Towards Automated Checking of Input Data Usage with Facebook Infer

Rui Ferreira[1]

INESC-ID & IST, University of Lisbon, Portugal
rui.s.ferreira@ist.utl.pt

Abstract. In modern, data-intensive applications the use and modification of input data is very frequent. During the various transformations that the data suffers, parts can remain unused due to programming errors. These errors can be hard to detect and locate due to the high amount of data transformations, and can have real-life consequences.
In this paper we propose the implementation of a recent analysis proposed by Urban and Müller as an analysis for Facebook Infer, a popular static program analyser for Java, C, C++ and Objective C. We show that our prototype can effectively identify input data usage errors in the same benchmark used by related work.

Keywords: Data Usage · Static Analysis · Software Reliability

1 Introduction

Data science applications normally deal with considerable amounts of input data that go through long pipelines of processes such as data acquisition, data cleansing, and data preparation. As data is processed, programming errors can cause parts of the input data to remain unused, leading to incorrect or unexpected outputs. The presence of such errors is difficult to detect because the errors are usually subtle and do not raise any compilation errors or warnings. Moreover, the results produced by these applications are usually plausible. The large amount of data transformations that can occur during program execution also hinders detection of these errors.

As pointed out by Urban and Müller [8], a real-world example that shows how input data usage errors can have nefarious societal consequences is that of the paper "Growth in a Time of Debt", published by economists Reinhart and Rogoff in 2010. This paper analyzed the correlation between economic growth, inflation and external debt, using data from forty four countries across two hundred years. However, it was later found by economists Herndon, Ash and Pollin that data from five countries was unintentionally excluded from the analysis due to a *programming error* [5]. It is worth noting that critics of this paper believe that it has led to unnecessary adoption of austerity policies in several countries [6].

To address this, Urban and Müller recently proposed the first static analysis, based on abstract interpretation, capable of detecting input data usage errors [8]. They implement their analysis in a research prototype, Lyra, that supports a

subset of Python. In this paper, we propose an implementation of their analysis for Facebook Infer, an industrial-strength static program analysis tool for Java, C, C++, and Objective C [2]. Our implementation is on Infer's intermediate language *SIL* so that the analysis can be potentially used with any language supported by Infer. We have manually converted Lyra's benchmark to Java and analyzed them using our implementation. The results obtained coincide with those obtained for Lyra. The implementation and benchmark are available at:

https://github.com/Rui1995/DataUsageCheck-FBInfer

2 Detecting Input Usage Errors

In this section, we describe the approach that we implemented to detect input data usage errors. It was originally developed by Urban and Müller [8] and implemented in a tool called Lyra[1]. The analysis is based on syntactic dependencies between variables and abstract interpretation. The analysis works backwards, starting the analysis in the last line of code of a program.

The idea behind using the syntactic dependency between variables is to examine the possible interactions between variables and give them a classification, so that in the end of the analysis, the classification of the input variables determines if those have been used.

The possible classifications in Urban and Müller's analysis are: "*Used*" (U), "*Not Used*" (N), "*Overwritten*" (W) and "*Below*" (B). "*Overwritten*" signifies a variable that was previously used but was re-written, and "*Below*" a variable that was used at a lower nesting level. Our work introduces two additional classifications: "*Below (Used before Push)*" (BU) and "*Not Used (Overwritten before Push)*" (NW). We describe below the two new possible classifications.

To simplify presentation, in this section we use as abstract domain a map that associates the name of each variable in the program with its classification and its location (for error message purposes). In Section 3, we discuss how we extended the abstract domain to simplify the implementation of the analysis. In the beginning of the program analysis the abstract domain consists of every input variable being classified as "N", and the output variables "U". The additional variables used during the program are added to the domain when they are first encountered by the analysis.

