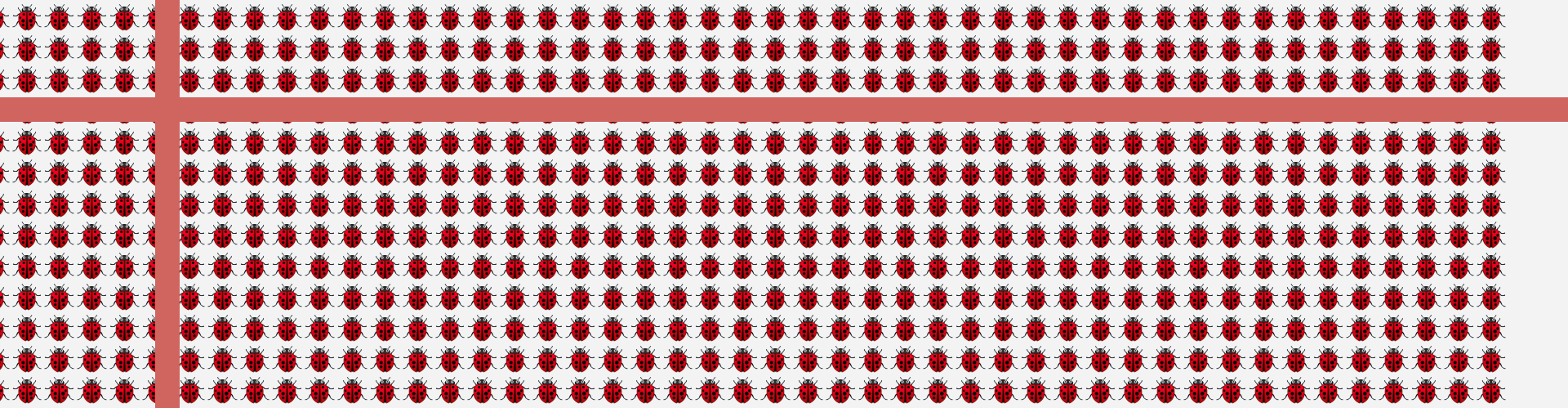


r2c

discover, exploit & eradicate

entire vulnerability classes at scale



$ whois r2c

Team of 15 at a non-traditional security/developer tools startup

Goals overlap somewhat with Mudge's CyberITL (cyber-itl.org) project but on source, rather than binaries

Relevant focus: build an “app store” for program analysis, use it to find large numbers of vulnerabilities (where others might not be looking!)

r2c = return-to-corp (a.l.a. return-to-libc)

$ whois “Isaac Evans”

- CEO @ r2c
- binary exploitation bypasses for techniques like control-flow integrity (ACM CCS ‘15) and novel hardware defenses on new architectures like RISC-V (MEng thesis) @ US DoD + MIT Lincoln Laboratory
- Made small contributions to Ghidra (ghidra-sre.org)
- BS+MS EECS at MIT

agenda

15min: Short demo of what's possible: idea to bug bounty report

45min: What is program analysis?
What are the current tools available to analyze source?

45min: Writing your first program analysis

1hr: Writing a program analysis that actually looks for something interesting

1hr: More complex analysis, refine the analysis TP/FP rate

Till end: Hunting time :)

part 1/5

Demo: analysis on ecosystems

demo

from “would anyone do this though?” concept to security report

write → run → triage → H1 report

part 2/5

What is program analysis?

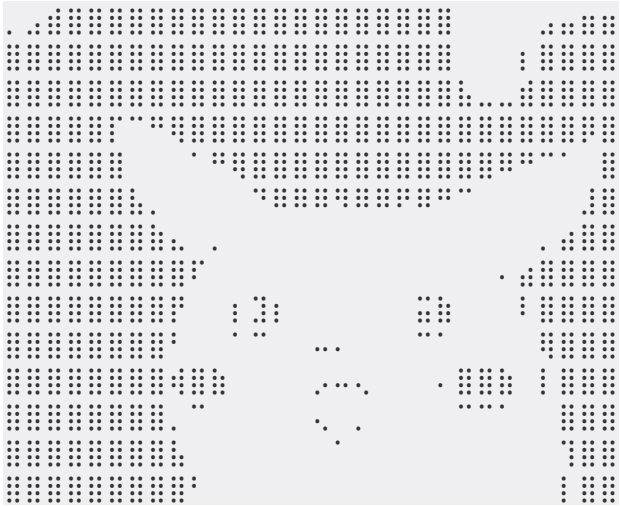
What is program analysis?

- Answer some question, measure some property about a program
- Programs are trees (trees are a kind of undirected graph)
- Many analyses reduce to *graph problems* with *domain-specific adaptations*
- Let’s look closely at a tree and then write a simple analysis

“What’s the worst that can happen?”



“What~~’s the worst that~~ can happen?”



What kind of questions?

- What values can this variable have?
- What types happen here?
- Who can influence this variable?
- How can the program get here?
- Who influences the program getting here?
- “What does this code do?”
- “What doesn’t this code do?”

What kind of problems?

- Validation, escaping sandbox, ACLs
- API usage, ACLs, correctness
- User information leakage, buffer overflows
- ACLs, error detection, find usages
- Return to libc, bypassing validation
- “Oh my God”
- “There is no God”

Lab 0: **Abstract Syntax Trees**

Step 1: pull up https://astexplorer.net

Step 2: `console.log("Hello, world!");`

Step 3: `x.map(i => i + 1)`

Step 4: Write a program that can't run, but still parses into an AST

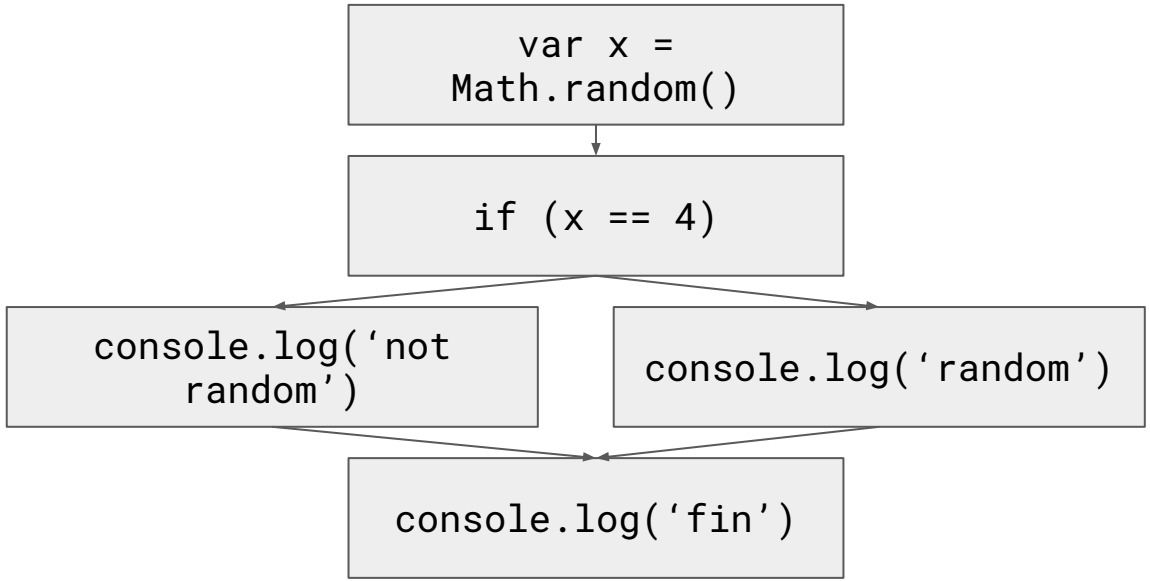
You are looking at a JSON representation of an *estree*, the representation used by Mozilla's Javascript parser (Javascript is formally specified as ECMAScript -- hence *es*)


```
{
  "type": "Program",
  "start": 0,
  "end": 29,
  "body": [
    {
      "type": "ExpressionStatement",
      "start": 0,
      "end": 29,
      "expression": {
        "type": "CallExpression",
        "start": 0,
        "end": 28,
        "callee": {
          "type": "MemberExpression",
          "start": 0,
          "end": 11,
          "object": {
            "type": "Identifier",
            "start": 0,
            "end": 7,
            "name": "console"
          },
          "property": {
            "type": "Identifier",
            "start": 8,
            "end": 11,
            "name": "log"
          },
          "computed": false
        },
        "arguments": [
          {
            "type": "Literal",
            "start": 12,
            "end": 27,
            "value": "Hello, world!",
            "raw": "\"Hello, world!\""
          }
        ]
      }
    }
  ],
  "sourceType": "module"
}
```

