Intra-procedural Static Taint Analysis of JavaScript Programs

Christoph Zorn 2956165
st115403@stud.uni-stuttgart.de
University of Stuttgart, Germany

ABSTRACT

Software development is known to be an error-prone technique since it involves the need for human software engineers. Methodologies such as unit testing, continuous integration and debugging have helped to increase the safety and security of software systems over the last decades. Nevertheless, there still exist parts of a software system that can not be tested in advance and require extensive work. Configurations or inputs made by users that produce unintended results are very hard to detect and reproduce. As the information of the user input can spread through a computer program it is necessary to isolate it from locations that are visible to other users or can be accessed from other services. This work gives an overview of static taint analysis and proposes a practical implementation that helps to show leaking parts of JavaScript programs.

1 INTRODUCTION

Taint analysis (TA) is an established methodology that can help to warn about the leaks of private information. TA can be used to identify variables of a program that contain and may leak secret information or use unsafe input that was given by a user.

This section introduces the concept of TA in Section 1.1 and related methodologies in the following sections. Section 2 focuses on our implementation approach of the static taint analysis into the google closure compiler (GoCC). Section 3 emphasizes on the use of our implementation while Section 4 summarizes the discussion and our findings.

1.1 Taint Analysis

Parts of computer programs, especially in bigger and more complex systems require modulaization and encapsulation. This method usually helps to store data of the same type in the same place and helps to reduce the room for errors. The analysis of information control flow takes a similar approach with the introduction of security classes [1]. For example, a module that does not use sensitive information has a lower security class than a module that works with sensitive data. Just like in big systems this approach can be applied to smaller programs too.

In the following, we call a flow of secret information that reaches a sink a leak. Listing 1 uses a JavaScript program to demonstrate a leak of secret information inside a computer program. A variable that is assigned a call the function *retSource* stores secret information while a call to *sink* with a parameter that contains a secret is viewed as a critical leak. At first sight, it may not obvious to the reader to say whether the program is leaking secret information to the outside or not. Scoped constructs like conditions or loops make it especially hard to remember and follow the flow of information. Also, reassignments of variables with secrets can disguise which variable is currently holding a secret.

```
function infoLeak() {
    // retSource returns secret information.
    var foo = retSource();
    var bar = foo + " - secret ";
    var baz = bar.length - 5;

    if(baz < 5) {
        sink(42);
    } else {
        sink(bar);
    }

    // Any call to sink leaks
    // information to the outside.
    sink(baz);
}
```

Listing 1: Example of a program that leaks information from a source to a sink.

The approach to detect which parts of the program may be leaked is called information flow analysis. This method is split into two separate types of flows. On the one hand, there is are explicit flows that can easily be detected by following the mentioning of a variable that contains a secret. On the other hand, there are implicit flows that require the dynamic evaluation of expressions and the following nested conditional expressions. In this work, we only care about explicit flows. However, we distinguish between variables that may reach the sink or must reach the sink if we are not sure whether a condition holds or not.

Studies of real-world JavaScript applications have shown that most programs lack a security mechanism to prevent implicit flows [5]. On the one hand, this is due to expensive costs and on the other hand due to complex flows. The study concludes that taint tracking suffices in most cases.

2 APPROACH

Our approach for a TA for JavaScript code is based on the google closure compiler [2]. GoCC was launched in 2009 as an open-source project under the Apache 2.0 License and it is based on existing Google's JavaScript frameworks and projects [4]. Since its launch, the GitHub repository has over 14000 commits and more than 450 contributors. GoCC already has implemented over 200 code checks and sanitizers for JavaScript code. The main goal of GoCC is to convert existing code into a minimized and high-performance JavaScript code [4]. For example, the GoCC removes unused and dead code to reduce the size of the resulting output file.

Since we do not evaluate expressions during run-time of a program, our approach will be a static taint analysis (STA). Our goal for the STA is to find all variables inside a function scope that may or must leak secret information. We then write the affected functions and variables to a JSON file with the format shown in Listing 3. In this format, we only consider leaking variables that were initially sources. For example, the variable A is a source and assigned to variable B. If B reaches a sink we only mark A as tainted.

2.1 Implement a Compiler Pass

One way to add a check for JavaScript code anomalies within a piece of code is to implement a compiler pass (CP) to the GoCC. We decided to implement our approach as a CP since it interacts well with the rest of the already existing GoCC checks. Since we extend the existing GoCC code base the first step is to create a custom class *StaticTaintAnalysisPass* in the package *jscomp* and inherit from the interface *Compiler Pass*. This class needs to implement the abstract method *process* which later allows the compiler to run the check if it is specified by the user. Later, the CP will be added as a compiler option to the command line runner which is explained in Section 2.3.

The actual analysis will be done in a separate class that inherits from the class *DataFlowAnalysis*. In the custom compiler pass, we create a new instance of this type every time we enter a new scope. This way we can inspect functions separately.