Taking into account the classifications of all variables inside an instruction, the interaction between variables and the type of instruction being executed, the analysis will update each variable classification according to the transfer function Θ_Q that transforms an abstract state q as follows:

[1] Lyra's webpage: https://caterinaurban.github.io/project/lyra

$$
\begin{aligned}
\Theta_Q[\![skip]\!](q) &\stackrel{\text{def}}{=} q \\
\Theta_Q[\![x = e]\!](q) &\stackrel{\text{def}}{=} \text{ASSIGN}[\![x = e]\!](q) \\
\Theta_Q[\![\text{if } b : s_1 \text{ else: } s_2]\!](q) &\stackrel{\text{def}}{=} \text{POP} \circ \text{FILTER}[\![b]\!] \circ \Theta_Q[\![s_1]\!] \circ \text{PUSH}(q) \\
&\quad \sqcup_Q \text{POP} \circ \text{FILTER}[\![b]\!] \circ \Theta_Q[\![s_2]\!] \circ \text{PUSH}(q) \\
\Theta_Q[\![\text{while } b : s]\!](q) &\stackrel{\text{def}}{=} \text{lfp}_q^{\sqsubseteq_Q} \Theta_Q[\![\text{if } b : s \text{ else : skip}]\!](q) \\
\Theta_Q[\![s_1 \ s_2]\!](q) &\stackrel{\text{def}}{=} \Theta_Q[\![s_1]\!] \circ \Theta_Q[\![s_2]\!](q)
\end{aligned}
$$

The rule for assignments stipulates that a variable is considered "U" if it is utilized inside an assignment to another variable already considered "U", "B" or "BU". In that case, the second variable classification changes to "W" unless it is also present in the first variable, in which case it remains with the same classification. More formally, it is defined as:

$$
ASSIGN[\![x = e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} W & y = x \wedge y \notin \text{VARS}(e) \wedge m(x) \in \{U, B, BU\} \\ U & y \in \text{VARS}(e) \wedge m(x) \in \{U, B, BU\} \\ m(y) & \text{otherwise} \end{cases}
$$

Note that if the variable being assigned does not have one of these classifications, then both variables remain unchanged.

Another way a variable can be classified as "U" is if it appears in the Boolean condition of a statement that uses or modifies another variable already classified as "U". This is captured in the definition of the $FILTER$ function:

$$
FILTER[\![e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} U & y \in \text{VARS}(e) \wedge \exists y \in X : m(y) \in \{U, W\} \\ m(y) & \text{otherwise} \end{cases}
$$

Informally, the $FILTER$ function searches the domain for variables that are classified as "U" or "W". If it finds any variables in those conditions, then the variables present on the conditional statement get classified as "U". Otherwise, the current classifications remain unchanged.

Single map. In order to facilitate our implementation, we deviate slightly from Urban and Müller and, instead of keeping a stack of maps, we use a single map that is updated throughout the analysis.

This changes the possible classifications of a variable, introducing the two new classifications "BU" and "NW", as discussed above. The classification "BU" is used when a variable that was classified as "U" suffers a $PUSH$. The classification "NW" is used when a variable that was classified as "W" suffers a $PUSH$.

We do this because the array introduced by Urban and Müller is only used for the $PUSH$ and POP functions, so the analysis ends up only using the last

two classifications of a variable. This change saves the program from having to go through and modify an array for every variable.

Our version of the $PUSH$ function is defined as:

$$PUSH(m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} BU & m(y) \in \{U\} \\ NW & m(y) \in \{W\} \\ m(y) & \text{otherwise} \end{cases}$$

In words, the $PUSH$ function changes the classification of all variables "U" or "W" so that the $FILTER$ function can capture any new variables being classified as "U" or "W" inside a conditional set of instructions.

The POP function reverses the changes that $PUSH$ does, but only to the variables that have not been changed since. It is defined as:

$$POP(m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} U & m(y) \in \{BU\} \\ W & m(y) \in \{NW\} \\ m(y) & \text{otherwise} \end{cases}$$

Note that the POP function is used to return the variable classifications to a regular state, after an analysis of a conditional set of instructions finishes.

At the end of the analysis the abstract domain is searched for input variables whose classification is "N". If there are any variables with this classification, it means that an input data usage error is present.

3 Implementation

We implemented the analysis discussed in the previous Section as a checker for Facebook's Infer[2], which is a static analysis tool that currently supports Java, C, C++ and C-Objective [2]. The implementation is written in OCaml.

