A Taint Analyzer for COBOL Programs

Alberto Lovato
University of Verona
Verona, Italy
Email: alberto.lovato@univr.it

Roberto Giacobazzi
University of Verona
Verona, Italy
Email: roberto.giacobazzi@univr.it

Isabella Mastroeni
University of Verona
Verona, Italy
Email: isabella.mastroeni@univr.it

***Abstract*—The potential damage injection attacks or information leakage can inflict to an organization is huge. It is therefore important to recognize vulnerabilities in software that can make these attacks possible. We are implementing a static analysis that tracks propagation of tainted values through a COBOL-85 program. This analysis is part of an already developed static analyzer performing many syntactic checks and a semantic interval analysis. It can be used to find untrusted values ending in dangerous places, for example executed as database queries, or to verify that sensitive information coming from a database is not displayed to the user.**

Keywords–Taint analysis; Injection attacks; Information leakage; COBOL.

I. INTRODUCTION

COBOL is a programming language for business use. It was designed in 1959, and is still employed in many organizations. The existing codebase is huge, and experienced COBOL programmers are aiming for retirement. Many organizations have migration plans, but a substantial part of COBOL code is not going to be dismantled in the foreseeable future. Being used for security critical tasks, e.g., transactions between bank accounts, it is important to verify that COBOL programs have as few vulnerabilities as possible. COBOL-85 programs are structured into *divisions*. In particular, the *data division* contains variable declarations, and the *procedure division* contains executable code. It is imperative, structured code, with no object-orientation.

This paper describes a prototype of a static analyzer for tracking of values that we consider *tainted*—e.g., coming from the user (untrusted), or from a database (sensitive)—in COBOL-85 code. This is done by first translating the program into an internal, simpler language, only considering modification of strings, and then by defining a *transfer function* propagating tainted values. The transfer function is applied by an *interpreter* to each statement of the intermediate program, to update the set of tainted variables at that program point.

Injections are the top vulnerabilities found in web applications [1], and although COBOL is generally not used in front end development, it can still be used in the back end part of the application. It is therefore desirable to be able to find injection vulnerabilities in COBOL code.

Injection detection in other languages is well studied, for example for the Java language [2], [3], JavaScript [4], Android [5], scripting languages, e.g., Python [6], and even web frameworks [7]. However, to our knowledge, taint analysis in COBOL is not considered in the academic community.

The paper is structured as follows. Section II describes the ARCTIC analyzer, that contains the code for the taint analysis that is explained in the paper. In Section III, the analysis is described in detail. In Section IV, the analysis of a small COBOL program is executed. Section V concludes the paper.

II. THE ARCTIC ANALYZER

The analysis code is part of ARCTIC, a general static analyzer for COBOL-85 programs.

ARCTIC currently performs a lot of syntactic analyses, along with a semantic analysis for the computation of variable intervals. More precise numerical analyses are in development. Interval analysis also is executed on a simpler internal language, this time only considering modification to numerical variables.

ARCTIC is written in Java, has command-line and remote interfaces, and can be run by a *SonarQube* [8] plug-in. SonarQube is a platform used in many organizations to run analyzers and tools for code quality management of software projects. A *scanner* module is responsible for running analyses on code and sending the result to the *server* module. A user can then connect to the server with a browser to look at nicely formatted statistics and issues. The SonarQube plug-in of ARCTIC sends analysis *rules* selected by the SonarQube user and the paths of files of the project to the ARCTIC server, which analyzes them and sends issues back to the plug-in.

The interaction between ARCTIC and SonarQube is shown in Figure 1.

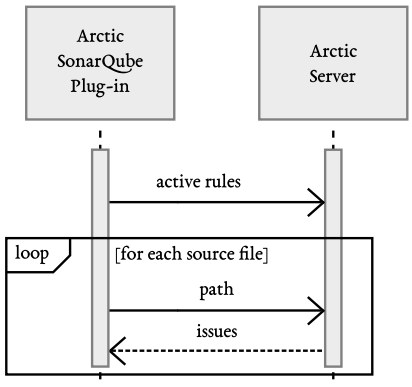

Figure 1. Interaction between the ARCTIC server and the *SonarQube* plug-in.

