Methodology

Hawk is a solidity static analyser which identifies vulnerabilities and gas optimizations. Hawk has been coded in rust and is accessible by command line interface. Hawk can be passed the following command line arguments.

Below is the hawk --help output

Usage: hawk [OPTIONS]

Options:

  -p, --path <PATH>  Path to the directory containing the files hawk will analyze. The default directory is `./contracts`

  -t, --toml <TOML>  Path to the toml file containing the hawk configuration when not using the default settings.

  -h, --help         Print help information

The command line arguments are straightforward, the toml file path can be used to check for specific vulnerabilities in particular. The --path refers to the directory of solidity files that need to be analyzed.

The code below is the hawk.toml example showing the way hawk can be configured for specific vulnerabilities only.

*# Directory that contain contracts to be analyzed*

path = './contracts'

*# Optimizations that each contract will be analyzed for*

optimizations = ["address\_balance", "address\_zero", "assign\_update\_array\_value", "cache\_array\_length", "constant\_variables",

"bool\_equals\_bool", "immutable\_variables", "increment\_decrement", "memory\_to\_calldata", "multiple\_require", "pack\_storage\_variables",

"payable\_function", "safe\_math\_pre\_080", "safe\_math\_post\_080", "shift\_math", "solidity\_keccak256", "solidity\_math",

"sstore", "string\_errors"]

*# Vulnerabilities that each contract will be analyzed for*

vulnerabilities = ["unsafe\_erc20\_operation"]

*# QA that each contract will be analyzed for*

qa = []

All of these options are handled by the opt.rs file which handles all of the Command Line Argument Parsing. Helps users get a polished experience. The CLI looks clean with coloured output. Shell completion. Help generation. suggested fixes for users on errors.

Below is the opt.rs file.

use std::{fs, process};

use crate::analyzer::{optimizations, vulnerabilities};

use crate::analyzer::{

    optimizations::{str\_to\_optimization, Optimization},

    qa::{self, str\_to\_qa, QualityAssurance},

    vulnerabilities::{str\_to\_vulnerability, Vulnerability},

};

use clap::Parser;

#[derive(Parser, Debug)]

#[clap(

    name = "hawk",

    about = "A Solidity static analyzer to identify contract vulnerabilities and gas efficiencies."

)]

pub **struct** Args {

    #[clap(

        short,

        long,

        help = "Path to the directory containing the files hawk will analyze. The default directory is `./contracts`"

    )]

    pub path: Option<String>,

    #[clap(

        short,

        long,

        help = "Path to the toml file containing the hawk configuration when not using the default settings."

    )]

    pub toml: Option<String>,

}

pub **struct** Opts {

    pub path: String,

    pub optimizations: Vec<Optimization>,

    pub vulnerabilities: Vec<Vulnerability>,

    pub qa: Vec<QualityAssurance>,

}

#[derive(serde::Deserialize, Debug)]

pub **struct** hawkToml {

    pub path: String,

    pub optimizations: Vec<String>,

    pub vulnerabilities: Vec<String>,

    pub qa: Vec<String>,

}

impl Opts {

    pub fn new() -> Opts {

**let** args = Args::parse();

**let** (optimizations, vulnerabilities, qa) = if args.toml.is\_some() {

**let** toml\_path = args.toml.unwrap();

**let** toml\_str =

                fs::read\_to\_string(toml\_path).expect("Could not read toml file to string");

**let** hawk\_toml: hawkToml =

                toml::from\_str(&toml\_str).expect("Could not convert toml contents to hawkToml");

            (

                hawk\_toml

                    .optimizations

                    .iter()

                    .map(|f| str\_to\_optimization(f))

                    .collect::<Vec<Optimization>>(),

                hawk\_toml

                    .vulnerabilities

                    .iter()

                    .map(|f| str\_to\_vulnerability(f))

                    .collect::<Vec<Vulnerability>>(),

                hawk\_toml

                    .vulnerabilities

                    .iter()

                    .map(|f| str\_to\_qa(f))

                    .collect::<Vec<QualityAssurance>>(),

            )

        } else {

            (

                optimizations::get\_all\_optimizations(),

                vulnerabilities::get\_all\_vulnerabilities(),

                qa::get\_all\_qa(),

            )

        };

**let** path = if args.path.is\_some() {

            args.path.unwrap()

        } else {

            match fs::read\_dir("./contracts") {

                Ok(\_) => {}

                Err(\_) => {

                    yellow!(

                        "Error when reading the target contracts directory.

If the `--path` flag is not passed, hawk will look for `./contracts` by default.

To fix this, either add a `./contracts` directory or provide `--path <path\_to\_contracts\_dir>\n"

                    );

                    process::exit(1)

                }

            }

            String::from("./contracts")

        };

        Opts {

            path,

            optimizations,

            vulnerabilities,

            qa,

        }

    }

}

Hawk consists of the following modules.

1. Parser

2. AST Generator

3. Static Analyser

4. Utilities

5. Vulnerability Detectors

6. Reporters

7. Printers

1. Parser and 2. AST Generator

The parser and AST Generator work simultaneously to build the Abstract Syntax Tree. All of the inputted solidity smart contracts are parsed into their Abstract Syntax Trees. We can access and see the output of this module so we can get a better understanding of what is happening underneath.

Below is a basic solidity smart contract and it’s Abstract Syntax Tree.

pragma solidity ^0.8.16;

contract SimpleStorage {

    uint x;

    function set(uint newValue) {

        x = newValue;

    }

    function get() returns (uint) {

        return x;

    }

}

"SourceUnit(

    [

        PragmaDirective(

            File(

                0,

                10,

                33,

            ),

            Identifier {

                loc: File(

                    0,

                    17,

                    25,

                ),

                name: \"solidity\",

            },

            StringLiteral {

                loc: File(

                    0,

                    26,

                    33,

                ),

                unicode: false,

                string: \"^0.8.16\",

            },

        ),

        ContractDefinition(

            ContractDefinition {

This is the start of the Abstract Syntax Tree in solidity. This resembles how javascript’s AST in JSON would look.

3. Static Analyzer

This is the core of the project. It takes the input as the abstract syntax tree that is generated. This module then starts analyzing from the source unit root node. We need to have complete understanding over the solidity AST. This module walks over the AST so that it is coherent for the code. We do this by creating our own enumerators, data types and functionality’s impl structs. This is the essential data that is used in the detectors.