VarSem: Declarative Expression and Automated Inference of Variable Usage Semantics

Yin Liu
Software Innovations Lab
Virginia Tech, USA
yinliu@cs.vt.edu

Eli Tilevich
Software Innovations Lab
Virginia Tech, USA
tilevich@cs.vt.edu

Abstract

Programmers declare variables to serve specific implementation purposes that we refer to as *variable usage semantics (VUS)*. Understanding VUS is required for various software engineering tasks, including program comprehension, code audits, and vulnerability detection. To help programmers understand VUS, we present a new program analysis that infers a variable's usage semantics from its textual and context information (e.g., symbolic name, type, scope, information flow). To support this analysis, we introduce VarSem, a domain-specific language, in which a variable's semantic category is expressed as a set of declarative rules. VarSem's execution determines which program variables belong to a given semantic category. VarSem translates high-level declarative rules into low-level program analysis techniques, including natural language processing and data flow, and provides a highly extensible architecture for specifying new rules and analysis techniques. We evaluate VarSem with eight real-world systems to identify their personally identifiable information variables. The evaluation results show that VarSem infers variable semantics with satisfying accuracy/precision and passable recall, thus potentially benefiting both software and security engineers.

Keywords: variable usage semantics, NLP-based analysis, domain specific language

ACM Reference Format:
Yin Liu and Eli Tilevich. 2020. VarSem: Declarative Expression and Automated Inference of Variable Usage Semantics. In *GPCE 2020 - 19th International Conference on Generative Programming: Concepts Experiences, Sun 15 - Fri 20 November 2020, Illinois, United States.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/1122445.1122456

GPCE'20, Sun 15 - Fri 20 November 2020, Illinois, United States
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

1 Introduction

In a computer program, variables are introduced for specific implementation purposes. We refer to such purposes as *variable usage semantics*. Understanding variable usage semantics is required for the majority of maintenance and evolution tasks, ranging from program comprehension to code audits [20, 31]. To that end, both the textual (e.g., naming scheme) and context (e.g., data flow) information of a variable should be examined, as it is their confluence that uncovers the variable's intent and responsibility [31].

Consider the task of identifying the variables that store user-entered passwords. This task is required for inspecting how passwords are protected in a system. As is shown in Figure 1, a security engineer, would need to ❶ search for variables whose names bear similarity to the word "password" (e.g., "pwd", "passwd", and "pass_word"), ❷ search for variables whose type can store password information (i.e., string), and ❸ search for variables whose context information matches certain usage patterns (e.g., information flows from user input to a variable). These search rules can be executed either independently or as a logical chain.

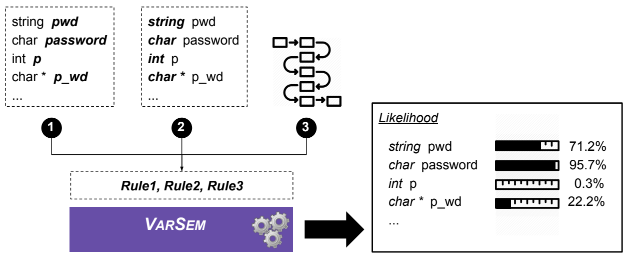


Figure 1. Identify variables storing user-entered passwords

For task ❶, programmers typically use a source code search facility, such as the Unix grep. For example, issuing the command `grep "pass*word" *.c` on the Unix shell would return all the textual matches with the prefix "pass" and the suffix "word" contained in the C source files located in the current directory. However, this method is quite brittle: 1) the actual names for password variables often differ greatly from the word "password". A simple search of GitHub reveals the following names for password variables and their frequencies: "passwd" – 11M cases, "p_wd" – 9,258 cases, "pass_wd" – 280 cases, and "p_w_d" – 164 cases. Even worse, a lazy developer may name a password variable with a single letter "p" or an irrelevant string "abc." Hence, string matching would be quite inadequate in its ability to identify all potential password variable names. Besides, not all matches would be against variables, as some of them will appear in comments, annotations, and class/function names, thereby requiring additional manual efforts to filter out.

For tasks ❷ and ❸, manually inspecting the variables' types and information flows would be feasible for small projects, but quite unrealistic for large projects. Compiler-based analysis tools (e.g., LLVM ASTMatcher [46]) and existing program analysis frameworks [32, 43, 52] can be used to search for variable types and information flows automatically. However, to apply these tools correctly requires that programmers build sufficient expertise in compilers and program analysis. It would be unrealistic to expect that level of expertise in esoteric topics from all application programmers or security engineers.

The latest developments in machine learning (ML), deep learning (DL), and natural language processing (NLP) provide new tools for analyzing variable usage semantics. However, applying these state-of-the-art approaches correctly is quite challenging due to the following reasons: **(a)** existing applications of these tools are quite dissimilar, including recommending identifier names [1, 2, 4, 5], summarizing code [3, 51], de-obfuscating code [50], completing code [40], and debugging method names [25]. In other words, these approaches apply dissimilar ML/DL/NLP models to fit their specific scenarios, so no single model or technique could be used out of the box for variable usage semantics analysis; **(b)** the accuracy, precision, and recall of many of these techniques are insufficient for practical variable usage semantics analysis tasks. Consider recommending descriptive identifier names as an example: the state-of-the-art approach code2vec [5] achieves an imperfect precision (63.1%) and recall (54.4%). Even worse, Jiang et al. [28] show that in more realistic settings, code2vec's performance decreases even further; **(c)** the targets of these techniques are usually code segments rather than individual variables. That is, the contained variables' textual and context information are treated as tokens and attributes of the entire code snippet. However, variables are distinct program constructs. Not only does a variable store information, but it also can interact with other program constructs (e.g., class fields, function parameters) and be part of every program context (e.g., control and data flow). Understanding what variables are used for (i.e., their usage semantics) has not been pursued previously as a distinct program analysis problem.

What if one could write simple statements that would execute tasks ❶❷❸ by correctly employing the required state-of-the-art analysis techniques and tools as well as leveraging the available domain knowledge (e.g., software/security engineers)? In this paper, we present *variable usage semantics analysis*, a new program analysis that identifies variable usage semantics, and VarSem, a domain-specific language that reifies this analysis. As shown in Figure 1, one can declaratively specify the rules to execute tasks ❶ (Rule1: compare variables' symbolic names to the word "password"), ❷ (Rule2: identify variables whose type is `string`), and ❸ (Rule3: identify variables whose values flow from user input). By executing the specified rules, VarSem assigns a statistical probability value to each program variable to designate how closely it matches the given rules (e.g., pwd 71.2%), thus assisting the user in understanding the variables' usage semantics.

To ensure satisfactory accuracy, precision, and recall, VarSem programs express both the textual and context information of variables. VarSem provides built-in analyses: data-flow for the context information and a novel NLP-based analysis for the textual information. To cover potentially enormous analysis scenarios, VarSem's extensible architecture enables software/security engineers to contribute highly customizable rules, and integrate with other ML/DL/NLP-based program analysis techniques.

The contribution of this paper is three-fold:

(1) We introduce a new program analysis, *variable usage semantics analysis (VUSA)*, that infers a variable's usage semantics from its textual and context information.

(2) We reify VUSA as the VarSem DSL that offers: (a) an intuitive declarative programming model; (b) built-in analyses: an NLP-based and data-flow analyses, for textual and context information, respectively; (c) an extensible architecture for including custom rules/analyses.

(3) We evaluate VarSem's performance in inferring personally identifiable information (PII) variables in eight open-sourced projects. VarSem infers variable semantics with satisfying *accuracy* (>= 80% in 13 out of 16 scenarios), *precision* (>= 80% in 13 out of 16 scenarios), and passable *recall* (> 60% in 8 out of 16 scenarios).

2 Design Philosophy and Overview

We first introduce the key idea behind VarSem and explain its high-level design. Then, we discuss the technical background required to understand our contribution. Finally, we revisit the motivating example above by applying VarSem.

2.1 Variable Usage Semantics Analysis

Our basic idea of *variable usage semantics analysis* is inspired by philosopher Ludwig Wittgenstein's Theory of Natural Language Construction: "the meaning of a word is its use in the language [55]." That is, the meaning of a word is determined not only by its textual representation, but also by the total set of usages of this word in the language. As a result, a word's meaning is not immutable: once its new usages appear and become acceptable, its meaning will gradually change as well. Hence, any dictionary-provided definition can only approximate the actual meaning of a word [25].