Issues are then available to be displayed in the SonarQube web interface. In Figure 2, there is an example result of taint

 ISBN: 978-1-61208-755-9

analysis performed by ARCTIC, as seen by a user connecting to the SonarQube server with a browser. Issues are shown below the line to which they belong, after the user clicks on issue markers on the left. In this example, the analysis output is a set of variables that are tainted before reaching every program point, and so are the issues.

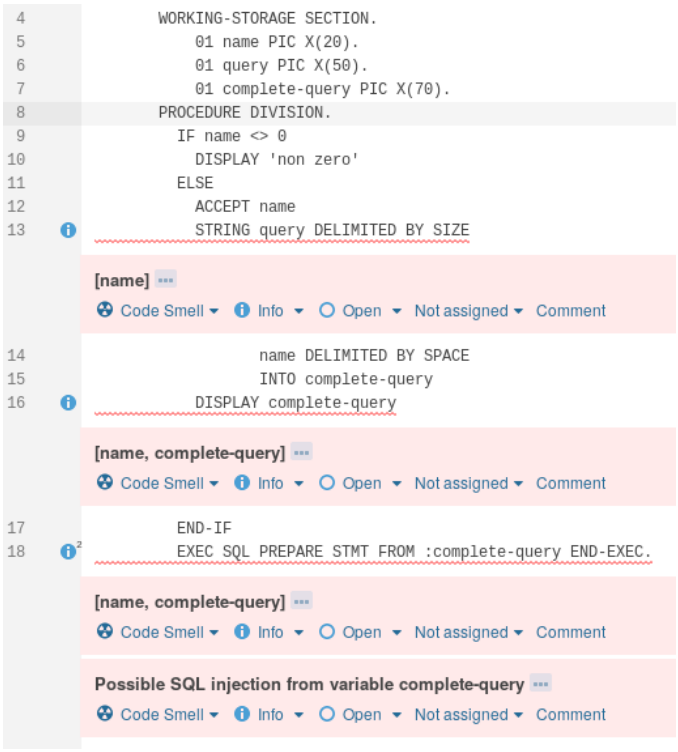

Figure 2. Taint analysis output displayed in the *SonarQube* web interface.

SonarQube rules represent some kinds of condition that users want to check. Rules may be activated in user defined *quality profiles*. Figure 3 is a screenshot of some rules defined in the ARCTIC SonarQube plug-in.

SonarQube itself contains a module for analysis of COBOL programs [9] in its Enterprise Edition, but at present it does not perform taint nor interval analysis.

ARCTIC uses a parser for COBOL-85 code that is a fork of *proleap-cobol-parser* [10], supporting ANSI 85, IBM OS/VS and MicroFocus dialects. The Abstract Syntax Tree (AST) produced by the parser contains representations of COBOL components, such as divisions, statements and declarations in a hierarchical way. Nodes of the AST can be accessed by using the *visitor* pattern, that allows client programmers to act on elements of a certain type by simply overriding a method in a class. Syntactic checks can be performed right after parsing, to detect situations like obsolete or insecure statements, bad coding practices, and type errors. Some other analyses verify the correctness of SQL code embedded into COBOL programs. This SQL code is extracted by the COBOL parser from `EXEC SQL` statements, and then parsed and analyzed with the aid of an external library, *JSqlParser* [11].

III. Taint Analysis

The analyzer works by interpreting the code, in order to compute a representation of the state where, at each program point, it is clear which variables are tainted. It considers a version of the program containing only relevant information, such as data about text variables and statements manipulating or using them. This simplifies the analysis a lot, as COBOL code is very verbose, and many statements are redundant for the analysis. The translation process is explained in Section III-A. The state in our case is simply the set of tainted variables at each program point. The interpreter executes each statement of the intermediate language by giving to it as input state the output state of the previous one, as shown in Figure 4.