Infer provides a framework, Infer.AI[3], that enables the creation of analyses based on abstract interpretation. As Infer translates the code analyzed into its Intermediate Language *SIL*, which is used in the analysis instead of the original code, one only needs to create one checker for all the languages supported by Infer. Moreover, the checker will be readily available to any new language supported by Infer.

Infer's workflow has two main phases: a) *Capture Phase*: In this phase Infer translates the files under analysis into Infer's internal intermediate language, *SIL*; b) *Analysis Phase*: This is the phase where each function and/or method is analyzed.

During the analysis phase, Infer uses *Pure Variables*, which only appear in the *SIL* representation, and *Program Variables*, which are the variables that appear in the source program under analysis. This means our analysis needs to register

[2] Facebook Infer: https://fbinfer.com.

[3] Building checkers with the Infer.AI framework: https://fbinfer.com/docs/absint-framework.

the relations between them in order to understand, for example, which *Program Variable* is being used in an assignment that has *Pure Variables*.

Infer separates instructions in the *SIL* intermediate language into five different ones: *Load* (loads a value from the heap), *Store* (stores a value in the heap), *Prune* (prunes the state based on an expression), *Call* (call to a function) and *Metadata* (additional information about the program). For example, an assignment such as `a = b` in the original program is translated to *SIL* as a *Load* instruction for `b` and a *Store* instruction for `a`. During the analysis, each *SIL*'s instruction is evaluated in order to detect for instructions considered by the transfer functions.

The abstract domain used in our implementation is different from the one of Lyra, being composed of three main structures:

- *Map* is the main structure, as described in Section 2, which holds the information about all the variables in the program regarding its current classification and location.
- *MapArray* is used when there is an assignment that accesses an array (e.g., `a=b[c]`). This is needed because the variable being assigned (`a`) is only declared after the assignment, and its classification is needed to determine the classification of the other variables being used (`b,c`). This occurs because the analysis is backwards.
- *MapCall* is used when there is an assignment that contains a *Call* instruction. This is needed because, given the backwards nature of the analysis, the variable being assigned appears before the assignment and before the *Call* instruction. This map registers the variable being assigned, and the *Pure Variables* being assigned to that variable through the program. This ensures that when the assignment arises, the analysis knows what variable is being used.

Additionally, a structure called *cfg_node* (*node*) is also present in the abstract domain, holding information about the node currently being analyzed, that is used to detect nesting level changes in the following instruction.

The analysis will update each variable classification according to the transfer function Θ_Q that we define in this Section. The function is defined by cases on *SIL* instructions and it transforms an abstract state q into another. Each abstract state is tuple $(m, c, a, node)$, where m is Map that contains the classification of each variable, c is $MapCall$ that contains the relation between variables in a $Call$ instruction, a is $MapArray$ that contains the relation between variables in an assignment with access to an array, and $node$ is cfg_node that contains the information needed to detect changes in nesting levels.

Due to space limitations only the high level Transfer Functions are going to be showed next.

3.1 Load

The *Load* instruction (Fig 1) loads a value (carried inside the expression *"e"*) into an Identifier (*"id"*).

$$\Theta_Q[\![\text{Load } id : e]\!](q) \stackrel{\text{def}}{=}$$
$$\begin{cases} \text{ASSIGN}[\![\text{KOV}[\![id]\!](c) = e]\!] \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & e \in \text{LINDEX} \\ \text{LOAD\_PVAR}[\![id, e]\!](q) \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & e \in \text{PVAR} \wedge id \in a \\ \text{LOAD\_GENERAL}[\![id, e]\!](q) \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & \text{otherwise} \end{cases}$$

Fig. 1. Load Transfer Function

In every *Load* instruction the analysis starts by executing the function CK (CHECK) that verifies if both variables are present in *Map (m)*, and if they are not it adds them to *Map (m)* with the classification "N". This also happens with *Store* and *Call* instructions. The *Load* instruction has different approaches depending on the type of the variables "*id*" and "*e*":

Case 1. If the expression "*e*" is an *Array Index Offset* (LINDEX) the analysis performs an assignment between the key matching the "*id*" value in *MapCall (c)*, and the variable "*e*"; this assignment is achieved using the function $ASSIGN$. These instructions capture an instruction `a=b[c]` that accesses an array during a *Load* instruction.