Inspired by Wittgenstein's theory, we treat program variables as words in a natural language, so the semantics of variable usage can be interpreted analogously to the meaning of words. Thus, to infer a variable's usage semantics, our approach is to consider both the variable's symbolic name (i.e., textual information) and how it is used in the program (i.e., context information).

A typical practical application of *variable usage semantics analysis* is determining the likelihood of variables belonging to a given *semantic category* (e.g., "password", "user information", "phone number"). Because of the intrinsic ambiguity of determining such likelihood, our approach relies on statistical probability to report the inferred results. Our approach defines a semantic category as a set of rules against which each program variable is evaluated. When matched, a rule increases the likelihood of a variable belonging to the category specified by the whole rulebase. This approach is driven by the well-known insight that the results of data analysis depend on the analyst's domain knowledge (e.g., the analyst understands how password variables are used in a system) [50, 54]. The ability to specify a highly configurable rulebase, with both predefined rules and new customized rules, avails VarSem for the dissimilar needs of domain experts in need of inferring variable usage semantics.

2.2 Definitions & Assumption

Textual & Context variable information. The properties of any program variable divide into intrinsic (e.g., symbolic name, type, and scope) and extrinsic (data and control flow). *Textual information* refers to a variable's symbolic name; *context information* refers to all its other properties.

Personally identifiable information (PII)[44]. PII encompasses all information that would harm a person's security or privacy if exfiltrated. Typical PII examples are social security numbers, credit/debit card numbers, and healthcare-related data. Our evaluation focuses on PII variables, specifically the ability of VarSem to identify them, so developers could then protect PII from being leaked and tampered with.

Expected Expertise Assumption. To be able to describe target variables in VarSem, users are expected to possess some domain knowledge about how the functionality of interest is implemented. For example, to specify variables that store passwords, a VarSem user is expected to be knowledgeable about how password-based authentication works in the analyzed projects. For example, in a certain "login" function, a password candidate, entered by an end user, is compared with the known password, retrieved from some storage.

2.3 Motivating Example Revisited

Recall the problem of identifying the variables that store user-entered passwords. Variables that belong to this semantic category match the following three rules: ❶ their symbolic names bear close similarity to the word "password", ❷ their type must be able to store textual data (i.e., `string`), and ❸ their value must have flown from some input. In VarSem, these rules can be expressed in a few lines of code as follows:

```
1 sem ("identifyPWD") {
2     $var.Name ~ "password"
3     $var.Type ~ STRING
4     $var.Value <~~ USER_INPUT
5 }
6 Result res = run ("identifyPWD")
7               on ("./src/*.c") threshold 0.8
```

The keyword **`sem`** creates a new semantic category, identified by its unique parenthesized name (**`sem`** `("indentifyPWD")` on line-1). This semantic category is defined by the three rules described above: ❶ (line-2), ❷ (line-3), and ❸ (line-4), respectively. The keyword **`$var`** represents a program variable, and it includes attributes (e.g., `Name`, `Type`, `Value`) that can be used to form rules. VarSem provides several built-in operators (e.g., ~: match, <~~: information flow), each of which returns a statistical probability, expressed as a percentage point. The informal semantics of Rule ❶ on line 2 is "compare a variable's symbolic name to the word 'password', returning their degree of similarity." Rule ❷ determines whether a given variable's type can hold a textual representation (e.g., **`char`** `*`, **`char`**`[]`, **`const char`**`*`, `string`). Rule ❸ determines whether an information flows from any user input to the given variable's value. Each program variable is bound to **`var`**, one at a time, with the rules executed, once their semantic category ("identifyPWD") is **`run`** on a given codebase (lines 6,7). These rules execute in any order, in sequence or in parallel. The optional **`threshold`** filter applies a weighted average formula to report only those variables whose likelihood of matching the rules are above the given threshold.

2.4 Required Language Features

The example above is simplified for ease of exposition. To become a truly versatile tool for inferring variable usage semantics, VarSem must provide the following features:

(1) Rules should be straightforward to reuse within and across semantic categories. It should be possible to make the execution of a rule conditional on the results of executing other rules.

To provide feature (1), a semantic category supports add/remove/update rule operations (§ 3.3-3). Besides, the results of running a category can be examined, with the results used as conditional and control flow statements (§ 3.4-2).

(2) It should be straightforward to redefine how rules and their analysis routines are bound to each other. Although each built-in rule has a default analysis routine, it should be possible to change these routines.

To provide feature (2), the **`bind`** keyword makes it possible to bind new analysis routines to existing rules (§ 3.3-2).

(3) It should be possible for rules to have different levels of importance. When determining whether a variable belongs to a given semantic category, each rule can have a dissimilar impact. For example, how can the user specify that it is more important for a variable's name to be similar to the word "password" than to have some user input to flow into the variable's value?

To provide feature (3), a rule can be followed by the keyword **`impact`** and an integer value between 1 and 10 (§ 3.3-2).

(4) VarSem should be extensible with both new rules and their analysis routines. Due to the large number and variety of analysis techniques and tools, it would be highly advantageous to be able to add them to VarSem or use them to update the analysis routine of existing rules.

To provide feature (4), VarSem can generate keywords/operators for new rules and function skeletons for corresponding analysis routines (§ 3.5).

3 The VarSem Language

We start by presenting the high-level architecture of VarSem. Then, we briefly introduce the major technologies that enable VarSem. Finally, we detail each major VarSem component.

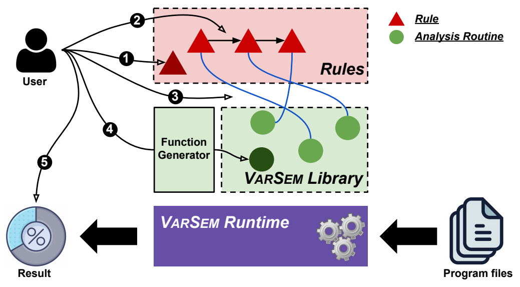


Figure 2. VarSem overview

3.1 Architecture

Figure 2 shows the three main components of VarSem: `Rules`, `Library`, and `Runtime`. `Rules` are user-defined search criteria that define a semantic category. `Library` provides a collection of program analysis routines (i.e., green circles), both standard and custom, that are bound to rules (i.e., red triangles) to reify them. The bindings can be default or user-defined. For example, a standard data-flow analysis method can be bound to the rule — "variable's value comes from user input." `Runtime` executes the rules in VarSem's semantic category by invoking the bound methods. It keeps track of each rule's execution, weighted-averaging the statistical probability of each program variable's belonging to the semantic category specified by the rulebase.

VarSem's programming model supports five inter-related development tasks: ❶ define a semantic category, ❷ add/remove/update rules in a semantic category, ❸ bind rules to analysis routines, ❹ create new rules and analysis routines, and ❺ apply a semantic category to a codebase, filtering and analyzing the results. Thus, by defining a semantic category with its associated rules (❶❷), VarSem can flexibly express a variable's combined textual and context information. VarSem's built-in rules and their analysis routines make use of a novel NLP-based analysis (for textual information) and classic data-flow techniques (for context information). By creating and binding rules and their analysis routines (❸❹), VarSem can be extended with new and improved analysis techniques and frameworks. Finally, VarSem's execution results can be analyzed and filtered as required to support various scenarios in inferring variable usage semantics (❺).

3.2 Technical Background

Next we briefly describe the main technologies that power VarSem. **1) Scala**, a modern programming language, combines object-oriented and functional programming features [24]. Scala serves as the host language for VarSem, defined as an embedded DSL. We choose Scala because of its flexible syntax (e.g., optional parentheses), its support for defining new keywords and custom syntax, and functional programming features (e.g., support for higher-order functions). It is these features of Scala that make it possible for VarSem's library to be easily extended with new analysis techniques. Finally, as a JVM-based language, Scala runs on multiple platforms, making VarSem platform-independent.
2) Natural language processing (NLP) approaches have been widely applied to identify program's sensitive textual information (e.g., comments, descriptions) [26, 37, 57]. VarSem relies on NLP to provide a novel textual analysis that is more powerful than that of a regular expression search.
3) Data-flow Analysis, a standard static program analysis technique, infers how values propagate through a program. Data-flow analysis has been applied to detect code vulnerabilities [10, 29, 43]. By employing the standard data-flow analysis, VarSem identifies variables' context information (e.g., information flows from user input to a variable).
4) LLVM [33] is a compiler-based program analysis infrastructure. LLVM features `libtooling tool` [48] and `AST-Matcher`[46] that analyze at the source code level. By means of a customized `libtooling tool`, VarSem extracts the intrinsic properties of variables (e.g, name, type, and scope).