The initial state is in general empty, as no variable is tainted at start. However, procedure divisions in COBOL may have *parameters*, since they can be called as subprograms by other programs. We do not analyze flows between programs yet, but the user can specify a flag, in order to consider parameters of procedure divisions as tainted. For each statement, the analysis tracks the current tainted variables, and if they flow into a sink, an issue is reported. An example is shown in Table I.

TABLE I. Example program analysis.

State before	Statement
$\emptyset$	**DISPLAY** x
$\emptyset$	**ACCEPT** x
$\{x\}$	**STRING** 'Input: ' x **DELIMITED BY SPACE INTO** y
$\{x, y\}$	...
$\{x, y\}$	$sink(y) \leftarrow$ report issue

Here, the second statement adds the variable x to the set, since its content is coming from the user. The following statement transfers taintedness to variable y. Lastly, the tainted value reaches a sink, and the analyzer reports an issue.

A. Translation

If we are interested in detecting injection of untrusted data, for example in a database query, we have to look for text variables, as numerical variables cannot be used to perform an injection attack.

ACCEPT. The means by which a user could directly insert a value into a COBOL-85 standard program is the **ACCEPT** statement, that reads input from the console, and is as such considered a source of untrusted data. It is translated into ***ACCEPT*** x, where x is the variable receiving the data, unless the statement accepts data from the system date; in that case it is translated into ***SKIP***, since the date is not an untrusted input.

STRING. The **STRING** statement concatenates several strings into one. It is translated into the intermediate statement ***STRING*** $x_1, \ldots, x_n$ ***INTO*** y.

MOVE. The **MOVE** statement moves the value of a variable into another variable. It is translated into ***MOVE*** x ***TO*** y.

Paragraphs. COBOL code is organized in paragraphs, labeled blocks of code. Procedure calls are implemented in COBOL by using the **PERFORM** statement, followed by the names of the blocks to execute, which form the body of the procedure. Considering for example a code subdivided in three paragraphs like this

```
PAR1.
  ACCEPT name
  DISPLAY name.
  ...
PAR2.
```