Kinds of Program Analysis: **Control-Flow Graph**

A graph where nodes are instructions and edges are possible transitions of the instruction pointer (execution)

```
var x = Math.random()
if (x == 4) {
  console.log(‘not random’)
} else {
  console.log(‘random’)
}
console.log(‘fin’)
```

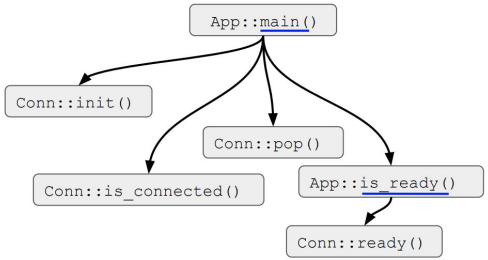


Kinds of Program Analysis: **Call Graph**

Nodes are functions; an edge exists from every function to the functions it calls

Why it's useful: does anyone even call this vulnerable function?

```
1  extern crate accepts;
2  extern crate send;
3
4  use accepts::Conn;
5  let is_ready = |conn: _| conn.ready();
6  fn main () {
7   let pipe = Conn::init();
8   let mut notifications = Vec::new();
9   while !is_ready(pipe);
10  // setup stuff
11  while pipe.is_connected() {
12   notifications.push(pipe.pop());
13  }
14 }
```



https://fosdem.org/2019/schedule/event/rust_praezi/

Kinds of Program Analysis: **Data-Flow**

Data-flow analysis: state and information flow on top of the control flow graph

Why it's useful: Detect Heartbleed (OpenSSL bug from 2014)

- Can user-controlled input reach *write_length*?
- Where is *write_length* defined?
- What variables does its definition depend on?

```
+                                                   payload + padding;
                int r;

+               if (write_length > SSL3_RT_MAX_PLAIN_LENGTH)
+                       return 0;
+
                /* Allocate memory for the response, size is 1 byte
```

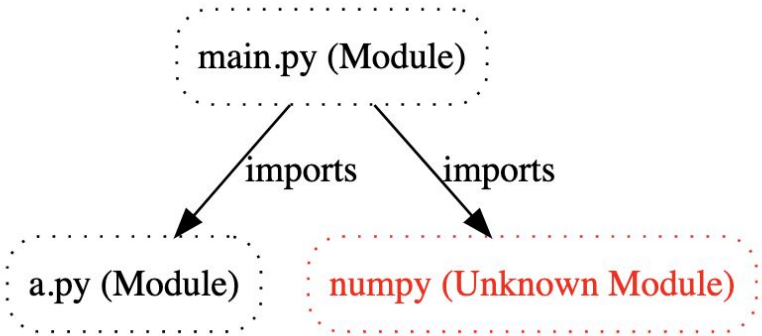
https://github.com/openssl/openssl/commit/96db9023b881d7cd9f379b0c154650d6c108e9a3

Kinds of Program Analysis: **Imports/Exports**

imports/exports analysis tells us:

- what the functions in the entire module are *exported* (available to others)
- or what exported functions from other modules it references (*imports*)

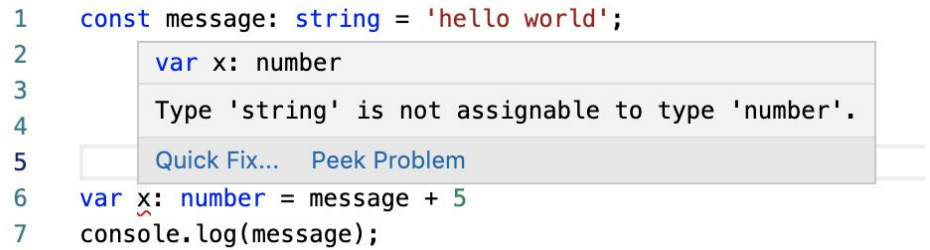
Usage example: just because we depend on a library, doesn't mean we use the vulnerable function! (and we might even use the function without exercising the vulnerable behavior)



(from https://github.com/github/semantic/blob/master/docs/examples.md)

Why Program Analysis: **Type Checking**

Simplest approach: bottom-up recursive AST type decoration



Why Program Analysis: **Type Inference**

Much more powerful; TypeScript type inference:

```
type S = { done: boolean, value: number }
type T =
    | { done: false, value: number }
    | { done: true, value: number };

declare let source: S;
declare let target: T;

target = source;
```

(from https://devblogs.microsoft.com/typescript/announcing-typescript-3-5/)

Why Program Analysis: **Vulnerabilities!**

Find vulnerable patterns:

```
strcpy(a, b)
```

Prove that vulnerable patterns are exploitable:

```
strcpy(a, b)
          ^ b comes from user input
```

In some cases, construct PoC automatically! (Usually dynamic analysis does this)

```
Fuzzer: Attekett_surku_fuzzer
Crash Type: Heap-use-after-free READ 1
Crash Address: 0x6150003faab0
 - crash stack -
 syncer::SyncBackupManager::Init
 browser_sync::SyncBackendHostCore::DoInitialize
 - free stack -
 browser_sync::SyncBackendHostCore::DoDestroySyncManager
 browser_sync::SyncBackendHostCore::OnInitializationComplete
```

(from https://bugs.chromium.org/p/chromium/issues/detail?id=395410)

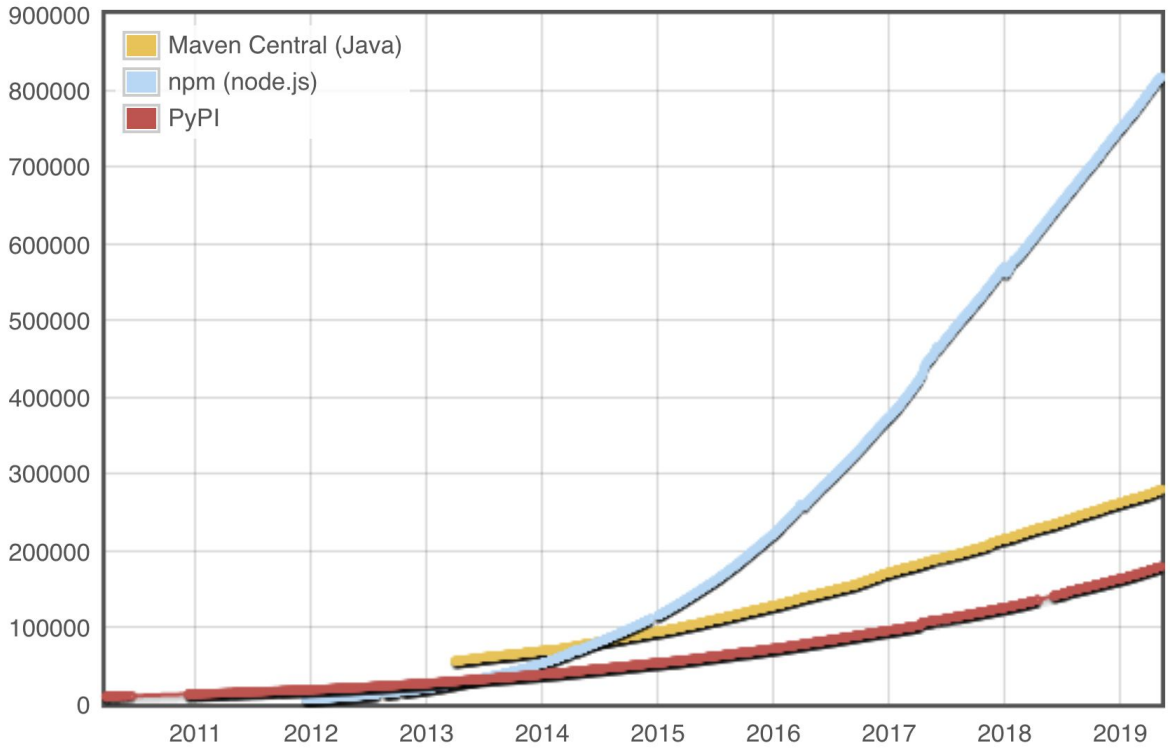
Why use program analysis to find vulnerabilities at scale?