3.3 Syntax, Semantic Category, and Rules

The two key abstractions of VarSem are *a semantic category* and *rules*, explained in turn next. [1]

(1) Semantic Category defines a category of variables with the same usage semantics. In VarSem, it is expressed as a group of rules (i.e., a rulebase) that describes a variable's intrinsic and extrinsic properties. A variable's membership in a semantic category is determined by the variable matching the category's rules. The keyword **`sem`** creates a new semantic category that comprises a unique name and a rulebase.

(2) Rule matches each program variable's intrinsic or extrinsic properties. A typical VarSem rule includes an operator and two operands. The *lhs* operand is an analyzed variable, while the *rhs* operand is a specific target (i.e., intrinsic or extrinsic targets). A rule can also include its impact modifier and custom analysis routine.

[1]VarSem's complete EBNF specification appears in drive.google.com/file/d/1cAl5XmN791TXjp3T8CUgjicfCbr_N4jw/view?usp=sharing

To describe a variable's intrinsic properties, a rule starts with the keyword **$var**, an intrinsic attribute (Name, Type, or Scope), the intrinsic operator (~), and the attribute's match-target. VarSem predefines several match-targets: string values for matching the attribute Name, STRING/INT/BOOL/DOUBLE/COMPOSITE for the attribute TYPE, and LOCAL/GLOBAL for the attribute Scope. $**var**.Scope ~ LOCAL expresses "match if the variable's scope is local."

To describe a variable's extrinsic properties, a rule starts with the keyword **$var**, the extrinsic attribute (Value), one of VarSem's extrinsic operators: data flow (<~~ or ~~>) or comparison (<=>), and one of the right-hand side (rhs) operands. The data flow operator's rhs operands can be one of USER_INPUT, STABLE_STORAGE, WWW, or a function name. The comparison operator's rhs operands can be either another variable's attribute (i.e., Value) or no operand. The comparison operator with no rhs operand (e.g., **$var**.Value <=>), expresses that any comparison with the variable's value should be matched.

To create rules that involve more than one variable, VarSem provides the var_ operator. If more than two variables need to be differentiated, var_ takes optional numeric parameters (e.g., var_(1)). To describe scenarios that use extrinsic properties, multiple rules can be combined. For example, the code snippet below describes one such scenario: one variable flows from some user input (line 1), another variable flows from some stable storage (line 2), and these two variables are compared (line 3).

```
1    $var.Value <~~ USER_INPUT &
2    $var_.Value <~~ STABLE_STORAGE &
3    $var.Value <=> $var_.Value
```

Because each matched rule can dissimilarly impact whether a variable belongs in a given semantic category, VarSem provides the **impact** keyword that takes an integer in the range [1, 10]. Impact levels control the calculation of weighted averages that determine which variables belong to the specified semantic category (see § 3.4).

Finally, a specific analysis routine can be bound to a rule using the **bind** keyword. VarSem's standard library comes with a set of predefined analysis routines for core analysis techniques (e.g., NLP_VAR_NAME_ANALYSIS, STRING_MATCH, and DATA_FLOW_ANALYSIS). For example, the code snippet below is the rule ("match a variable's name to the word 'password'"), with the rule impacting the final statistical result to the degree of 7, and the analysis routine NLP_VAR_NAME_ANALYSIS performing the required analysis.

```
$var.Name ~ "password" impact 7
              bind NLP_VAR_NAME_ANALYSIS
```

If **impact** is not specified, the default value of 10 is assigned. Without custom analysis routines, VarSem binds NLP_VAR_NAME_ANALYSIS for rules of textual information and DATA_FLOW_ ANALYSIS for rules of context information.

(3) Supporting Reuse and Evolution of VarSem programs. Both semantics categories and individual rules can be systematically reused and evolved. In particular, rules can be inserted, removed, or updated in a semantic category, while semantic categories can be combined or intersected.

3.4 Reporting Output

(1) Obtaining results. The overloaded function **run** takes either a single rule or a semantic category as the parameter. The keyword **on** specifies the target codebase. The returned **Result** is a collection of key-value pairs, where the key points to a program variable and the value points to the likelihood of the variable belonging to the specified semantic category. The returned collections can be filtered to include only those variables whose likelihood exceeds a given threshold. The code snippet below runs the semantic category "identifyPWD_2" on the codebase located in "./src/*.c", and filters out all the variables whose likelihood value is less than 0.8.

```
Result res_2 = run ("identifyPWD_2")
                   on ("./src/*.c") threshold 0.8
```

(2) Analyzing the results: The **Result** object provides the max, min, mean, and standard deviation operations, to be executed against the likelihood values. In addition, top/bottom percentages can be retrieved and filtered out. The code snippet below retrieves the maximal likelihood value from "res_2" (line 1), and then retrieves those variables whose likelihood value is in the top 1% (line2).

```
1 var max = res_2.max
2 res_2 = res_2.top(0.1)
```

Conditional and control flow statements can operate on **Result**. The code snippet below has the following logic: if the results of executing the semantic category ("identifyPWD_2") are unsatisfactory (i.e., the max likelihood is less than 0.9), then run the semantic category ("identifyPWD").

```
Result res_2 = run ("identifyPWD_2") on ("./src/*.c")
if (res_2.max < 0.9)
    res_2 = run ("identifyPWD") on ("./src/*.c")
```

Revisiting The Motivating Example Again. Consider evolving the motivating example to return those variables whose likelihood values are in the top 1% and their mean value greater than 0.7. Further, we want to increase the importance of the variable's name. The code snippet accomplishes these changes.

```
Result res_2 = run ("identifyPWD_2") on ("./src/*.c")
for (i <- 1 to 10 if res_2.top(0.1).mean > 0.7) {
    update ("identifyPWD_2") {$var.Name ~ "password"}
    to {$var.Name ~ "password" impact i}
    res_2 = run ("identifyPWD_2") on ("./src/*.c")
}
```

Please, note that even with an **impact** suffix, the updated rules are identified by their **$var**.Name ~ "password" prefix.

3.5 Meta-Programming Support for Creating new Operators and their Analysis Routines

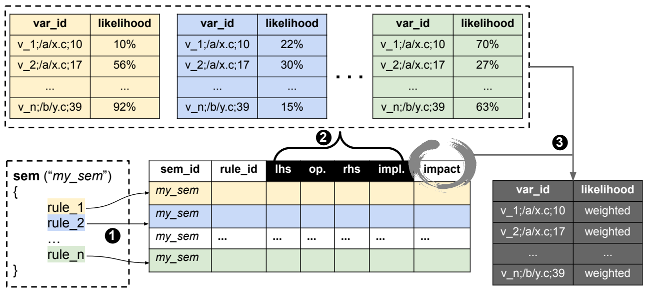
Because variable usage can be defined in a variety of different ways, it would be impossible to provide a fixed set of rules and their analysis techniques to cover all possible scenarios. To make it possible to easily create new rules and their analysis routines, VarSem provides meta-programming facilities. A new attribute can be created with the keyword `attr` followed by the attribute's name. A new operator can be created with the keyword `op` followed by the operator's symbols. VarSem supports the creation of binary operators only, so each new operator has to be accompanied with the declarations of its right-hand side and left-hand side operands by means of the `lhs` and `rhs` keywords, respectively. A new analysis routine can be created with the keyword `impl` followed by the analysis routine's unique name. The code snippet below shows how to add to VarSem a new attribute "Const"(line 1), a new operator"tainted" with its lhs "$var.Value" (i.e., variable's attribute), rhs "("source_f","sink_f")" (i.e., a string pair), and an analysis routine "TAINT" that can be bound to this rule (line 2). Given this meta-programming declaration, VarSem will generate a library that reifies the new attribute (usage example: `$var.Const`), as well as the new operator `tainted` and its analysis routine "TAINT" (usage example: `$var.Value tainted ("source_function","sink_function") bind "TAINT"`).

