A Robust Taint Analyzer for JavaScript in OCaml

Ziqi Yang
ziqi.yang@uzh.ch

1 Project Description

The project involves the development of a static analysis tool in OCaml designed to detect potential vulnerabilities in a real-world JavaScript codebase. The analyzer will operate on a pre-processed JavaScript file which is compiled from TypeScript and bundled to resolve modules. It will parse a full JSON AST generated by an external tool, but only transform and analyze a small, well-defined minimal syntactic subset relevant to taint tracking. This approach leverages OCaml's strengths in symbolic processing while pragmatically avoiding the complexity of parsing the full JavaScript language.

2 Motivation

I select a frontend project codebase which I was involved in developing from the SoPra course, since I thoroughly understand its implementation details and realize that something vulnerable may exist. Therefore, JavaScript is the target language of the analyzer, and more JavaScript code files from other sources can be used to test its analytical capabilities.
The project is well-suited for OCaml, a language with deep roots in program analysis and compiler design. The problem of static analysis is fundamentally one of symbolic manipulation and proving properties about recursive data structures.

- **Domain Specialization:** OCaml is the language of choice for many of the world's most advanced program analysis tools, including Facebook Infer (bug finding), Coq (proof assistant), and F* (secure language). This project directly aligns with the language's primary problem domain.
- **Data Structure Fidelity:** The essence of program analysis is data structure manipulation. OCaml's algebraic data types (ADTs) provide the most precise and natural way to model the Abstract Syntax Tree (AST) of a program.
- **Correctness & Safety:** OCaml's strong, static type system and its sophisticated `option` and `Result` types eliminate entire classes of bugs like null pointer exceptions in the analyzer's own logic, leading to a more reliable and robust tool.

3 Key Language Features

My design will rely on the features that make OCaml ideal for the project:

- **Algebraic Data Types (ADTs) and Pattern Matching:** AST is a recursive, variant data structure. An AST node can be an if statement, a variable declaration or a call expression. Therefore, I will define my minimal subset using OCaml's `type` keyword like the following structure:

```
type expression =
  | Literal of string
  | CallExpr of ...
  | MemberExpr of ...
  | Identifier of string
  | OtherExpr
```

The core analysis logic will then be a simple recursive function which matches on this type. This is safe enough, and trivial to extend.

- **Module System:** A clean architecture is essential for static analysis. I will use OCaml modules to separate concerns described in the Basic System Architecture section. The encapsulation makes the system maintainable.
- **Immutability:** The analysis should not mutate the AST. OCaml's functional-first approach encourages writing pure functions that take the AST and a taint state as input and return a new taint state as output, which makes the logic easy to reason about, test, and debug without side effects.

4 Basic System Architecture

The system is initially designed as the following pipeline with four stages, with clear boundaries between external tools and the OCaml program.

1. **Stage 1 - Pre-processing:** The target project in TypeScript is compiled to JavaScript by `tsc` and then bundled using `esbuild`, which resolves all local `import` statements and transform the problem between files into a simpler one in a single file.
2. **Stage 2 - Parsing:** A mature JavaScript parser reads the bundled file and produces a complete, concrete JSON AST. The parsing process or commands should be integrated into the OCaml code if necessary.
3. **Stage 3 - Transformation:** The OCaml program ingests the JSON and recursively walks the JSON tree. It will transform only the nodes corresponding to my minimal subset ADT and maps all others to a placeholder which will not interference the analysis.
4. **Stage 4: Analysis:** The analysis engine written in OCaml traverses the lightweight ADT, tracks variables tainted by known sources and flags any flow into a known sink.

5 Scope

Must-Haves

- Ability to ingest the JSON AST correctly
- A defined OCaml ADT for a minimal language subset
- Ability to find at least one vulnerability which can be audited manually and reproduced afterwards in the target project
- Must not crash on the full, complex bundled JS file

Nice-to-Haves

- Ability to track taint flow through simple function calls
- ADT Expansion for more complex flow like class members `this.x = y`
- Ability to report with a clear format

Out-of-Scope

- A JavaScript parser in OCaml
- Ability to analyze third-party code from `node_modules`
- Ability to find all potential vulnerabilities

6 Potential Challenges and Mitigation

- The full JSON AST from external tools may be overwhelmingly complex. I will not try to model the entire thing but only cherry-pick the keys and values I need and ignore everything else. As a result, the OCaml ADT will remain simple.
- Taint flow may be more complex than expected. Therefore, I will start from the simplest possible analysis which keep insensitivity for flows where a variable will be always tainted once tainted. The result can be somewhat imprecise but achievable.
- The `esbuild` bundled output may be mangled and hard to analyze, which means I have to spend time inspecting the `esbuild` output before writing code. The analyzer will be tailored to the actual code patterns produced by the bundler. I will try to adjust the `esbuild` configuration options to produce a more analyzable but still bundled output.