2.2 Implementing the Analysis

The complete source code to compile the GoCC with the STA can be found on our GitHub page [6]. Since we want to run a data flow analysis we create another class *StaticTaintAnalysis* that extends the class *DataFlowAnalysis*. This class will do the whole analysis and can provide the results of it to the CP. In the class *StaticTaintAnalysis* we decided to go for a recursive approach. Using recursion in an environment where syntax trees are used seems logical. In the CP we create an instance of the *StaticTaintAnalysis* class and call the function *doSTA* which will trigger our analysis. This function is given in Appendix A. The *doSTA* method takes one argument of the type *Node*. When the method is initially called, it is given the root node of a scope which represents the beginning of a function. The first child of this node is the function name, the second holds the parameter list and the third one holds the body of the function. As we analyze variables within a function we can only start the analysis when the syntax tree of the input file consists of a function with a non-empty body. This means that the third child node must have at least one child node.

In the following, we define a context as the type of the parent token(s) of the current node. For example, a variable declaration consists of a name and an optional assignment. If an assignment exists, elements of its syntax tree are in the context of a variable. The code depicted in Listing 2 will produce the syntax tree shown in Figure 1. *bar* and *baz* are in the context of *foo*, while *baz* is also in the context of *bar*. We use this notion of a context in our implementation to check for sources, sinks, and statements that are inside a conditional block.

```
var foo = bar(baz);
```

Listing 2: Variable declaration with assign statement of a function call.

As mentioned before we first extract the name and the line number of the function which scope we currently inspect. After

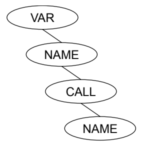

Figure 1: Syntax tree of the Listing 2. The node CALL is within the context of VAR.

this, we look for tokens that are of particular interest to us. Those are variables, calls to functions or objects, and conditional statements. We separated the analysis of each of those into separate functions that are called from the *doSTA* function. Appendix A show how the nodes of specific tokens get distributed to the respective functions.

Some of the tokens do only make sense to analyze when they are in the context of a condition or variable. For example, an object that is not assigned to a variable can not leak information to a sink and therefore is only important to us when it is called as an assignment of a variable, or as an inline argument to a function call. The order of evaluation of the tokens can be represented as a flow chart which is depicted in Figure 2. The figure does not show all token we inspect during the analysis but those we omitted are assembled with existing tokens. For example, a function call from a prototype consists of a label, call, and another label. Since the full configuration of all states would reduce the readability of the flow diagram we dispensed those.

We marked the connections that are from particular interest to us with red. An assignment that calls a function with the name *retSource* is a source. A function with the name *sink* that takes exactly one parameter is a sink.

Conditional block statements such as loops and if conditions make it more difficult to decide if a nested statement can be reached. For this, we tracked the current scopes with a reachability property. A condition that contains only constants is seen as always reachable, while a condition that contains a variable name is maybe reachable. Upon entering the scope we mark all already defined with this property and reset it to the outer scopes reachability when leaving it. As conditional statements can contain rather big subtrees of logical operators, a condition is also evaluated recursively. Important to note here is that an addition of two different reachabilities results in the lower reachability class. For example, the condition *true* is always reachable while *len > 0* is only maybe reachable. Therefore, the resulting reachability class is maybe.

2.3 Add a Compiler Option

By default, the GoCC compiler makes a distinction between checks and optimizations. Checks will only notify a user if there exists a problem in their code while optimizations will alter the code and reduce the size of the input. Since we can not remove a leak, even if it exists, we add the STA as a compiler check. For our implementation, we followed the instructions on the official GoCC GitHub repository [3]. First we added a flag *staticTaintAnalysis* and the associated getter and setter functions to the class *CompilerOptions* that is located in the package *jscomp*. The next step is to enable this flag within a compiler options group. This is done in the the