Note that, it would be impossible to generate a complete analysis routine, so VarSem generates skeletal code that is to be completed as necessary. For example, in the code snippet below, VarSem generates the analysis routine skeleton "TAINT", which uses the variable's attribute information and the string pair ("source_function", "sink_function") as the parameters. To complete the analysis routine, the skeletal body must be filled with the required logic.

```
1  attr "Const"
2  lhs $var.Value op "tainted"
3         rhs ("source_f","sink_f") impl "TAINT"
```

4 Executing VarSem Programs

We first discuss how VarSem executes rules in a given rulebase and then describe VarSem's built-in library.



var_id	likelihood
v_1;/a/x.c;10	10%
v_2;/a/x.c;17	56%
...	...
v_n;/b/y.c;39	92%

var_id	likelihood
v_1;/a/x.c;10	22%
v_2;/a/x.c;17	30%
...	...
v_n;/b/y.c;39	15%

var_id	likelihood
v_1;/a/x.c;10	70%
v_2;/a/x.c;17	27%
...	...
v_n;/b/y.c;39	63%

```
sem ("my_sem")
{
        rule_1
        rule_2
        ...
        rule_n
}
```

sem_id	rule_id	lhs	op.	rhs	impl.	impact
my_sem						
my_sem						
my_sem	...	...	...	...	...	...
my_sem						

var_id	likelihood
v_1;/a/x.c;10	weighted
v_2;/a/x.c;17	weighted
...	...
v_n;/b/y.c;39	weighted

Figure 3. Executing VarSem rules

4.1 Executing Rules

Figure 3 details how VarSem executes the rules in a rulebase. Each included rule is processed in three steps:

❶ generate a rule's information vector: semantic category id, rule id, lhs, operator, rhs, bound analysis routine, and impact level. Specifically, the combination of semantic category id and rule id defines a rule's unique identity; the operator, lhs, rhs, and bound analysis routine determine how to execute the rule; the impact level determines how to weight the rule's results. As discussed in § 3.3, if the impact level or analysis routine is omitted, VarSem assigns the default impact value of 10, and binds the rule to its built-in analysis routine (i.e., `NLP_VAR_NAME_ANALYSIS` for analyzing textual information ; `DATA_FLOW_ANALYSIS` for analyzing context information).

❷ each rule in a rulebase is executed in the listing order (see § 4.2). Each rule follows a strict syntax (i.e., lhs op rhs) and outputs either a percentage or a boolean value for likelihood. VarSem also follows a uniformed format when outputting the intermediate results: a unique id (a variable's name, file path, and line number) and a likelihood value (i.e., see the top of Figure 3). Recall that the likelihood value is the statistical probability of a program variable belonging to the specified semantic category.

❸ the likelihood values are weighted by their impact levels. These weighted results are returned as the final output (i.e., the grey table in Figure 3) as an array of mappings.

4.2 Customizable Runtime

Because VarSem's runtime uses various program analysis techniques, it must be highly configurable to support their dissimilar execution environments. Hence, an advanced user can add to VarSem not only custom analysis routines, but also their specialized execution environments. Although the VarSem's native runtime environment is Scala on the JVM, a new analysis routine may require Python support, as is common for machine learning-based techniques. VarSem supports custom execution environments by deploying them in a Docker container (Figure 4).

5 NLP-Powered Textual Analysis

VarSem includes built-in context and textual analysis routines. A stock data-flow analysis is used out-of-the-box for context information, albeit configured and invoked via declarative rules. In contrast, for VarSem's built-in textual information analysis, we created a novel technique, powered by NLP, that we discuss next.

(1) Rationale. When determining whether a variable belongs to a semantic category, its textual information needs to be compared with some target value to compute their similarity. Recall our motivating example of identifying variables that store user-entered passwords. The rule `$var.Name ~ "password"` matches a variable's symbolic name with the string "password." However, this match would be shallow and insufficient

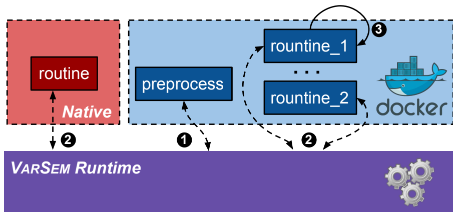


Figure 4. Executing Libraries

❶ **preprocessing a codebase.** Before applying any analysis routine, the runtime makes the target codebase readable by moving it to a shared folder. From the moved codebase, preprocessing extracts the variables' intrinsic properties (e.g., name, type, scope).
❷ **invoking analysis routines.** To invoke a native analysis routine, the runtime calls its entry point's Scala or Java function. To invoke a containerized routine, the runtime launches the corresponding container, so the entry point's function can be called via Docker APIs.
❸ **adding containerized modules.** New containerized routines can be added to the Docker, while existing routines can be updated through a configuration file.

if other aspects of textual information, including type, enclosing unit, and file path, were not taken into account as explained next:

a) Variables with a specific usage tend to appear in certain enclosing units (e.g., functions) and source files. For example, the variable named "pwd" is more likely to store passwords if it is also referenced within a function named "login" whose implementation source code appears in the "authenticate.c" file. However, if a variable with the same symbolic name is referenced within the "unix_shell" function, its usage is likely be related to the implementation of the Unix pwd command (i.e., print working directory), an unrelated semantic usage category altogether.

b) If a variable's type is developer-defined (e.g., struct/class names), the specified type name can be indicative of the variable's usage. For example, many such variable types in our evaluation subjects (e.g., `struct feature_info_t` in the"Biometric Authentication" project) have type names that reveal their variables' usage semantics (§ 6).

c) Adjacent words are semantically connected. That is, if an obscurely named identifier is adjacent to an obviously named identifier, they are semantically related. It is also likely that both of them are describable by the obvious name. For example, the obscurely named variable "pd" whose source file is in the path "a/b/c/password/x.c" should be more likely to store a password than if its source file path were "password/a/b/c/x.c". Although the word "password" appears in both file paths, its position is closer to the source file containing the variable in the first path.

(2) Processing Steps. VarSem uses NLP to compute the similarity between each encountered program variable's textual info and the target word ("password" in our motivating example). The similarity score then determines the likelihood of variables matching the specified usage semantics.

a) Pre-processing: To be used for analysis, the extracted variable's textual information is pre-processed: splitting/-combining identifiers and removing redundancies.

Splitting/Concatenating Identifiers: Even if variables, types, functions, files, and directories are named in a meaningful way, their names can follow dissimilar naming conventions, such as delimiter-separated (e.g., the_pass_word) or camel case (e.g., thePassWord).

To be able to process such names irrespective of their naming conventions, their textual information is pre-processed into semantic sets by splitting and concatenating their constituent substrings. This pre-processing is demonstrated by example below for the names `a_b_c` and `aBcDe`:

$$a\_b\_c \longrightarrow [a, b, c], [ab, c], [a, bc], [abc] \quad (1)$$

$$aBcDe \longrightarrow [a, bc, de], [abc, de], [a, bcde], [abcde] \quad (2)$$

More specifically, an identifier is first split into its constituent parts, sans delimiters, each of which is lower-cased. For example, both "the_pass_word" and "thePassWord" would become identical sets of "the", "pass", and "word." Then, the resulting parts are concatenated into semantic sets. For example, "the_pass_word" could become [the, pass, word], [thepass, word], [the, password], and [thepassword]. Similarly, the semantic sets can be generated for an identifier, separatable into multiple parts (e.g., it_is_a_real_pass_word as a rare case example). Finally, for each semantic set, its similarity score is computed, with the highest observed similarity score reported as the final result.