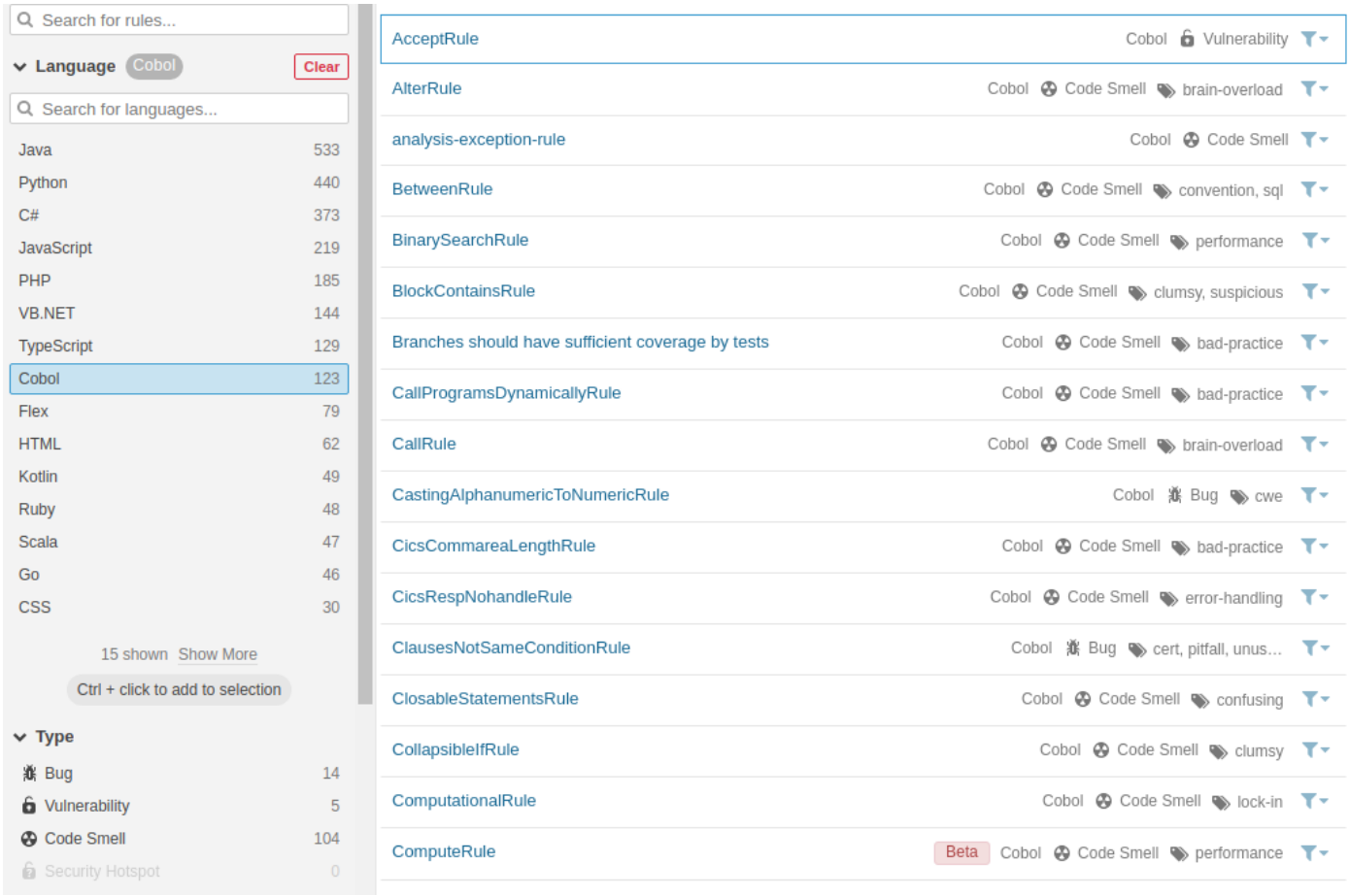

Figure 3. ARCTIC rule list in the SonarQube plug-in

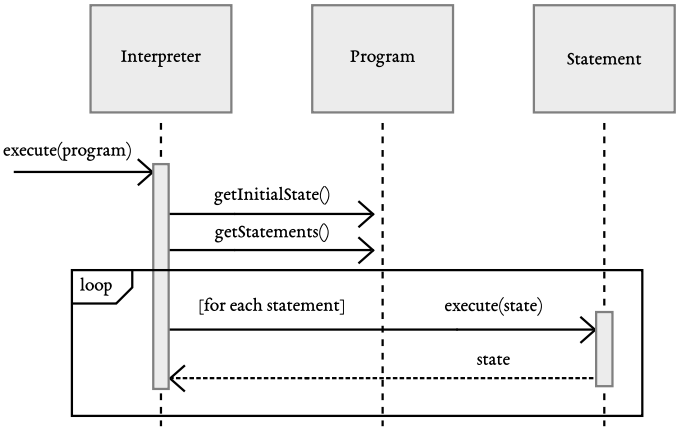

Figure 4. Execution of the program by the interpreter.

```
...
PAR3.
...
```

a single block would be called with **PERFORM** PAR1, whereas a sequence of blocks would be called with something like **PERFORM** PAR1 **THRU** PAR3. Procedure calling statements like those above are translated into *EXECUTEBLOCKS first [last]*. **PERFORM** can also describe loops, with the clause **UNTIL**, that repeats the body of the statement until a certain condition becomes true, or with the clause **TIMES**, that repeats the body of the statement a specified number of times. Taintedness does not change in loops, so these statements are translated like any other kind of **PERFORM**.

Injection. We are also interested in statements that may cause the unintended execution of code. For SQL injection, the statement EXEC SQL PREPARE STMT FROM :DYNSTMT END-EXEC executes a possibly dynamically created query stored in DYNSTMT. It is translated into ***EXECSQLPREPARE*** *DYNSTMT*. These statements, where flow of tainted information can cause unintended interaction with other parts of the system, are called *sinks*. We also denote the previous statement by $sink(\texttt{DYNSTMT})$.