Because the world is changing:

@dstufft, PyPi maintainer →



published packages vs. time

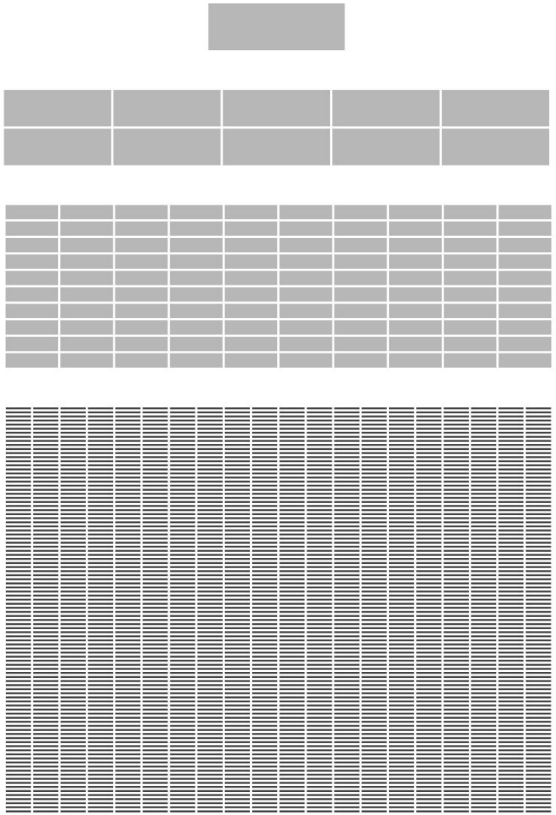


by volume, majority of JS code is 3rd party:



trust explosion

10x surface area growth per level

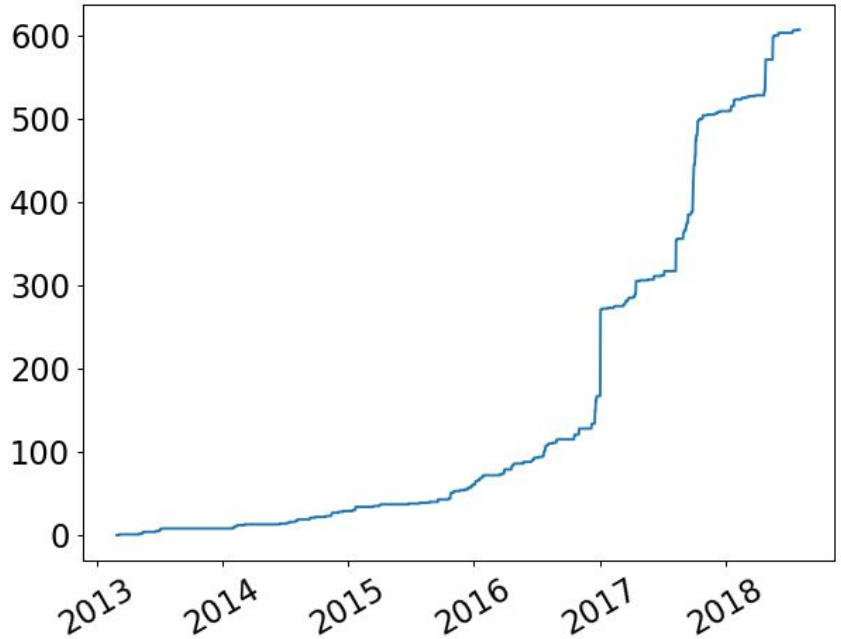


237
direct JS dependencies

1,918
Transitive JS dependencies

23,271
Independent code contributors

advisories published for npm vs. time



part 3/5

Let’s write some analysis!



security & privacy caveats

r2c is in beta; so at the moment please assume the following:

- no expectation of privacy for pushed code
- running an analyzer = arbitrary code execution (maybe)

why build analyzers in r2c format?

- **open source** CLI, tools, and docs
- easy to be **deterministic** (Docker!)
- infra for running at **scale**
- **composable** outputs with analyzers written by others

Prereqs: OS

Follow along at: https://docs.r2c.dev

Tested/Supported OS:
MacOS 10.14, Ubuntu 17.10

Ask instructors for a VM if you need it!

Prereqs: tools

docker: a tool for running software in isolated containers

python3.6+: the programming language needed to run (but not to develop) analyzers

pip3: the Python package manager used to install and update r2c-cli

Make sure everything is working

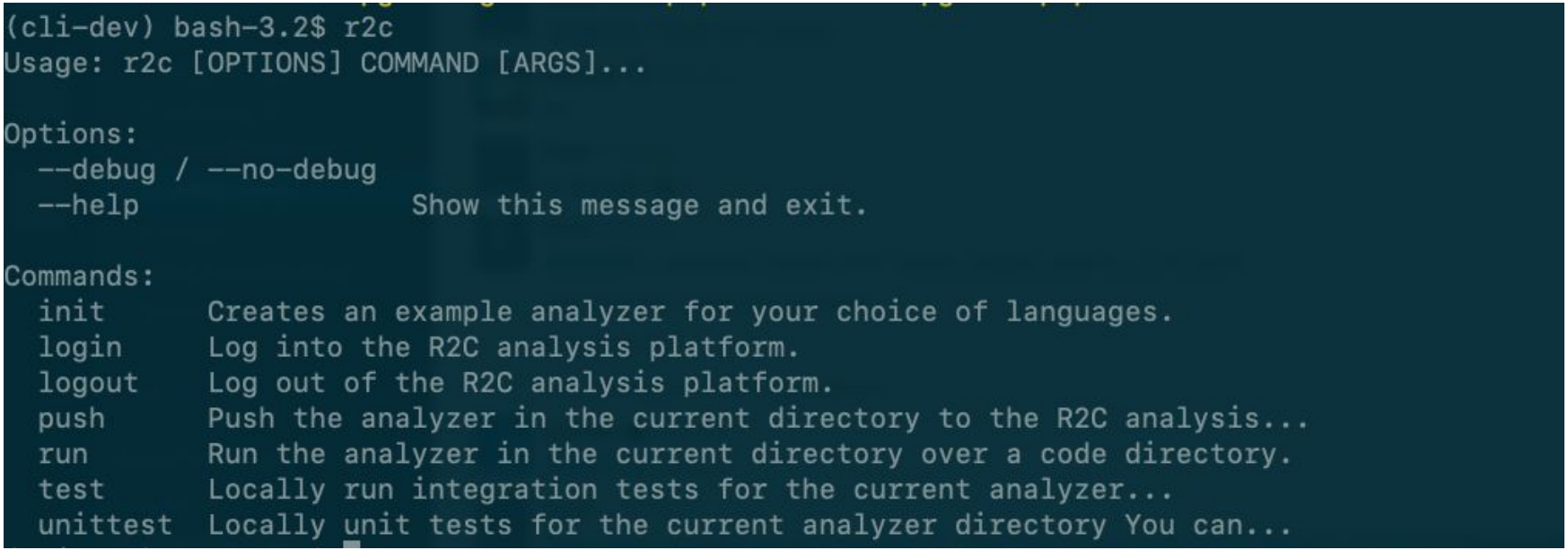
```
$ docker run hello-world
```

```
$ python3 --version # make sure its 3.6+
```

```
$ pip3 --version
```