Removing Redundancies: Only certain identifier parts indicate their construct's usage semantics. For example, in "the_password", "password" is highly indicative, while "the" provides no useful information. Hence, dictionary-based removal gets rid of those parts that correspond to prepositions (e.g., in, on, at), pronouns, articles, and tense construction verbs (i.e., be, have, and their variants). For "the_password", "password" will be retained and "the" removed.

b) NLP Analysis: Algorithm-1's function `NLP_main` outputs variables and their statistical matching likelihood, given a collection of program variables (i.e., `variables`) and a target word (i.e., `target_word`). Each input variable is analyzed, and its identifiers are extracted from the pre-processed variable's name, type, function, and file path (lines 17,19,21,23). Next, function `get_similarity` computes the similarity score between each extracted identifier and the target word (lines 18,20,22,24). Each extracted identifier is broken into constituent words, or combined to a single word. This process generates two representations as described above (i.e., `pass_word` ⟶ (1) `pass`, `word`; ⟶ (2) `password`). Each representation is tokenized (line 3). For each token, its similarity to the target word is computed (lines 5-6), with the similarities accumulated (line 7) and averaged (line 9). The maximum similarity is then selected (lines 10-12) and returned (line 14). An attenuation rate, λ, differentiates adjacent vs. nonadjacent semantic connections. The sum of the obtained similarities of identifiers is the variable's statistical matching likelihood (line 25).

c) An Example: Following the example in § 2.3, consider how variables would be matched with word "password." Assume the variable collection contains variable `pass_word_info` with type `struct information`. The variable is accessed within function `login`, defined in "src/login.c". The name `pass_word_info` is pre-processed into four representations: [`pass, word, info`], [`pass, wordinfo`], [`password, info`], and [`passwordinfo`], while the type, function, and file path into [`information`], [`login`], and [`src, login`], respectively. Each of the tokens [`pass, word, info`] is matched with the target `password`, obtaining the similarity scores 0.5, 0.2, and 0.1, respectively. Hence, the resulting similarity scores becomes 0.27 (i.e., (0.5 + 0.2 + 0.1)/3). The procedure is applied to all representations as well as to those of the corresponding type, function, and file path, with the highest score passed to the next step. All similarity scores are scaled by λ (80% by default). That is, for `pass_word_info`'s file path array [`src, login`], the similarity of `src` is scaled by 80%. If the original similarity of `src` is 1, then it becomes 0.8 when scaled (i.e., the original value multiplies λ: $1 * 80\%$). Finally, the resulting similarity scores are summed and normalized to the range of 0 to 1 (i.e., a percentage value), which becomes the reported statistical matching likelihood.

Algorithm 1: VarSem's variable labeling.

```
Input  : variables (i.e., a collection of variables)
         target_word (i.e., the target word)
         λ (i.e., the attenuation rate for file paths)
Output : variables with likelihood values
1  Function: get_similarity(groups, target_word, λ):
2  max_avg ← 0
3  foreach word_array : groups do
4      simi_group ← 0
5      foreach word : word_array do
6          d ← similarity(word, target_word)
7          increase simi_group by d * λ
8      end
9      avg ← average(simi_group)
10     if avg > max_avg then
11         max_avg = avg
12     end
13 end
14 return max_avg
15 Function: NLP_main(variables, target_word):
16 foreach var : variables do
       /* for variable name.                  */
17     var_name ← get_var_name(var)
18     simi_var
         ← get_similarity(var_name, target_word, 1)
       /* for type name.                      */
19     type_name ← get_type_name(var)
20     simi_type
         ← get_similarity(type_name, target_word, 1)
       /* for function name.                  */
21     func_name ← get_func_name(var)
22     simi_func
         ← get_similarity(func_name, target_word, 1)
       /* for file path.                      */
23     path ← get_path(var)
24     simi_path
         ← get_similarity(path, target_word, 0.8)
25     var.likelihood
         ← (simi_var+simi_func+simi_type+simi_path)
26 end
```

6 Evaluation

Our evaluation seeks answers to the following questions:
Q1. Correctness: (a) Does VarSem infer variable usage semantics correctly? (b) How does combining the textual and context information of variables in VarSem programs affect the resulting correctness? (c) How does the correctness of VarSem compare to that of existing approaches?
Q2. Effectiveness: (a) How effective VarSem would be in helping software and security engineers to strengthen system security? (b) Are impact levels effective in controlling the correctness of inferring variable usage semantics?
Q3. Programming Effort: (a) How much programming effort does it take to express and execute VarSem programs? (b) How does this effort compare to using existing approaches?

6.1 Environmental Setup

The syntax of VarSem is defined by using the Scala embedded DSL facilities (Scala 2.12 and JDK 8). VarSem's preprocessing routines use Clang 6.0's *libtooling tool*. VarSem's textual information analysis uses Python 2.7, while its data-flow analysis routines use LLVM custom passes (LLVM public release 4.0). All experiments are performed on a workstation, running Ubuntu 16.04, 2.70GHz 4-core Intel i7-7500 CPU, with 15.4 GB memory.

6.2 Built-in Analysis Routines

VarSem comes with built-in analysis routines: an NLP-based textual information analysis and a data-flow analysis, whose implementations we discuss in turn next.

a) NLP-based textual information analysis. To compute the similarity between two words, we use Google's official pre-trained NLP model (Google News corpus with 3 billion running words [22]), and *Skip-Gram* algorithm implemented using *Word2vec*, a Google's word embedding tool [21].

b) Information-flow Analysis. An information flow analysis infers how a variable is used in terms of its flows from/to a specific function, its storing/reading into/from stable storage, and its comparison with other variables or constants. These inferencing tasks rely on traditional static analysis techniques, implemented as LLVM passes and Clang's analysis tools [11].

VarSem integrates these routines (i.e., a and b) in the same way as it would any new custom routines, thanks to its extensible architecture.

6.3 Evaluation Requirements

As a general solution for inferring variable usage semantics, it would be unrealistic to evaluate all possible scenarios in various domains. Hence, we constrained our evaluation to meet the following requirements: (1) **Domains:** The evaluation scenarios should be representative of current real-world domains that need variable usage semantics analysis. (2) **Beneficiaries:** The evaluation scenarios should be helpful for software and security engineers in strengthening the

Table 1. Project Information

Project	Author Info	PII	LoC	Var. Usage	VarSem Rules
(1) MimiPenguin [23]	8 contributors	login user info	574	user	`$var.Name ~ "user"` `$var.Type ~ STRING`
(2) emailaddr for PostgreSQL [19]	1 contributor	email address	246	email address	`$var.Name ~ "email address"` `$var.Value <=>`
(3) findmacs [15]	1 contributor	mac/ip address	515	mac/ip address	`$var.Name ~ "mac ip"` `$var.Type ~ STRING`
(4) emv-tools [7]	1 contributor	bank & credit info	9684 (1534)	smart card	`$var.Name ~ "smart card"` `$var.Value <~~ "scard_connect"`
(5) ssniper [47]	Technology Services Group (UIUC)	SSN	2421	ssn	`$var.Name ~ "ssn"` `$var.Value <~~ "represent_ssn"`
(6) phone number scanner [27]	1 contributor	phone number	381	phone number	`$var.Name ~ "phone number"` `$var.Value <~~ "findPhoneNumber"`
(7) wdpassport-utils [34]	7 contributors	password	1504	password	`$var.Name ~ "password"` `$var.Value <~~ "fgets_noecho"` `$var.Value <=>`
(8) Biometric Authentication [30]	KYLIN Information Technology Co., Ltd.	biometric info	18305(1679)	bio feature	`$var.Name ~ "biometric feature"` `$var.Value <~~ "bio_ops_get_feature_list"`

protection of sensitive data. (3) **Scope:** The evaluation subjects should be real systems, with realistic variable usage cases. To meet these requirements, our evaluation focuses on the domain of personally identifiable information (PII), all data that can be used to identify a specific individual [38]. By compromising PII, attackers can threaten people's security and privacy. If VarSem can assist in identifying PII variables, developers would be able to adequately protect these variables' security and privacy.

a) to evaluate correctness, we use 8 open-source projects that manipulate PII, including user information, email addresses, MAC/IP addresses, bank/credit information, social security numbers, phone numbers, passwords, and bio-metric information ("PII" column in Table 1). *b) to evaluate effectiveness*, we use Yubico PAM [56], an open-source system for authenticating users that was subjected to two real security attacks (details in § 6.4).

6.4 Evaluation Design

Correctness: In fact, PII variables differ greatly in their respective usage semantics, so each PII variable type requires a different VarSem program to identify it. As shown in Table 1, we wrote project-specific VarSem programs[2] that infer given variable usage semantics (the "Var. Usage" column) by following their rules (the "VarSem Rules" column). Recall the syntax of VarSem rules: `$var.Name ~ "a"` means the variable's name matches "a", `$var.Type ~ "b"` means the variable's type matches "b", `$var.Value <=>` means the variable's value is compared, and `$var.Value <~~ "foo"` means the variable's value must have flown from function "foo". As an example, consider project "(8) Biometric Authentication" in which VarSem infers those variables that store biometric features. The specified semantic category has two rules: (1) the variable's name matches "biometric feature" (textual information) and (2) its value should have flown from function `bio_ops_get_feature_list` (context information).