Case 2. If the variable inside expression "*e*" is a *program variable* and "*id*" is in *MapArray (a)*, then, depending if the variable "*id*" is one of the keys of *MapArray (a)*, either executes the $ASSIGN$ function or updates the value from "*id*" into "*e*" in *MapArray (a)*.

Case 3. If none of the previous conditions are verified then, if the length of expression "*e*" is not 1, the program updates the classification of "*e*" with the same classification of "*id*" (transferring the classification that the pure variable "*id*" carried into the program variable that it represents "*e*"). If the length of expression "*e*" is 1 then it also updates the value that matched "*id*" to "*e*".

3.2 Store

$$\Theta_Q[\![\text{Store } e1 : e2]\!](q) \stackrel{\text{def}}{=}$$
$$\begin{cases} \text{ASSIGN\_HASH}[\![e1, e2]\!](a) \circ \text{CK}[\![e1]\!] \circ \text{CK}[\![e2]\!](m) & e1 \text{ is LINDEX} \wedge e2 \text{ is VAR} \wedge e1 \notin m \wedge e2 \notin m \\ \text{STORE\_RETURN}[\![e1, e2]\!](q) \circ \text{CK}[\![e1]\!] \circ \text{CK}[\![e2]\!](m) & e1 \text{ is LVAR} \wedge e1 \text{ is RETURN} \\ \text{STORE\_LVAR}[\![e1, e2]\!](q) \circ \text{CK}[\![e1]\!] \circ \text{CK}[\![e2]\!](m) & e1 \text{ is LVAR} \wedge e2 \text{ is VAR} \wedge e1 \text{ not RETURN} \\ \text{STORE\_GENERAL}[\![e1, e2]\!](q) \circ \text{CK}[\![e1]\!] \circ \text{CK}[\![e2]\!](m) & \text{otherwise} \end{cases}$$

Fig. 2. Store Transfer Function

The *Store* instruction (Fig 2) stores the value on an expression *"e2"* into another expression *"e1"*. Just like *Load*, the analysis starts by executing a CK

(CHECK) that verifies if both variables are present in *Map (m)*, and if they are not it adds them to *Map (m)* with the classification "N". The *Store* instruction has four cases that are described below.

Case 1. If expression "*e1*" is an *Array Index Offset* (LINDEX), then this is the first instruction of an assign with an array on the left side and a Pure Variable "*e2*" on the right. When this is detected the analysis adds to *MapArray (a)* the first variable in expression "*e1*" as key and the remaining variables of "*e1*" and "*e2*" as values.

Case 2. If during the *Store* instruction the expression "*e1*" is a *return* expression ("*return*" is considered a *Program Variables* (Lvar)) then the analysis changes the classification of all variables contained in the expression "*e2*" to "*U*". If the node (that represents a single Java instruction) has two predecessors ("*PREDS*") then it represents that the instruction before is the beginning of a conditional method body. If this is the case, then the analysis will also apply the function $PUSH$.

Case 3. When a value is stored from a *Pure Variable* into a *Program Variable*, first it changes the value in *MapCall (c)* associated with "*e1*" and replaces it with "*e2*". Next, the analysis checks if "*e2*" is associated with a key in *MapCall (c)* and if it is not, then it adds "*e1*" as a key and "*e2*" as a value. Finally the program changes the classification of "*e2*" into the same classification of "*e1*". Just like in the previous case, if the previous instruction has two predecessors ("*PREDS*") the analysis executes a $PUSH$ instruction.

Case 4. If none of the previous conditions are true, then the analysis updates the value in *MapCall (c)* associated with "*e1*" and replaces it with "*e2*". After that, the analysis executes an assignment between "*e1*" and "*e2*". Once again the analysis performs a $PUSH$ instruction if the previous instruction has two predecessors ("*PREDS*").

3.3 Call

The *Call* instruction (Fig 3) represents a `[ret_id = e_fun(arg_ts)]`instruction, that calls a function in *"e_fun"*, whose arguments are *"arg_ts"*, and stores its value in *"ret_id"*.