Install the r2c cli!

```
$ pip3 install r2c-cli
```

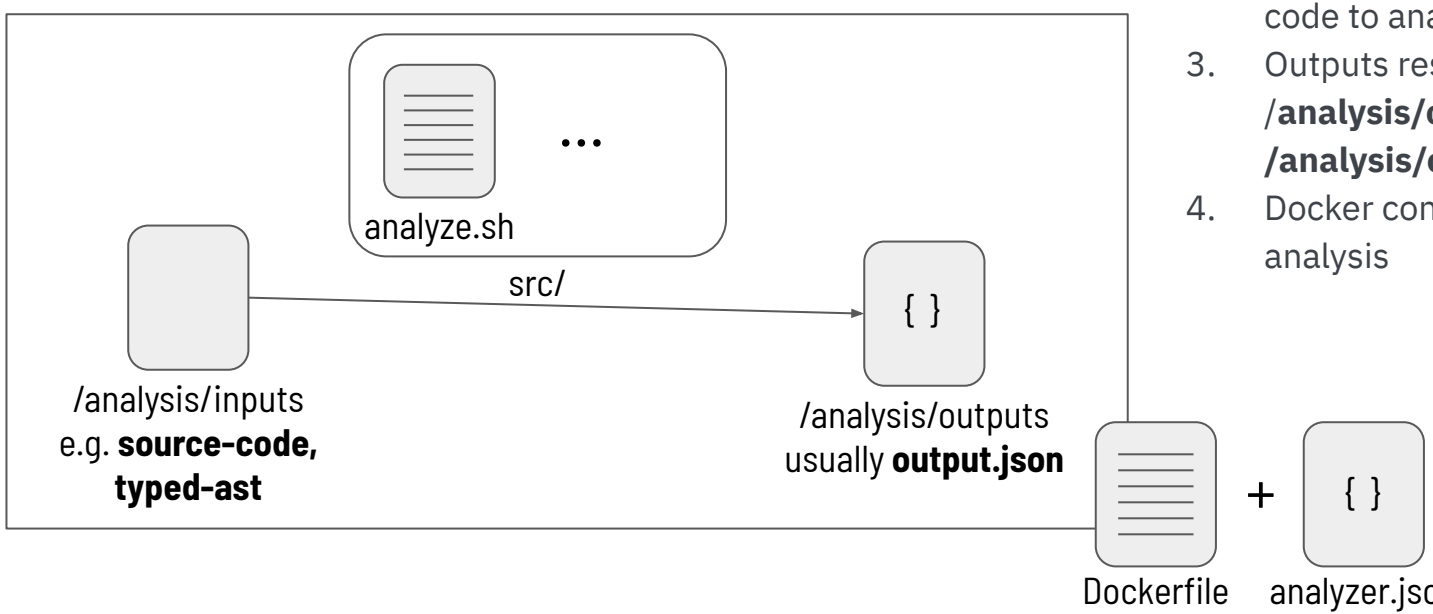


Create an analyzer

```
$ r2c init
```

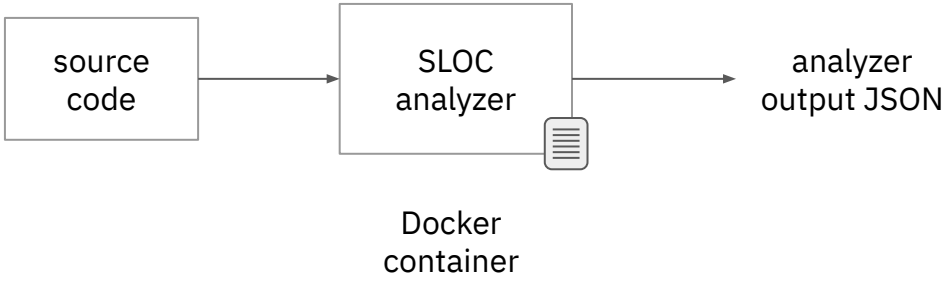
json vs *filesystem*: **choose json**
commit vs *git*: **choose commit**

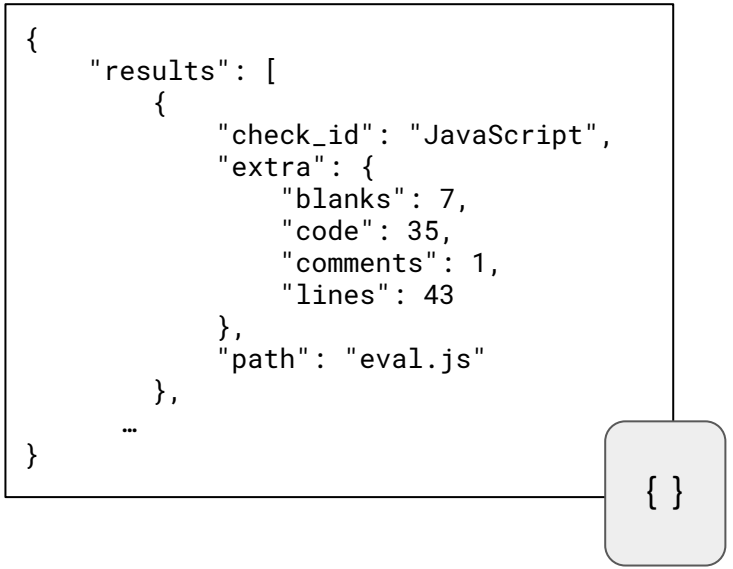
What is an analyzer?



1. An `analyze.sh` script that kicks off the actual analysis
2. Gets inputs in **/analysis/inputs** (e.g. code to analyze)
3. Outputs results to /**analysis/outputs/output.json** or **/analysis/outputs/fs**
4. Docker container that wraps the analysis

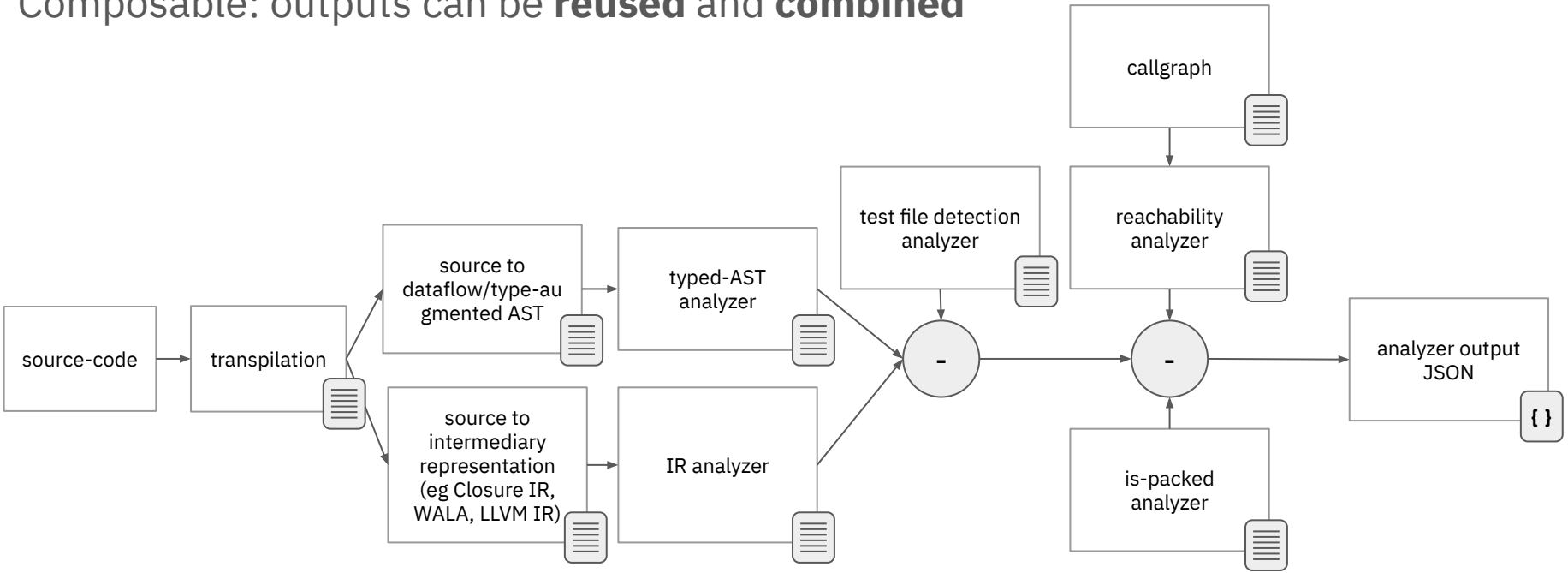
Basic Analyzer





Complex Analyzer

Deterministic components: all intermediary stages are **fully cached**
Composable: outputs can be **reused** and **combined**



What happens under the hood?