To evaluate correctness, we calculated the metrics of accuracy, precision, and recall. To obtain the ground truth, we recruited a volunteer (6+ years C/C++ experience) to manually find all variables that store the corresponding PII variables. To reduce the manual effort, we evaluated only the core modules of the larger projects 4 and 8 (their LoC metrics are parenthesized in the "LoC" column).

To answer question **Q1-a**, we executed each VarSem program to obtain the aforementioned metrics. To answer question **Q1-b**, we evaluated the cases in which semantic categories have only textual or context information. To answer question **Q1-c**, we evaluated the cases that use the simple string comparison function (i.e., `strcmp`[3]) to find target variables instead of VarSem's textual information analysis.

Without loss of generality, we assigned each rule the default impact value, while setting their likelihood thresholds to 0.6 and 0.8. That is, VarSem is to calculate the likelihood under the same impact level of each rule, and report only those variables whose likelihood of matching the rules is above 60% and 80% (we evaluated these two cases of likelihood independently).

Effectiveness: Our evaluation subject — Yubico PAM (four C source files, totalling 2486 lines of code) — has been subjected to two security attacks: one circumvented the authentication process through a particular password string [13], while the other leaked and tampered data through an unclosed file descriptor of the debug file [14]. Based on these two real-world attacks, we evaluate VarSem on the Yubico PAM codebase under two scenarios: (1) infer variables that store passwords, and (2) infer variables that store file descriptors of a debug file that has not been closed. By locating the variables in these scenarios, VarSem can help software and security engineers to defend against the attacks above.

We wrote VarSem programs for the "password" and "debug file descriptor" scenarios. The first program's semantic category has 2 rules: (1) the variable's name matches "password" (i.e., textual information) and (2) it should be compared (i.e., context information). The second program's semantic

[2]The complete VarSem programs appear in drive.google.com/file/d/1cAl5XmN791TXjp3T8CUgjicfCbr_N4jw/view?usp=sharing.

[3]it returns the variables whose name matches the search pattern exactly.

Table 2. Results of Correctness

Project	Init. Set	Prepro. Set	THLD	(a) Textual Info only			(b) Context Info only			(c) Textual + Context			strcmp		
				accuracy	precision	recall	accuracy	precision	recall	accuracy	precision	recall	accuracy	precision	recall
(1) MimiPenguin	115	89	0.6	95.5%	70.0%	87.5%	75.3%	26.7%	100%	**80.0%**	**30.1%**	**100%**	91.0%	NA	0%
			0.8	98.9%	100%	87.5%	75.3%	26.7%	100%	**98.9%**	**100%**	**87.5%**			
(2) emailaddr for PostgreSQL	31	31	0.6	25.8%	0%	0%	83.9%	100%	73.7%	**74.2%**	**82.4%**	**73.7%**	38.7%	NA	0%
			0.8	29.0%	0%	0%	83.9%	100%	73.7%	**45.1%**	**100%**	**10.5%**			
(3) findmacs	97	75	0.6	80.0%	23.1%	37.5%	77.3%	32.0%	100%	**82.7%**	**38.1%**	**100%**	93.3%	100%	37.5%
			0.8	86.7%	37.5%	37.5%	77.3%	32.0%	100%	**93.3%**	**100%**	**37.5%**			
(4) emv-tools	142	140	0.6	57.9%	7.9%	11.1%	95.0%	100%	74.1%	**95.0%**	**100%**	**74.1%**	80.7%	NA	0%
			0.8	79.3%	37.5%	11.1%	95.0%	100%	74.1%	**82.9%**	**100%**	**11.1%**			
(5) ssniper	620	323	0.6	96.6%	33.3%	10%	100%	100%	100%	**100%**	**100%**	**100%**	97.2%	100%	10%
			0.8	97.2%	100%	10%	100%	100%	100%	**97.2%**	**100%**	**10%**			
(6) phone number scanner	23	19	0.6	52.6%	45.5%	62.5%	84.2%	100%	62.5%	**84.2%**	**100%**	**62.5%**	57.8%	NA	0%
			0.8	52.6%	44.4%	50%	84.2%	100%	62.5%	**78.9%**	**100%**	**50%**			
(7) wdpassport-utils	205	190	0.6	95.8%	66.7%	22.2%	79.5%	17.4%	88.9%	**97.9%**	**85.7%**	**66.7%**	96.3%	100%	22.2%
			0.8	96.3%	100%	22.2%	96.7%	80.0%	44.4%	**95.8%**	**100%**	**11.1%**			
(8) Biometric Authentication	579	569	0.6	87.3%	0%	0%	99.6%	100%	50%	**99.3%**	**50%**	**50%**	99.3%	NA	0%
			0.8	92.4%	0%	0%	99.6%	100%	50%	**99.6%**	**100%**	**50%**			

category has 3 rules: (1) the variable's name matches "debug file" (textual information), (2) its type matches "FILE", and (3) it is not closed[4] (context information).

Similar to our evaluation of correctness, we also used accuracy, precision, and recall for evaluating effectiveness. Also, we used a volunteer to manually find all variables that store passwords and unclosed debug file descriptors to establish the ground truth.

To answer question **Q2-a**, we evaluated those VarSem programs whose semantic categories include both textual and context information. To answer question **Q2-b**, we evaluated the cases whose rules have different impact levels.

Programming Effort: To answer question **Q3-a**, we counted the uncommented lines of code (ULoC) of VarSem programs for each scenario of our evaluation subjects. To answer question **Q3-b**, we counted ULoC for the same scenarios as the control group: (1) manually execute the required NLP and data-flow analysis routines; (2) package these routines as libraries, and use the Unix shell to invoke them.

6.5 Results

Correctness: The accuracy, precision, and recall results appear in Table 2. VarSem programs start their execution with the following data collection procedure, which entails (1) extracting all program variables, producing the initial dataset, whose size is reported in the "Init Set" column; (2) pre-processing the initial dataset to eliminate duplicates, producing the pre-processed dataset, whose size is reported in the "Prepro. Set" column.

Then, VarSem's execution differs based on the provided rules. For each of the 8 evaluation subjects, we provided a VarSem program containing both the textual and context information rules. For each program, we experimented with the likelihood thresholds of 0.6 and 0.8 ("THLD" column). The results reported in the "Textual+Context" column performed satisfactorily in *accuracy* (13 out of 16 times >= 80%), *precision* (13 out of 16 times > 80%), and passably in *recall* (8 out of 16 times > 60%). That is, VarSem infers variable usage semantics mostly correctly, with some PII variables missed (i.e., false negatives). These high false negatives are positively correlated with the specified likelihood thresholds: the larger the threshold, the less likely would VarSem designate variables as PII, with more *true* PII variables missed and smaller recall. Also, the larger the threshold, the higher the precision value. That is, when the threshold is large (0.8 in our case), VarSem indeed misses some *true* PII variables, but it correctly excludes many non-PII variables, so the false positives rate decreases, with the precision increasing.

Also, we modified its VarSem program to produce 2 variants: keep only the textual information rules (case-a, the "Textual Info only" column) and keep only the context information rules (case-b, the "Context Info only" column). Then we compared VarSem's output for these two variants with that of the original program, which includes both the textual and context information rules (case-c, the "Textual + Context" column). Case-c outperformed (or performed equal to) the other two cases (14 out of 16 times better than or equal to case-a while 9 out of 16 times better than or equal to case-b. 5 out of 8 times better than or equal to both case-a and b for the threshold 0.6). That is, considering both the textual and context information can increase VarSem's performance. However, if the performance of case-a or b is poor, case-c's performance deteriorates as well. For example, in the project-2 "emailaddr for PostgreSQL", because its case-a generates unsatisfactory results (accuracy < 30%, precision and recall = 0%), case-c performs no better than case-b. Recall that we assigned the same impact levels to all rules. By adjusting the rules' impact levels to fit specific variable usage semantics, the performance of case-c can be improved (we evaluate how impact levels work below).