As before, the analysis checks a series of conditions, starting with the instruction CK (CHECK). There are four cases, which we describe below.

Case 1. If "*e_fun*" is an object then the analysis verifies if a key associated with the value *ret_id* in *MapCall (c)* exists, and if that key is classified as "U" changes the classification of all variables inside the function called into "U". If the key is not classified as "U" then associates the first variable in *arg_ts* (as key) with the other variables in *arg_ts* (as values) in *MapCall (c)*. If none of the above conditions verify, then the analysis executes an $ASSIGN$ between the key associated with *ret_id* and the variables of *arg_ts*.

$$\Theta_Q[\![\text{Call } ret\_id, e\_fun, arg\_ts]\!](q) \stackrel{\text{def}}{=}$$
$$\begin{cases} \text{CALL\_OBJECT}[\![ret\_id, arg\_ts]\!](q) \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & e\_fun \text{ is Object} \\ \text{CALL\_ARRAY}[\![ret\_id, arg\_ts]\!](q) \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & e\_fun \text{ is ArrayList} \\ \text{CALL\_PRINT}[\![ret\_id, arg\_ts]\!](q) \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & e\_fun \text{ is PrintStream} \wedge ret\_id \text{ is IRVAR} \\ \text{CALL\_GENERAL}[\![ret\_id, arg\_ts]\!](q) \circ \text{CK}[\![id]\!] \circ \text{CK}[\![e]\!](m) & \text{otherwise} \end{cases}$$

Fig. 3. Call Transfer Function

Case 2. If the analysis verifies that *e_fun* is an *Arraylist* then the analysis verifies if a key associated with the value *ret_id* in *MapCall (c)* exists and if that key has a classification of "U", and if confirmed changes the classification of all variables inside the function called into "U". If not then it associates the first variable in *arg_ts* (as key) with the other variables in *arg_ts* (as values) in *MapCall (c)*.

Case 3. If the analysis verifies that *e_fun* is an *Printstream* then the analysis turns the classification of the variables in *arg_ts* to "U" and, if the analysis detects that the next instruction has 2 predecessors then the analysis also performs a $PUSH$.

Case 4. If none of the previous clauses are verified for $CALL$ then the analysis changes the value in *MapCall (c)* associated with "*ret_id*" and replaces it with "*arg_ts[0]*" and performs an $ASSIGN$ between "*ret_id*" and "*arg_ts*".

3.4 Prune

The $Prune$ instruction (Fig 4) represents the beginning of a conditional method body with an expression *exp* as the condition and the boolean *bol* indicating which branch is being analyzed.

$$\Theta_Q[\![\text{Prune } exp, bol]\!](q) \stackrel{\text{def}}{=} \begin{cases} \text{PRUNE\_TRUE}[\![exp]\!](q) & bol = \text{TRUE} \\ \text{PRUNE\_FALSE}[\![exp]\!](q) & \text{otherwise} \end{cases}$$

Fig. 4. Prune Transfer Function

If *bol* is $TRUE$ then the analysis executes the $FILTER$ instruction. After that the analysis executes the POP instruction and if the next instruction has 2 predecessors then the analysis also performs a $PUSH$. If *bol* is $FALSE$ then the analysis executes the POP and if the next instruction has 2 predecessors then the analysis also performs a $PUSH$.

4 Evaluation

In order to evaluate the effectiveness and accuracy of our implementation, we assessed it with the same input data usage benchmark used to evaluate Lyra.[4] The benchmark contains 10 programs written in Python; since Infer does not support Python, we manually converted it to Java. The benchmark suite, averaging 25 lines of code, is specifically designed to test the detection of unused input data, ranging from simple examples with only variables containing booleans as input and a single conditional set of instructions, to tests containing dictionaries as input.

Our implementation behaved similarly to Lyra on the 10 programs: both our analysis as well as Lyra's presented no false-positives or false-negatives. In terms of performance, Lyra takes on average $0.0802s \pm 0.0820s$ and our implementation takes $0,1886s \pm 0,1336s$ The performance penalty was expected, since Infer has to build the intermediate representation before starting the analysis. Nevertheless, integration of the analysis in an industrial-strength tool that can support multiple languages justifies this penalty.