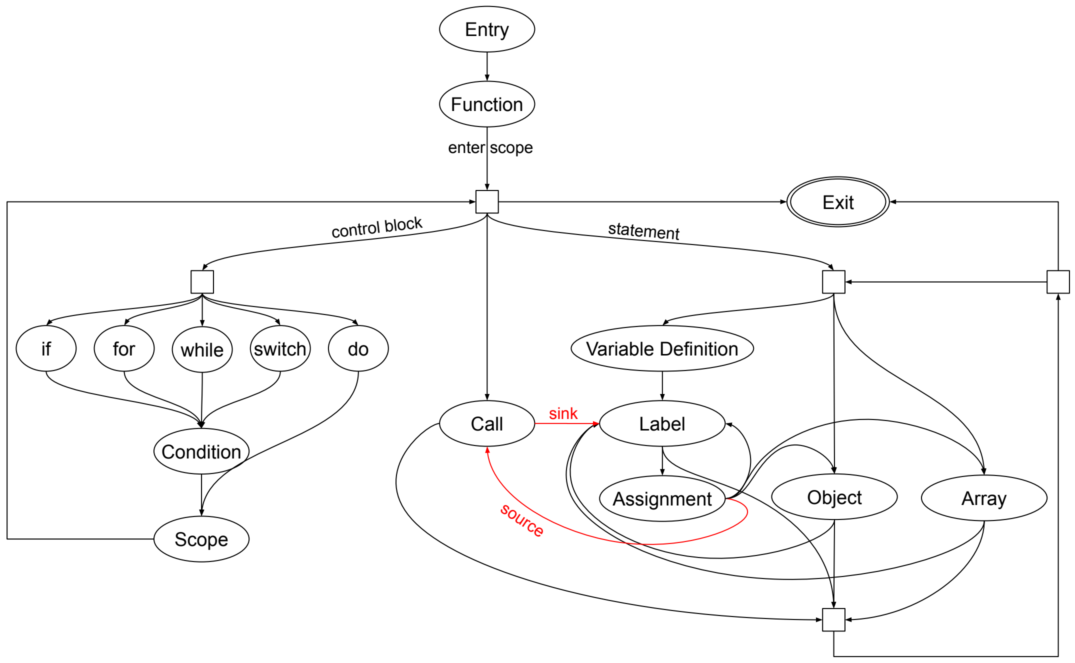

Figure 2: The order of evaluation of tokens and constructs.

function *applySafeCompilationOptions* as well in the function *applyFullCompilationOptions* from the class *CompilationLevel*. The function *applySafeCompilationOptions* only enables safe compiler options meaning those which do not alter the code. Lastly we have to add our CP class to the compiler checks which are in the class *DefaultPassConfig* in the function *addChecks*. If the GoCC is compiled now and executed our implementation will check for any leaks in a given program.

```
{
  "infoLeak@1": {
    "sources_that_must_reach_sinks": ["foo@2"],
    "sources_that_may_reach_sinks": ["foo@2"]
  }
}
```

Listing 3: The output our implementation will produce when the JavaScript example from Listing 1 is given.

3 STATIC TAINT ANALYSIS IN PRACTICE

In the following, we shortly show how our implementation works.

Once the GoCC is compiled into an executable jar file we can do a static taint analysis on a JavaScript file. The command of how to execute the analysis is given in Listing 4.

```
java -jar closure-compiler.jar --js=script.js
```

Listing 4: Execution command of the GoCC that takes a JavaScript file.

This command will take the given script and looks for any information leaks inside every function within this script and performs an STA. When this is completed, a JSON result is written in the same directory where the input script is located. The output file will have the same name as the input file, with a postfix *_out* after the name and before the file format.

4 CONCLUSION

In the following, we conclude our findings from the previous sections and discuss them. In the end, we give a short outlook to the future.

4.1 Discussion

For our approach of a static taint analysis, we extended the GoCC with a compiler pass. The implementation can detect information leaks in a JavaScript program. It can recognize when a variable containing secret information is leaking to an outside function. This includes the analysis of complex conditional and nested expressions.

Although our approach can warn about information leaks for most JavaScript programs, it is still limited in its usage. For example, our implementation does not include checks for the available syntax of JavaScript. The checks are also limited in the dynamic evaluation of conditional statements. Also, our approach works currently only for static names of the source and sink functions. Besides these shortcomings, we are sure that our work can help developers to detect information leaks.

4.2 Future work

The presented work can be used in practice for JavaScript programs. However, our approach does only perform a static taint analysis with a static name for sources and sinks. Future enhancements could improve our work with an additional compiler option to configure those. The STA is also not able to discover information leaks in programs with the more complex syntax but rather on a subset that is commonly used. Additional checks for nested functions and prototype declarations could help to make our approach more robust.

REFERENCES

[1] Dorothy E Denning and Peter J Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (1977), 504–513.

[2] Google. 2009. Google Closure Compiler. https://github.com/google/closure-compiler. [Online; accessed 13-Januar-2020].

[3] Google. 2009. Google Closure Compiler - Compiler Pass. https://github.com/google/closure-compiler/wiki/Writing-Compiler-Pass. [Online; accessed 18-Januar-2020].

[4] Google. 2020. GoogleBlog. http://googlecode.blogspot.com/2009/11/introducing-closure-tools.html. [Online; accessed 13-Januar-2020].

[5] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. 2019. An Empirical Study of Information Flows in Real-World JavaScript. In *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*. 45–59.

[6] Christoph Zorn. 2020. Static taint analysis - PA project. https://github.com/styinx/PA. [Online; accessed 30-Januar-2020].

A STA ENTRY FUNCTION

```java
public void doSTA(Node n) {
  if (n == null) {
    return;
  }

  switch (n.getToken()) {
    case FUNCTION:
      // First child is the function name.
      result.name = n.getFirstChild().getString()
            + "@" + n.getLineno();
      doSTA(n.getLastChild());
      break;

    case CALL:
      doSTACall(n);
      break;

    case VAR:
    case CONST:
    case LET:
      doSTAVar(n);
      break;

    case NAME:
      doSTAName(n);
      break;

    case ASSIGN:
      doSTAAssign(n);
      break;

    case IF:
    case FOR:
    case FOR_IN:
    case WHILE:
      doSTAConditional(n);
      break;

    case SWITCH:
      doSTASwitch(n);
      break;

    default:
      doSTAChildren(n);
  }
}
```

Listing 5: Entry function for all new scopes that are analyzed by the *StaticTaintAnalysis* class.