We also evaluated the performance of a simple string comparison function (the "strcmp" column) to process textual information rules. Overall, its performance is worse than any of the cases a, b, and c above. In five out of eight projects (i.e., projects 1, 2, 4, 6, and 8), `strcmp`'s inference results are

[4]With this attack fixed and all file descriptors correctly closed, we modified the file descriptors of debug file to remain open for our evaluation.

useless (i.e., the precision and recall values are either NA—the number of true/false positives is "0" or 0%—the number of true positives is "0"). This result reveals that the symbolic names of the PII variables in these projects are not descriptive of their usage semantics. For example, in the project-4 "phone number scanner", the variables that store phone numbers are named as "inStr" and "outputStr" rather than anything resembling "phone_number." However, because of VarSem's NLP-based analysis for rules of textual information, VarSem's results (i.e., "Textual Info only" column) for these projects always outperform (or perform equal to) the strategy of comparing strings for equality.

Table 3. Results of Effectiveness

Var. Usage	Rules	Impact Lv.	THLD	Textual + Context		
				accuracy	precision	recall
password	(a)`$var.Name~"password"` (b)`$var.Value<=>`	a<b (a=5, b=10)	0.6	40.60%	4.20%	100%
			0.8	86.90%	15.80%	92.30%
		a=b (a,b=10)	0.6	60.60%	5.80%	92.30%
			0.8	95.60%	32%	61.50%
		a>b (a=10, b=5)	0.6	86.90%	15.80%	92.30%
			0.8	97.60%	53.80%	53.80%
debug file	(a)`$var.Name~"debug file"` (b)`$var.Type~FILE` (c)`$var.Value!~~>"fclose"`	a<b=c (a=5, b,c=10)	0.6	98.20%	100%	18.20%
			0.8	97.80%	NA	0%
		a=b=c (a,b,c=10)	0.6	100%	100%	100%
			0.8	97.80%	NA	0%
		a>b=c (a=10, b,c=5)	0.6	100%	100%	100%
			0.8	97.80%	NA	0%

Effectiveness: Table 3 shows the effectiveness results of inferring the variable usage semantics scenarios, described as "password" and "debug file" (the "Var. Usage" column). Overall, VarSem performed satisfactorily in inferring the variables that store unclosed debug file descriptors (100% accuracy, precision, and recall in the best case), and passably in inferring the variables that store passwords (97.6% accuracy, 53.8% precision, and 53.8% recall in the best case). Hence, based on the returned variable lists, developers can directly locate the unclosed debug file descriptors and reduce the manual effort required to locate password variables. VarSem performed less effectively in the "password" scenario, as it was given more general rules. Indeed, in the "Yubico Pam" project, in addition to password variables, other variables are semantically related to "password" and are compared for equality. In contrast, only those variables that store unclosed debug file descriptors have a symbolic name related to "debug file", the type matching "FILE", and the value never flowing to function `fclose`.

When evaluating the influence of impact levels in the "Yubico Pam" subject, increasing the impact level of textual information rules does improve VarSem's performance, as the given textual information rules are more descriptive of the target semantics than the context information rules. That is, many variables compared for equality may not store passwords (the "password" case), and many variables never flowing to `fclose` function may not store debug file descriptors. Since the likelihood values are weighted by their impact levels, the textual information with impact levels higher than those of the context information would contribute more to the final results, resulting in better accuracy/precision/recall.

Programming Effort: Table 4 shows the evaluated effort as hand-written ULoC. For all evaluated subjects, it took only ≈6 ULoC to write a project-specific VarSem program. However, to accomplish the same inferencing task without VarSem took ≈2,500 ULoC of analysis routines written in C and Python. Even if these routines were packaged as libraries, invoking them would take ≈16 ULoC of Unix shell scripts. Without VarSem, developers would need to possess specialized expertise: (1) a familiarity with all relevant libraries/routines and their interactions, and (2) an understanding of the output format of each routine involved. In contrast, with VarSem, developers specify a semantic category using high-level declarative rules, thus lowering the programming effort.

Table 4. Programming Efforts of VarSem

Languages	Inference Tasks
VarSem	~6 (ULoC)
Unix shell	~16 (ULoC)
All relevant routines	2442 (ULoC)

6.6 Discussion

Rules. As the effectiveness results demonstrate (§ 6.5), specific rules outperform general rules. Although easier to express, general rules (e.g., a variable type is a string) match too many irrelevant variables, which more specific rules can effectively filter out.

Thresholds. As the correctness results demonstrate (§ 6.5), precision and recall can be improved by modifying thresholds: in general, a high threshold increases precision, while a low threshold increases recall. This behavior is due to higher thresholds causing fewer reported variables, thus decreasing false positives. In contrast, lower thresholds cause more reported variables, thus decreasing false negatives. With the thresholds being in the range of [0,1], choosing the thresholds of 0.6 and 0.8 reports the variables whose likelihoods of matching the rules are above 60% and 80%, respectively.

Impact levels. It would be unrealistic to expect programmers to follow the prescribed naming convention all the time. Hence, a good practice for VarSem users is to pre-check the presence of and adherence to a naming convention. Depending on the findings, the VarSem performance can be improved by adjusting rules' impact levels: in the presence of unsystematic naming practices, reduce the impact levels of textual information rules, and vice versa.

Runtime. VarSem programs execute in time proportional to the size of target codebases and the complexity of rules: it takes longer to analyze larger projects with complex rules. In our evaluation, the VarSem executes within an acceptable time: less than 1 minute (projects 1,2,3,6), less than 3 minutes (project 4), and less than 6 minutes (projects 5,7,8). The most time-consuming tasks are the NLP-based textual information analysis and data-flow analysis.

Applicability. Our built-in data-flow analysis routines are LLVM-based, thus limiting their applicability to C/C++ projects. Hence, our evaluation subjects are written in C/C++. However, VarSem's applicability can be extended to other languages thanks to its containerized deployment model (§ 4.2). When it comes to VarSem's syntax, it would need to change to accommodate how the target language expresses variables. For example, in dynamically typed languages, variable types are determined and can change at runtime, rendering VarSem's static type-related rules (e.g., `$var.Type ~ "int"`) inapplicable. Nevertheless, due to VarSem's extensible architecture and meta-programming support, developers can introduce rules with suitable operator/operands and analysis routines that can handle new analysis scenarios (e.g., in a dynamically typed language, the type can be bound to numeric values: `$var.DynType ~ "numeric"`)

Threats to validity & Limitations. The internal validity is threatened by comparing VarSem's results with those of `strcmp` rather than of regular expressions. Two observations mitigate this threat: (1) `strcmp` provides a standard comparison baseline; (2) as discussed in § 2.2, we assume user familiarity only with variable usage semantics (VUS), not expecting them to be able to list all variations of symbolic names. For example, a user would be looking for the "password" VUS, being unaware which symbolic names ("pwd", "p_wd", "p_w_d", etc.) to enumerate with a regular expression.

The external validity is threatened by evaluating only with eight third-party C/C++ subjects (see § 6.3). Although covering various PII variable scenarios, these projects cannot represent all possible scenarios. Further, our findings might not generalize to other languages, and only additional studies can mitigate this threat. We plan to open-source VarSem and our experiments, so others could conduct studies with different subjects and settings.

For limitations, although VarSem outperforms `strcmp` in most cases, `strcmp` can search projects with highly descriptive variable names with acceptable accuracy/precision and passable recall (i.e., rows (3)(5)(7) and "strcmp" column in Table2). As VarSem typically executes in seconds or minutes at worst, `strcmp` can execute in a second or less, achieving comparable accuracy/precision/recall. In addition, extending VarSem to support new operators/operands can require changing the runtime, incurring an additional programming effort, amortized only by subsequent uses of the extension. However, with sufficient domain knowledge, VarSem users should be able to select the most suitable analysis algorithms and meaningfully extend VarSem.

7 Related Work

Understanding program semantics. Weimer et al. mine for code specifications beneficial for debugging a given codebases [53]. Høst et al. use data mining to extract rules from existing Java applications to identify unusual and unacceptable method names [25]. Mishne et al. apply static analysis-based semantic search over partial code snippets to understand API usage [36]. Allamanis et al. learn, via an n-gram language model, a code snippet's coding style to recommend identifier names and formatting conventions [1, 2]. Raychev et al. use structured support vector machines (SSVM) to predict identifiers names and type annotations in JavaScript projects [41]. Alon et al. apply deep learning to learn code embeddings, used to predict method names [5]. Rice et al.'s algorithm detects whether a function call is passed correct parameters from an identifier's name [42]. Sridhara et al.'s algorithm generates documents for Java methods by selecting critical code statements and expressing them as natural language phrases [45]. Buse et al.'s descriptive model assigns human-annotated readability scores to source code features to measure code readability [8]. These solutions are intended for specific applications scenarios and rely on dissimilar theories and algorithms. They also focus on the source code's context information, without considering it in concert with textual information. In contrast, VarSem considers both textual and context information to infer variable semantics. Further, VarSem can incorporate the approaches above due to its extensible architecture.