Additionally we tested the analysis in two data science and data analysis open-source projects in order to test the tool's capabilities in real world applications.

The first program analyzed was *Neo4j Graph Data Science Library* [5] , a plugin for *Neo4j graph database* that consists of a library with graph algorithms.

The analysis took on average $15,67s \pm 0,23s$ to package the library with *gradle* and convert the instructions into intermediate language *SIL*, plus an additional $21,29s \pm 0,13s$ on average to analyze the intermediate language. In comparison packaging the library alone, without the intervention of Infer took on average $6,33s \pm 0,47s$. The analysis evaluated 40 Java files with an average of 120 lines of code. The analysis caught 371 false-positives and 0 true-positives. The analysis raised a substantial amount of false-positives due to external packages being imported and utilized in the code and by methods that altered objects but did not return variables.

The second program analyzed was *Ananas Desktop*[6], a open source data integration and analysis tool that allows non technical users to edit data processing jobs and visualize data. Unfortunately the analysis does not compile, but three files were able to be individually analyzed. The analysis took on average $1,94s \pm 1,38s$ to compile each Java file with *javac*, convert the instructions into intermediate language *SIL* and to analyze the intermediate language. In comparison compiling the files alone, without the intervention of Infer took on average $0,39s \pm 0,15s$. The analysis evaluated 40 Java files with an average of 83 lines of code. The analysis caught 25 false-positives and 0 true-positives. The causes of these false-positives are the same as the previous program: external packages being used and methods that altered objects without returning variables.

[4] Lyra benchmark: https://github.com/caterinaurban/Lyra/tree/master/src/lyra/tests
[5] Neo4j Graph Data Science Library: https://github.com/neo4j/graph-data-science
[6] Ananas Desktop: https://github.com/ananas-analytics/ananas-desktop

5 Related Work

To the best of our knowledge, Urban and Müller's work on Lyra [8] is the first that aims at detecting programming errors in data science code using static analysis. Engel [4] extends Lyra with usage analyses that can analyse map data structures. More recently, Urban et al. propose a parallel static analysis for certifying causal fairness of feed-forward neural networks used for classification of tabular data [7].

Also related are the works of Cheng and Rival [3] and Barowy et al. [1], but these focus on spreadsheet applications and on errors in the data, rather than the code that analyzes the data.

6 Conclusion

In conclusion we were able to adapt Lyra's approach to work in Infer. Our Implementation can effectively identify input data usage errors in the same benchmark used to assess Lyra.

Unfortunately our analysis only works for java, but we plan to add support for other languages supported by Infer in the future. Additionally the analysis does not run in every program, which is something we are focusing on in the future.

References

1. Barowy, D.W., Gochev, D., Berger, E.D.: Checkcell: Data debugging for spreadsheets. ACM SIGPLAN Notices **49**(10), 507–523 (2014)
2. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: NASA Formal Methods Symposium. Springer (2015)
3. Cheng, T., Rival, X.: Static analysis of spreadsheet applications for type-unsafe operations detection. In: European Symposium on Programming Languages and Systems. pp. 26–52. Springer (2015)
4. Engel, L.: Usage of Data Stored in Map Data Structures. Ph.D. thesis, Master's thesis, ETH Zurich, Zurich, Switzerland (2018)
5. Herndon, T., Ash, M., Pollin, R.: Does high public debt consistently stifle economic growth? a critique of reinhart and rogoff. Cambridge journal of economics **38**(2), 257–279 (2014)
6. Mencinger, J., Aristovnik, A., Verbic, M.: The impact of growing public debt on economic growth in the european union. Amfiteatru Economic Journal **16**(35), 403–414 (2014)
7. Urban, C., Christakis, M., Wüstholz, V., Zhang, F.: Perfectly parallel fairness certification of neural networks. arXiv preprint arXiv:1912.02499 (2019)
8. Urban, C., Müller, P.: An abstract interpretation framework for input data usage. In: European Symposium on Programming. pp. 683–710. Springer (2018)