Control flow statements. **IF** statements execute a branch or another according to the valuation of a condition. The corresponding intermediate statement is ***IF*** *condition* ***THEN*** B_1 ***ELSE*** B_2, where B_1 and B_2 are blocks of intermediate statements.

Other statements. Statements that do not deal with string manipulation are translated into the intermediate statement ***SKIP***, that is ignored by the analysis.

Intermediate statements store information about the orig-

inal COBOL statement, such as the line number, in order to map issues back to the original position in the source code. They also store the state before their execution, so that it is readily available to be displayed at the right program point.

B. Transfer Function

Let $\mathbf{T}$ be the set of all sets of tainted variables. We define a transfer function $f : \mathbf{T} \to \mathbf{T}$ that specifies which variables are tainted after the execution of each statement, given the input state $T \in \mathbf{T}$. The interpreter implements this transfer function in the `execute` method of the class corresponding to each statement type, to produce a set of tainted variables for every program point.

ACCEPT x**.** This statement gets an input string directly from the user. This is a potential source of untrusted data, and so we mark the receiving variable as tainted.

$$f_{ACCEPT}(T) = T \cup \{x\} \tag{1}$$

STRING $x_1, \ldots, x_n$ **INTO** y**.** String values are concatenated with the **`STRING`** statement, that as such transfers the taintedness properties from source variables to the receiving variable. If at least one of the variables containing the strings that are being concatenated is tainted, then we mark the receiving variable as tainted.

$$f_{STRING}(T) = \begin{cases} T \cup \{y\} & \text{if } \exists x \in \{x_1, \ldots, x_n\}.x \in T \\ T & \text{otherwise} \end{cases} \tag{2}$$

MOVE x **TO** y**.** **`MOVE`** makes the receiving variable tainted if and only if the source variable is tainted.

$$f_{MOVE}(T) = \begin{cases} T \cup \{y\} & \text{if } x \in T \\ T & \text{otherwise} \end{cases} \tag{3}$$

Paragraphs. Each paragraph is a list of statements, e.g., $P1 = S_1 \ldots S_n$; the transfer function of a paragraph is the composition of the transfer functions of the statements.

$$f_{P1}(T) = f_{S_n}(\ldots(f_{S_1}(T))\ldots) \tag{4}$$

EXECUTEBLOCKS $first$ **[**$last$**].** For every COBOL program, we build a list of blocks corresponding to its paragraphs, for example `P1, P2, P3`. When we then execute a block with the intermediate statement EXECUTEBLOCKS P1, its transfer function is that of the executed block.

$$f_{EB}(T) = f_{P1}(T) \tag{5}$$

The transfer function of the execution of several blocks, e.g., EXECUTEBLOCKS P1 P3, is the composition of the functions of the executed blocks.

$$f_{EB}(T) = f_{P3}(f_{P2}(f_{P1}(T))) \tag{6}$$

IF *condition* **THEN** B_1 **ELSE** B_2**.** We conservatively keep taintedness information of both branches of a conditional statement, and so the transfer function is the union of the two functions.

$$f_{IF}(T) = f_{B_1}(T) \cup f_{B_2}(T) \tag{7}$$

SKIP. This statement does nothing regarding the modification of taintedness of variables, and so its transfer function is the identity function.

$$f_{SKIP}(T) = T \tag{8}$$

Sinks do not modify taintedness, and thus they have an identity transfer function. Table II sums up translation and transfer function result for every intermediate statement.

C. Implementation

Figure 5 shows the transformation of a COBOL program. Variables containing text are extracted in a list, whereas COBOL statements are translated into intermediate language. A list of the paragraphs found in the program is kept in memory to allow the interpreter to execute EXECUTEBLOCKS statements.

Figure 6 outlines the execution of the intermediate program. The interpreter executes the transfer function of every statement, and the produced state is retained in the executed program, associated to the next statement.