- Dockerfile is used to build a docker image
- Docker image is run for each input (e.g. github repo)
- First, all inputs of an analyzer are fetched and mounted onto the container in an `inputs/` directory
- Container is run by running its `analyze.sh` file
- Output is read out of an `outputs/` directory and saved

Understanding analyzer files

```
.
├── Dockerfile
├── README.md
├── analyzer.json
└── src
    ├── analyze.sh
    └── unittest.sh
```

Understanding analyzer files, cont’d:

`analyzer.json`:

`analyzer_name`: A namespaced analyzer name. For the beta, all analyzers are namespaced with beta automatically.

`version`: The version of the analyzer. Versions should follow semantic versioning.

`dependencies`: This analyzer depends on the most basic component, public/source-code. It specifies that it depends on any version of the public/source-code component by specifying "*" as the version. For more complicated analysis, we will depend on components such as r2c/transpiler or r2c/typed-ast later.

Understanding analyzer files (cont’d):

./Dockerfile:

- Responsible for the container’s setup and configuration.
- Docker lets us use any language because each image can have its own set of software installed
- Installs software to build and run your analysis
- Specifies `analyze.sh` as an entrypoint for when the Docker container is run

  To learn more about Dockerfiles in general, see Docker’s tutorial.

Understanding analyzer files (cont’d):

./src:

- `src/analyze.sh` is the main entry point. From this file, we’ll run your program that performs analysis!
- `src/unittest.sh` run your analyzer’s unit tests, if it has them, inside the container by calling them from this file.
- Everything else in `src/` is your code, and you want to run it from `analyze.sh`

Analyzer 1: Minifinder

- let’s find JS projects on github that have minified files
- we’ll do it by finding how much of each file is whitespace
- we’ll write this analyzer in Python
- we can test it locally on a project we’ve cloned using the r2c cli first!
- follow along at docs.r2c.dev

Analyzer 1: **Minifinder**

First, the Dockerfile, responsible for the container's setup and configuration.

Note that we're installing Python:

```
1   FROM ubuntu:17.10
2
3   RUN apt update && apt install -y python3 \
4       && rm -rf /var/lib/apt/lists/*
5
6   RUN groupadd -r analysis && useradd -m --no-log-init --gid analysis analysis
7
8   USER analysis
9   COPY src /analyzer
10
11  WORKDIR /
12  CMD ["/analyzer/analyze.sh"]
```

Analyzer 1: **Minifinder**

The meat of the analyzer: read file at the given path, produce results, write them to `output.json`

```
import json
import re
import sys

WHITESPACE_RE = re.compile("\s")


def count_whitespace(path):
    print("Counting whitespace in file {}".format(path))
    with open(path, "r", encoding="utf-8") as file:
        data = file.read()
    result = {}
    result["check_id"] = "whitespace"
    result["path"] = path
    result["extra"] = {}
    result["extra"]["size"] = len(data)
    result["extra"]["num_whitespace"] = len(WHITESPACE_RE.findall(data))
    return result


all_results = []
for path in sys.argv[1:]:
    all_results.append(count_whitespace(path))

with open("/analysis/output/output.json", "w") as output:
    output.write(json.dumps({"results": all_results}, sort_keys=True, indent=4))
```

Analyzer 1: **Minifinder**

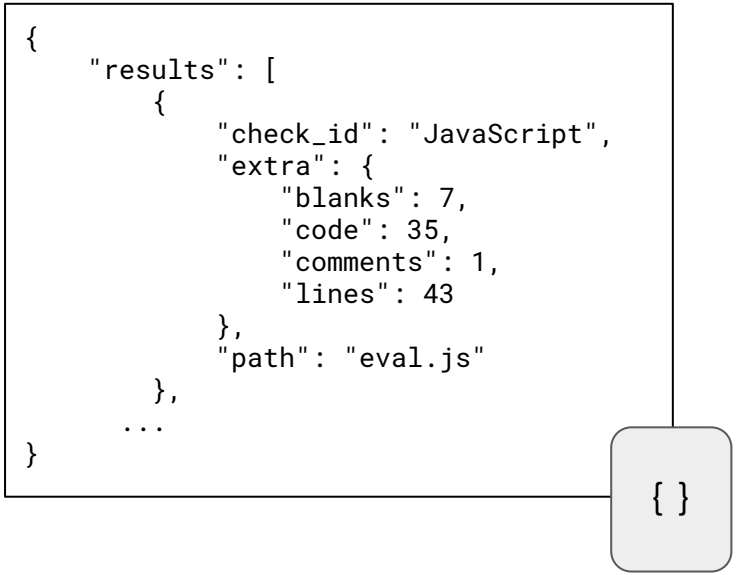
We run it from `analyze.sh`, which you may remember, is our Docker entrypoint. Note:

1. We navigating to the directory where source-code input is mounted for us
2. Find every js file and passing it to whitespace.py
3. Our code lives at `/analyzer/whitespace.py` because in our Dockerfile, we copied the `src` folder to the `/analyzer` folder inside the container.

```
1  #!/bin/bash
2
3  set -e
4  CODE_DIR="/analysis/inputs/public/source-code"
5
6  cd ${CODE_DIR}
7
8  find . -type f -name '*.js' -print0 | xargs -0 python3 /analyzer/whitespace.py
```

Analyzer 1: **Minifinder**

- we expect input in `/analysis/inputs/public/source-code/` because we're depending on the r2c `source-code` analyzer which clones repos for us
- we're expected to write to `/analysis/outputs/output.json`
- we're trying to produce results and save them in the schema at right →



Analyzer 1: **Minifinder**

Let’s run this on an actual repo like left-pad!

```
$ mkdir ~/test-repos
$ git clone https://github.com/stevemao/left-pad ~/test-repos/left-pad
```

```
$ r2c run ~/test-repos/left-pad/
🏃 Running analyzer...
Counting whitespace in file ./test.js
Counting whitespace in file ./perf/es6Repeat.js
Counting whitespace in file ./perf/perf.js
Counting whitespace in file ./perf/O(n).js
Counting whitespace in file ./index.js
{
    "results": [
...
```


```
{
    "results": [
        {
            "check_id": "whitespace",
            "extra": {
                "num_whitespace": 638,
                "size": 3673
            },
            "path": "./test.js"
        },
        {
            "check_id": "whitespace",
            "extra": {
                "num_whitespace": 61,
                "size": 216
            },
            "path": "./perf/es6Repeat.js"
        },
        {
            "check_id": "whitespace",
            "extra": {
                "num_whitespace": 213,
                "size": 1442
            },
            "path": "./perf/perf.js"
        },
        {
            "check_id": "whitespace",
            "extra": {
                "num_whitespace": 77,
                "size": 241
            },
            "path": "./perf/O(n).js"
        },
        {
            "check_id": "whitespace",
            "extra": {
                "num_whitespace": 360,
                "size": 1137
            },
            "path": "./index.js"
        }
    ]
}
```

Analyzer 1: **Minifinder**

This is nice, but we don’t know what cutoff to use: what % of whitespace is minified?

Running our analyzers on a ton of repos will give us a better idea!

Let’s push our analyzer to the r2c platform and run it on more projects!