Software Querying Languages. Dyer et al.'s DSL extracts statistical information from large software repositories [16]. Martin et al.'s PQL is a program query language for searching specific source code patterns [35]. Chen et al.'s VFQL is a program query language for expressing and searching for code defects through value flow graphs [9]. Urma et al. describe Wiggle, a querying system that uses Neo4j's Cypher language to search source code with user-specified textual or context properties [49]. Cohen et al. design a language for querying specific patterns in Java source code [12]. Eidorff et al.'s type-based approach finds and replaces the problematic dates in source code to solve the Year 2000 problem [17, 18]. Some online tools and IDE plugins locate given code patterns [6, 39]. However, none of these languages focus on inferring variable semantics. In contrast, VarSem reifies the novel *variable usage semantics analysis*, understanding variable semantics rather than that of the entire program.

8 Conclusion

This paper has presented *variable usage semantics analysis*, reified as a DSL—VarSem—with a novel NLP-based analysis. VarSem features declarative and customizable rules bound to analysis routines for inferring both textual and context variable information. By reducing the effort of variable usage semantics analysis, VarSem can benefit both developers and security analysts.

Acknowledgements

The authors would like to thank the anonymous reviewers, whose insightful comments helped improve the technical content of this paper. The research is supported by the NSF through the grants #1650540 and #1717065.

References

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 281–293.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 38–49.

[3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*. 2091–2100.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices* 53, 4 (2018), 404–419.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[6] antlersoft. 2011. Browse-by-Query. http://browsebyquery.sourceforge.net/.

[7] Dmitry Baryshkov. 2019. Tools to work with EMV bank cards. https://github.com/lumag/emv-tools.

[8] Raymond PL Buse and Westley R Weimer. 2008. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 121–130.

[9] Guang Chen, Yuexing Wang, Min Zhou, and Jiaguang Sun. 2019. VFQL: combinational static analysis as query language. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 378–381.

[10] Yue Chen, Mustakimur Khandaker, and Zhi Wang. 2017. Pinpointing vulnerabilities. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*. 334–345.

[11] Clang Front End for LLVM Developers. 2019. Clang Static Analyzer. https://clang-analyzer.llvm.org/.

[12] Tal Cohen, Joseph Gil, and Itay Maman. 2006. JTL: the Java tools language. *ACM SIGPLAN Notices* 41, 10 (2006), 89–108.

[13] CVE site. 2011. CVE-2011-4120. https://cvesite.com/cves/CVE-2011-4120.

[14] CVE site. 2019. CVE-2019-12210. https://cvesite.com/cves/CVE-2019-12210.

[15] drkblog. 2018. findmacs. https://github.com/drkblog/findmacs.

[16] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 422–431.

[17] Peter Harry Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, Morten Heine Sørensen, and Mads Tofte. 1999. AnnoDomini: from type theory to Year 2000 conversion tool. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1–14.

[18] Peter Harry Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, Morten Heine B Sørensen, and Mads Tofte. 1999. AnnoDomini in practice: A type-theoretic approach to the year 2000 problem. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 6–13.

[19] Peter Eisentraut. 2015. emailaddr type for PostgreSQL. https://github.com/petere/pgemailaddr.

[20] Edward M Gellenbeck and Curtis R Cook. 1991. An investigation of procedure and variable names as beacons during program comprehension. In *Empirical studies of programmers: Fourth workshop*. Ablex Publishing, Norwood, NJ, 65–81.

[21] Google. 2019. word2vec. https://code.google.com/archive/p/word2vec/.

[22] Google. 2019. word2vec-GoogleNews-vectors. https://github.com/mmihaltz/word2vec-GoogleNews-vectors.

[23] Hunter Gregal. 2019. MimiPenguin 2.0. https://github.com/huntergregal/mimipenguin.

[24] Cay S Horstmann. 2012. *Scala for the Impatient*. Pearson Education.

[25] Einar W Høst and Bjarte M Østvold. 2009. Debugging method names. In *European Conference on Object-Oriented Programming*. Springer, 294–317.

[26] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. {SUPOR}: Precise and Scalable Sensitive User Input Detection for Android Apps. In *24th USENIX Security Symposium (USENIX Security 15)*. 977–992.

[27] J. Karau. 2014. phone number scanner. https://github.com/wittycoder/phone_number_scanner.

[28] Lin Jiang, Hui Liu, and He Jiang. 2019. Machine Learning Based Recommendation of Method Names: How Far are We. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 602–614.

[29] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. 2018. Discovering software vulnerabilities using data-flow analysis and machine learning. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*. 1–10.

[30] KYLIN Information Technology Co., Ltd. 2019. Biometric Authentication. https://github.com/ukui/biometric-authentication.

[31] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 3–12.

[32] Xing Liu, Jiqiang Liu, Wei Wang, Yongzhong He, and Xiangliang Zhang. 2018. Discovering and understanding Android sensor usage behaviors with data flow analysis. *World Wide Web* 21, 1 (2018), 105–126.

[33] llvm-admin team. 2019. The LLVM Compiler Infrastructure. https://llvm.org/.

[34] Kenny MacDermid. 2016. wdpassport-utils. https://github.com/KenMacD/wdpassport-utils.

[35] Michael Martin, Benjamin Livshits, and Monica S. Lam. 2005. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 365–383. https://doi.org/10.1145/1094811.1094840

[36] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 997–1016.

[37] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. 2015. Uipicker: User-input privacy identification in mobile applications. In *24th USENIX Security Symposium (USENIX Security 15)*. 993–1008.

[38] Arvind Narayanan and Vitaly Shmatikov. 2010. Myths and fallacies of" personally identifiable information". *Commun. ACM* 53, 6 (2010), 24–26.

[39] NetBeans. 2012. Jackpot. http://wiki.netbeans.org/Jackpot.

[40] Son Nguyen, Tien Nguyen, Yi Li, and Shaohua Wang. 2019. Combining Program Analysis and Statistical Language Model for Code Statement Completion. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 710–721.

[41] Veselin Raychev, Martin Vechev, and Andreas Krause. 2019. Predicting program properties from'big code'. *Commun. ACM* 62, 3 (2019), 99–107.

[42] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting argument selection defects. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–22.

[43] Luciano Sampaio and Alessandro Garcia. 2016. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software* 113 (2016), 337–361.

[44] Paul M Schwartz and Daniel J Solove. 2011. The PII problem: Privacy and a new concept of personally identifiable information. *NYUL rev.* 86 (2011), 1814.

[45] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 43–52.

[46] The Clang Team. 2020. Matching the Clang AST. https://clang.llvm.org/docs/LibASTMatchers.html.

[47] Technology Services Group, University of Illinois at Urbana-Champaign. 2014. Ssniper Social Security Scanner for Linux. https://github.com/racooper/ssniper.

[48] The Clang Team. 2019. LibTooling. https://clang.llvm.org/docs/LibTooling.html.

[49] Raoul-Gabriel Urma and Alan Mycroft. 2015. Source-code queries with graph databases—with application to programming language usage and evolution. *Science of Computer Programming* 97 (2015), 127–134.

[50] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 683–693.

[51] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[52] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2018. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of Android apps. *ACM Transactions on Privacy and Security (TOPS)* 21, 3 (2018), 1–32.

[53] Westley Weimer and George C Necula. 2005. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 461–476.

[54] Ian H Witten, Eibe Frank, and Mark A Hall. 2005. Practical machine learning tools and techniques. *Morgan Kaufmann* (2005), 578.

[55] Ludwig Wittgenstein. 2009. *Philosophical investigations*. John Wiley & Sons.

[56] Yubico Company. 2019. Yubico PAM module. https://developers.yubico.com/yubico-pam/.

[57] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. 2019. Recdroid: automatically reproducing Android application crashes from bug reports. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 128–139.