IV. Example

In this section, we reconsider the example of Figure 2, and show how it is transformed and executed.

```
1   IDENTIFICATION DIVISION.
2   PROGRAM-ID. example.
3   DATA DIVISION.
4   WORKING-STORAGE SECTION.
5       01 name PIC X(20).
6       01 query PIC X(50).
7       01 complete-query PIC
            X(70).
8   PROCEDURE DIVISION.
9    PAR1.
10    IF name <> 0
11      DISPLAY 'non zero'
12    ELSE
13      ACCEPT name
14      STRING query DELIMITED BY
            SIZE
15             name DELIMITED BY
                  SPACE
16             INTO complete-query
17      DISPLAY complete-query
18    END-IF
19   PAR2.
20    EXEC SQL PREPARE STMT FROM
          :complete-query END-EXEC.
```

Three alphanumerical variables are declared at lines 5-7, so the variable list `name, query, complete-query` is produced by the `VariableExtractor`. The paragraph list `PAR1, PAR2` is saved in the intermediate program. The `IF` statement at lines 10-18 is translated as

```
IF condition
  SKIP
ELSE
  ACCEPT name
  STRING query, name INTO complete-query
  SKIP
```

TABLE II. TRANSFER FUNCTION f.

COBOL	Intermediate	$f(T)$
ACCEPT <identifier>	ACCEPT x	$T \cup \{x\}$
STRING ... **INTO** ...	STRING $x_1, \ldots, x_n$ INTO y	$\begin{cases} T \cup \{y\} & \text{if } \exists x \in \{x_1, \ldots, x_n\}.x \in T \\ T & \text{otherwise} \end{cases}$
MOVE ... **TO** ...	MOVE x TO y	$\begin{cases} T \cup \{y\} & \text{if } x \in T \\ T & \text{otherwise} \end{cases}$
P1. <statements>	P1 = $S_1 \ldots S_n$	$f_{S_n}(\ldots(f_{S_1}(T))\ldots)$
PERFORM P1	EXECUTEBLOCKS P1	$f_{P1}(T)$
PERFORM P1 **THRU** P3	EXECUTEBLOCKS P1 P3	$f_{P3}(f_{P2}(f_{P1}(T)))$
control flow	IF condition THEN B_1 ELSE B_2	$f_{B_1}(T) \cup f_{B_2}(T)$
sinks	e.g., EXECSQLPREPARE source	T
other statements	SKIP	T

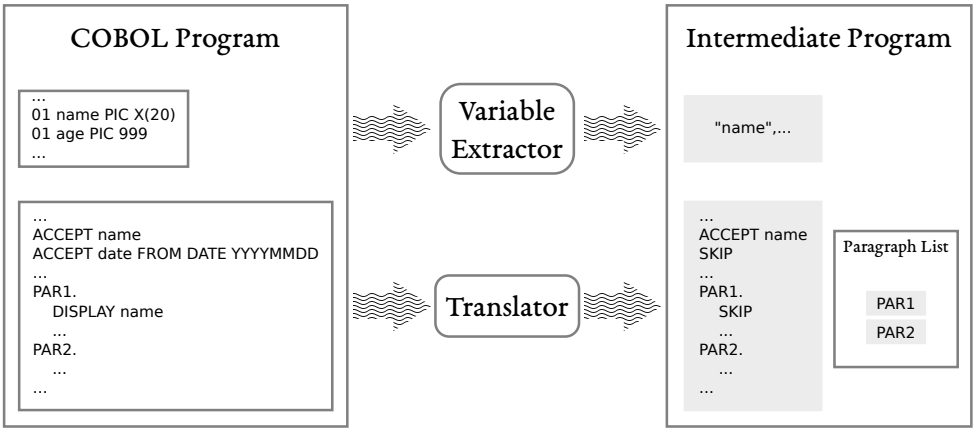

Figure 5. Transformation of a COBOL program into the intermediate representation.