We’ll need to:

- Log in on the r2c cli
- push our analyzer to the platform
- select a set of projects to run on from the web UI

Analyzer 1: **Minifinder**

Logging in is easy. Follow the on-screen instructions:

```
$ r2c login
Please enter your org name, or to use the common r2c platform, press enter [wildwest]:
Opening web browser to get login token. Do you want to continue? [Y/n]:
trying to open https://app.r2c.dev/settings/token in your browser...
Please enter the API token: ...
```

(*Stick with the default wildwest org name that's tied to our beta*)

Analyzer 1: **Minifinder**

Before pushing your analyzer, we need to change a few things in the `analyzer.json` file:

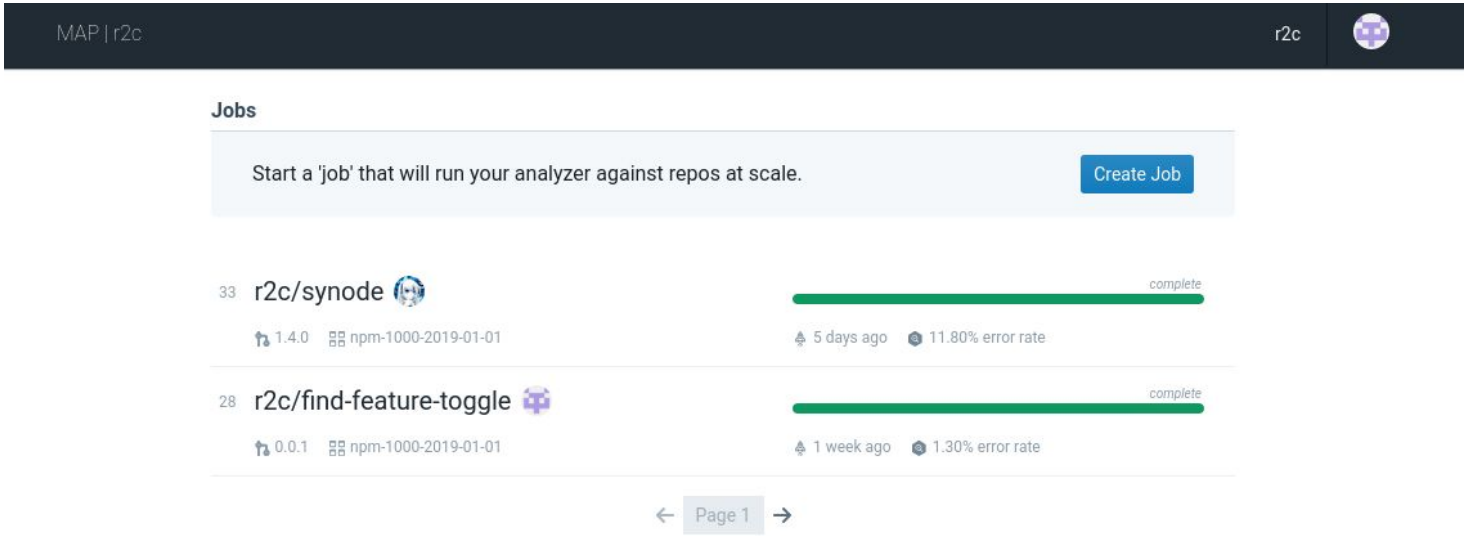
```
{
    "analyzer_name": "[YOUR GITHUB ID]/minifinder",
    "author_name": "Jav. A. Script",
    "author_email": "hello@example.com",
    "version": "0.0.1",
    "spec_version": "1.2.0",
    "dependencies": {
        "public/source-code": "*"
    },
    "type": "commit",
    "output": {
        "type": "json"
    },
    "deterministic": true
}
```

Set the name/version that you're pushing your analyzer with here! Use Semantic Versioning. Once you're ready, run:

```
$ r2c push
```

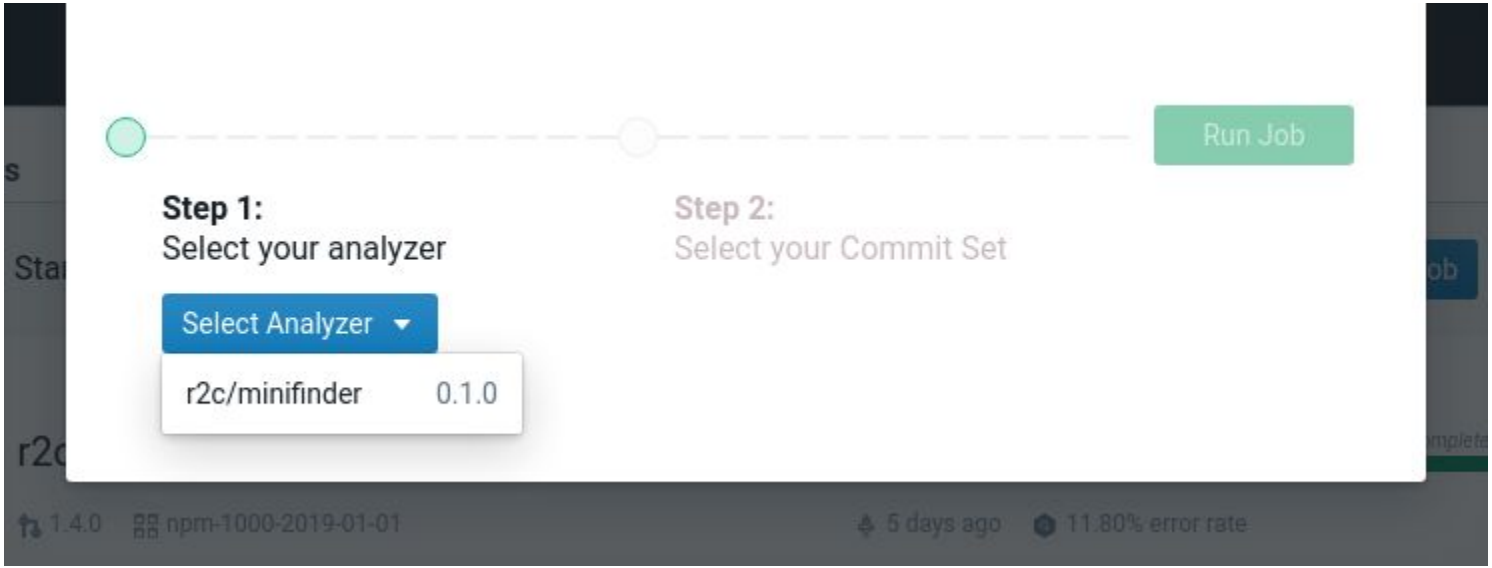
Analyzer 1: **Minifinder**

Let’s go over to the web UI at [app.r2c.dev](app.r2c.dev) and run our analyzer on lots of projects:



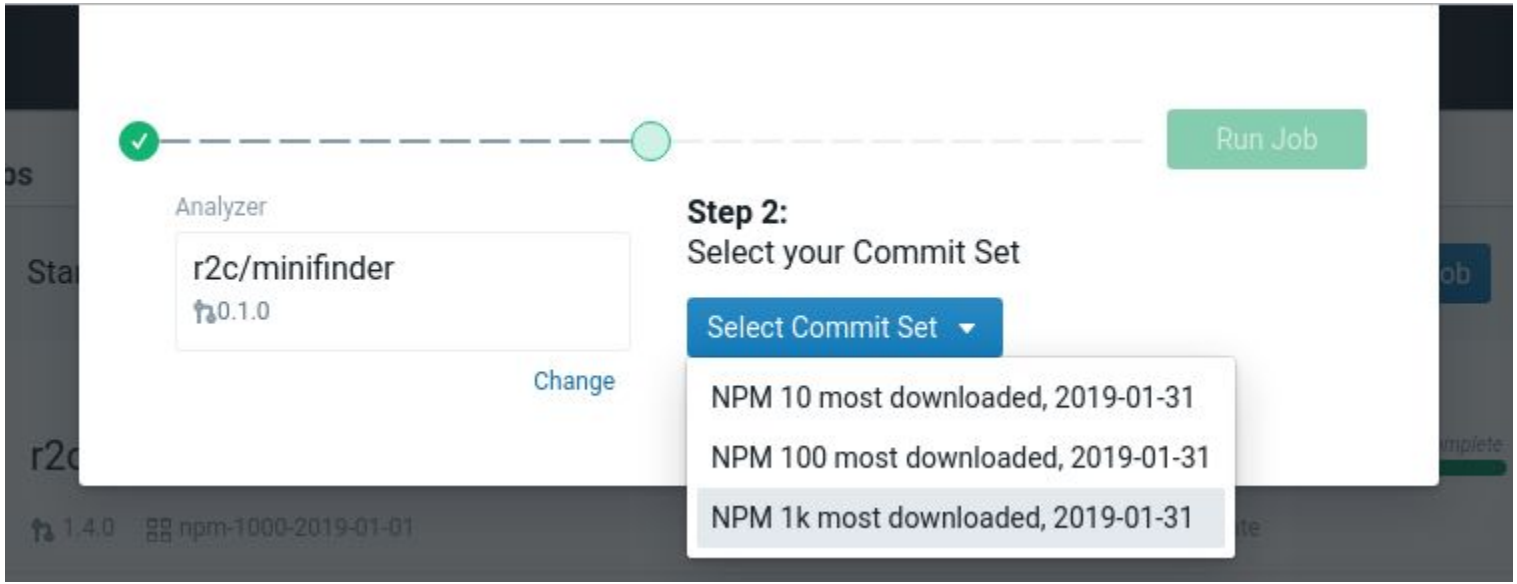
Analyzer 1: **Minifinder**

Let’s select our analyzer from the list of all analyzers on the platform:



