class
AuthenticationException
public
Date
super
InvalidSessionException
return
getSession
Map
extends
final
GREATER
new
200
100
0
−100
−200
200
34



Word Embeddings
Word embeddings for Apache Shiro's 99 most common code elements
CHAR
Set
destroy
checkPermission
AuthenticationToken
getName
getLastAccessTime
checkRoles
SimpleSession
AuthorizationException
abstract
null
AuthorizationInfo
SecurityManager
MINUS
protected
INT
private
isDebugEnabled
EQUALS
false
hasRole
equals
boolean
OR
AND
instanceof
size
isPermitted
LOGICAL_COMPLEMENT
catch
getPrincipals
Session
this
trace
throw
NamingException
LESS_EQUALS
else
remove
Serializable
isPermittedAll
true
interface
getHost
PrincipalCollection
PLUS
ASSIGN
Subject
isEmpty
add
hasText
IllegalArgumentException
String
void
debug
getLogger
SubjectContext
Object
finally
for
int
AuthenticationInfo
getStartTimestamp
long
POSTFIX_INCREMENT
static
try
implements
toString
getTimeout
STRING
get
DelegatingSubject
Collection
Logger
class
AuthenticationException
public
Date
super
InvalidSessionException
return
getSession
Map
extends
final
GREATER
new
200
100
0
−100
−200
−200
−100
0
100
200
35

Limitations of the Behavioral-Oriented Model

- Still subject to the data sparsity problem
- Still cannot handle new code elements

References

- **John Heaps**, Xiaoyin Wang, Travis Breaux, and Jianwei Niu. *Toward detection of access control models from source code via word embedding.* In Proceedings of the 24th ACM Symposium on Access Control Models and Technologies, pages 103–112. ACM, 2019.

- Johnson, Claiborne and MacGahan, Thomas and **Heaps, John** and Baldor, Kevin and von Ronne, Jeffery and Niu, Jianwei. "Verifiable Assume-Guarantee Privacy Specifications for Actor Component Architectures." *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*. ACM, 2017.