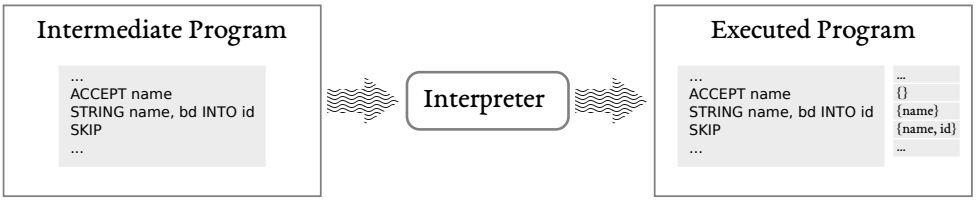

Figure 6. Execution of the intermediate program by the interpreter.

whereas the statement at line 20 is translated as

```
EXECSQLPREPARE complete-query
```

Then, each statement is executed by applying the logic defined in Section III-B. For example, the transfer function of the IF statement is defined as

$$\begin{aligned} f_{IF}(T) &= f_{SKIP}(T) \cup f_{SKIP}(f_{STRING}(f_{ACCEPT}(T))) \\ &\overset{(8)}{=} T \cup f_{STRING}(f_{ACCEPT}(T)) \\ &\overset{(1)}{=} T \cup f_{STRING}(T \cup \{\texttt{name}\}) \\ &\overset{(2)}{=} T \cup T \cup \{\texttt{name}, \texttt{complete-query}\}) \\ &= T \cup \{\texttt{name}, \texttt{complete-query}\} \end{aligned}$$

Before executing the `IF` statement, the set T is empty, since none of the declared variables are tainted at that point. The final set is thus $\{\texttt{name}, \texttt{complete-query}\}$.

V. CONCLUSION

The prototype we developed is able to track propagation of tainted values in COBOL-85 programs. We are not aware of any other taint analyzer for COBOL code. At the moment, the analyzer only checks for SQL-injection, but, as future work, we could also consider other kinds of injection, or the other way round, the leaking of sensitive values. Information from a database, e.g., a credit card number, may be displayed to the user if a variable containing it is used as argument of the COBOL statement **DISPLAY**. This kind of analysis would require considering **DISPLAY** statements and analogous ones (e.g., GUI output statements) as sinks, while sources of tainted

information would be statements reading from the database into host variables. Also, other versions of COBOL may allow users to inject values via other means, e.g., a graphical user interface, and not only via the **ACCEPT** statement, so we may extend the analysis to include this possibility.

REFERENCES

[1] "Owasp Top 10 Project," 2019, URL: https://www.owasp.org/index.php/Category:OWASP˙Top˙Ten˙Project [retrieved: 2019-10-27].

[2] F. Spoto, E. Burato, M. D. Ernst, P. Ferrara, A. Lovato, D. Macedonio, and C. Spiridon, "Static Identification of Injection Attacks in Java," ACM Trans. Program. Lang. Syst., vol. 41, no. 3, 2019, pp. 18:1–18:58.

[3] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: effective taint analysis of web applications," in Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009, 2009, pp. 87–97.

[4] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the World Wide Web from Vulnerable JavaScript," in Proceedings of the 2011 International Symposium on Software Testing and Analysis, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 177–187.

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269.

[6] S. Liang and M. Might, "Hash-flow Taint Analysis of Higher-order Programs," in Proceedings of the 7th Workshop on Programming Languages and Analysis for Security, ser. PLAS '12. New York, NY, USA: ACM, 2012, pp. 8:1–8:12.

[7] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, "F4F: Taint Analysis of Framework-based Web Applications," in Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 1053–1068.

[8] "SonarQube platform," 2019, URL: https://www.sonarqube.org [retrieved: 2019-10-27].

[9] "SonarCOBOL," 2019, URL: https://www.sonarsource.com/products/codeanalyzers/sonarcobol.html [retrieved: 2019-10-27].

[10] "Proleap Cobol Parser," 2019, URL: https://github.com/uwol/proleap-cobol-parser [retrieved: 2019-10-27].

[11] "Java SQL Parser," 2019, URL: https://github.com/JSQLParser/JSqlParser [retrieved: 2019-10-27].

 ISBN: 978-1-61208-755-9