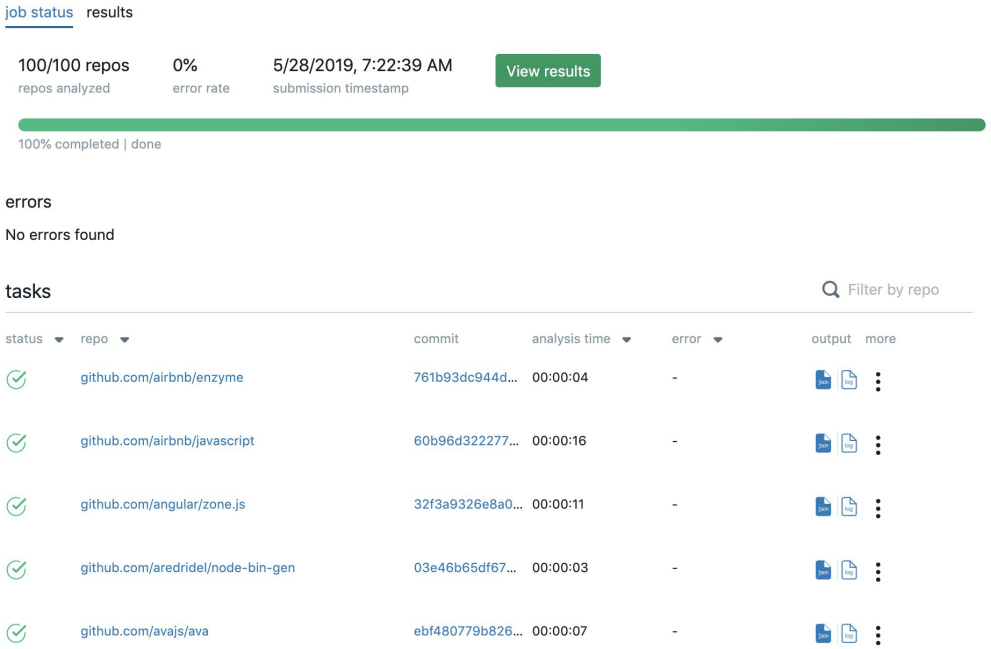
Analyzer 1: **Minifinder**

And select the set of projects we want to run it on! Let’s choose the top 100 projects on NPM.



Analyzer 1: **Minifinder**

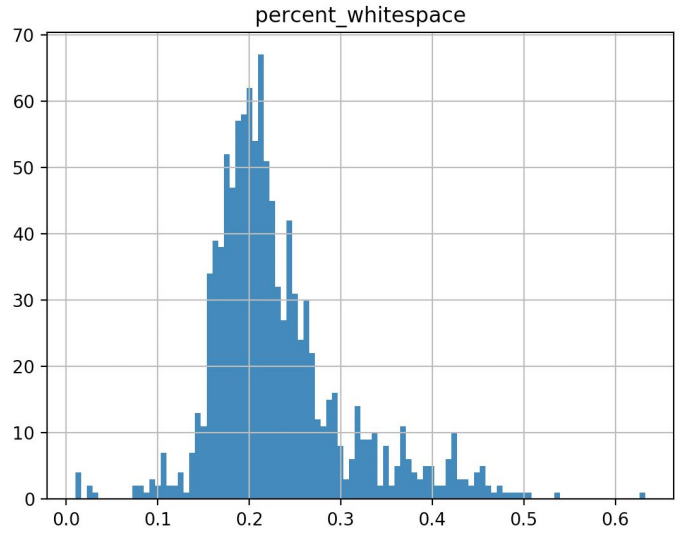
You can download the results of each run:



Analyzer 1: **Minifinder**

Or ask questions about them in aggregate:

```
# Canonical SQL query to get job-specific results back.
JOB_QUERY = """
SELECT *
FROM   result,
       commit_corpus
WHERE  result.commit_hash = commit_corpus.commit_hash
       AND analyzer_name = %(analyzer_name)s
       AND analyzer_version = %(analyzer_version)s
       AND corpus_name = %(corpus_name)s
"""

QUERY_PARAMS = {
    "corpus_name": CORPUS_NAME,
    "analyzer_name": ANALYZER_NAME,
    "analyzer_version": ANALYZER_VERSION,
}

# Connect to PostgreSQL host and query for job-specific results
engine = create_engine(f"postgresql://notebook_user:{DB_PASSWORD}@XXXXXXXXXX/postgres")
job_df = pd.read_sql(JOB_QUERY, engine, params=QUERY_PARAMS)

# Print pandas dataframe to stdout for debugging
print("Raw job dataframe:")
print(job_df[1:10])

# Helper method to compute % whitespace from the num_whitespace and size fields in our 'extra' column
def get_percent_whitespace(row):
    size = row.extra["size"]
    # Avoid 'division by zero' exceptions.
    return row.extra["num_whitespace"] / size if size else 0


# Add 'percent_whitespace' column
job_df["percent_whitespace"] = job_df.apply(get_percent_whitespace, axis=1)

# Create a histogram of our data using the `percent_whitespace` column
job_df.hist(column="percent_whitespace", bins=100)
plt.show()
```



part 4/5

I’m a real analyzer now!

(At least, for JavaScript)

Analyzer 2: **Javascript is a terrible language**

Vulnerability in the Buffer API

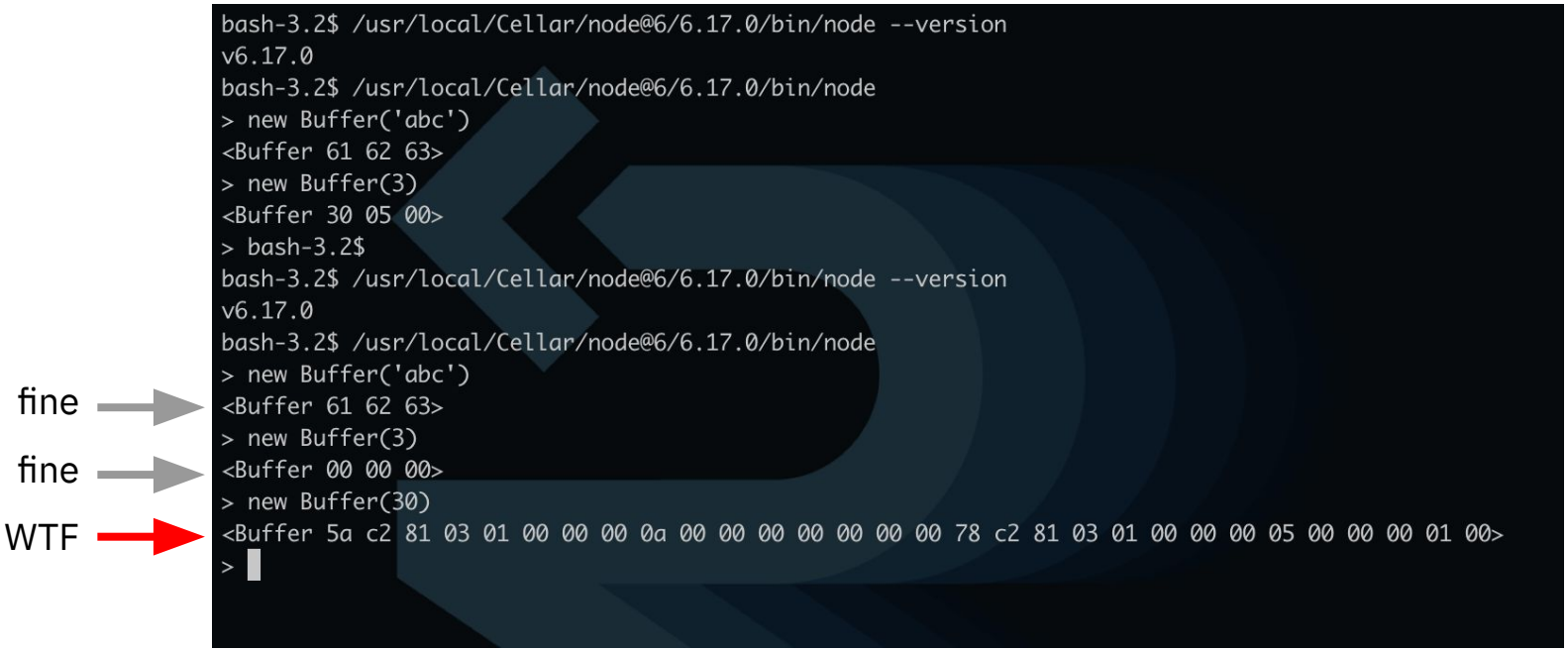
- New Buffer API in NodeJS
- Node is common in server-side/desktop apps
- Buffers in NodeJS can be constructed in the following ways:

```
const buf1 = new Buffer([1,2,3]);
// creates a buffer containing [01, 02, 03]
const buf2 = new Buffer('test');
// creates a buffer containing ASCII bytes [74, 65, 73, 74]
const buf3 = new Buffer(10);
// creates a buffer of length 10
```

On versions of Node prior to Node 6: when passed a number, the buffer constructor allocates memory without initializing (zeroing) it out!

Analyzer 2: **Javascript is a terrible language**

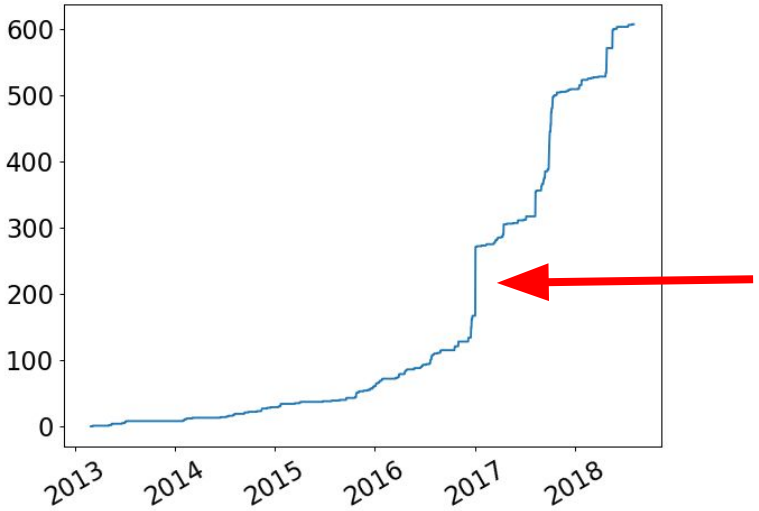
Vulnerability in the new Buffer API



Analyzer 2: **Javascript is a terrible language**

Vulnerability in the new Buffer API

- Discovered almost 3 years ago! Several instances reported
- Led to the deprecation of the API
- Vulns in mongoose, ws, request and most recently sequelize
- Remember that graph of vulnerabilities over time?



Analyzer 2: **Javascript is a terrible language**

Vulnerability in the new Buffer API

```
21  function MongooseBuffer(value, encode, offset) {
22    var length = arguments.length;
23    var val;
24
25    if (length === 0 || arguments[0] === null || arguments[0] === undefined) {
26      val = 0;
27    } else {
28      val = value;
29    }
30
31    var encoding;
32    var path;
33    var doc;
34
35    if (Array.isArray(encode)) {
36      // internal casting
37      path = encode[0];
38      doc = encode[1];
39    } else {
40      encoding = encode;
41    }
42
43    var buf = new Buffer(val, encoding, offset);
44    utils.decorate(buf, MongooseBuffer.mixin);
45    buf.isMongooseBuffer = true;
```



Analyzer 2: **Getting started**

```
$ r2c init
```

Name the analyzer: *yourname-buffer-finder*

This analyzer will be an AST visitor that reports every instance of the API -- safe or not (will have a lot of false positives--we'll fix that later)

Your goal: detect all the positive cases in r2c-easy via an ast-based analysis

Analyzer 2: **Dependencies**

The analyzer will depend on *r2c/typed-ast*

For now, all you need to know is that the AST format is **estree** (Mozilla Javascript AST representation)

https://hexdocs.pm/estree/api-reference.html

Check out an example analyzer at https://github.com/returntocorp/typed-ast-rule-example

Analyzer 2: **AST visiting**

```
import * as estree from "estree";
import { Context, Rule } from "./index";

export const rule: Rule = {
 create(context: Context) {
   return {
     JSXIdentifier(node: estree.Node) {
       if ((node as any).name == "dangerouslySetInnerHTML") {
         context.report({
           node,
           checkId: "dangerously-set-innerhtml"
         });
       }
     }
   };
 }
};
```

Analyzer 2: **Triage**

This time, after running your analyzer, try triaging the results at app.r2c.dev/triager!

Analyzer 2: **what about those false positives?**

```
function alloc(safe_var) {
  if (typeof safe_var === 'string' || safe_var instanceof String) {
     return new Buffer(safe_var)
  }
}

function broken_alloc(unsafe_var) {
  x = new Buffer(unsafe_var)
  return 4;
}

function unsafe_but_filled(unsafe_var) {
  x = new Buffer(unsafe_var)
  var i = 0;
  for (i = 0; i < x.length; i++) {
    x[i] = 0
  }
}
```

Analyzer 3: **types → dataflow**

Let’s use some of the program analysis tooling we talked about earlier to eliminate the false positives!

What we need is a dataflow analysis that can answer the question of whether a specified identifier refers to a *number* or a *string*

```
new Buffer(x: number) // bad
```

```
new Buffer(x: string) // ok
```

```
new Buffer(x: not a number or a string) // ??
```

typed-ast + stadt for JS/TS

type-based dataflow analysis by r2c

typed-ast: **JS AST with type annotations**

- JSON: estree + extra inferredType property
- Consume in any language!
- Inference: TypeScript + @types packages
  - Works even with invalid JS, with non-typed JS, etc.
- 56.4% AST nodes with nontrivial types over top 100K packags on npm
- Separate ‘definitions’ file for classes to avoid repetition

Why types?

- Can’t know if this is safe otherwise!
- string is safe, number isn’t (uninitialized memory)

```
const arg = getSomeArgument();
const buf = new Buffer(arg);
```

stadt: **serialization for TypeScript types**

- github.com/returntocorp/stadt
- MIT-licensed!
- It’s just JSON: anyone can use this for type serialization/deserialization


```
{
  "returnType": {
    "kind": "nominative",
    "name": "Promise",
    "fullyQualifiedName": {
      "builtin": true,
      "name": "Promise"
    },
    "typeArguments": [
      {
        "kind": "nominative",
        "name": "Array",
        "fullyQualifiedName": {
          "builtin": true,
          "name": "Array"
        },
        "typeArguments": [
          {
            "kind": "nominative",
            "name": "Dirent",
            "fullyQualifiedName": {
              "builtin": false,
              "fileName": "fs.d.ts",
              "packageName": "@types/node",
              "name": "Dirent"
            },
            "typeArguments": []
          }
        ]
      }
    ]
  }
}
```

Analyzer 3: **Javascript is a terrible language**

Checkout the *typed* branch of https://github.com/returntocorp/typed-ast-rule-example
(Selectively copy into your analyzer as needed)

```
.
├── CHANGELOG.md
├── Dockerfile
├── README.md
├── analyzer.json
└── src
    ├── analyze.sh
    ├── package-lock.json
    ├── package.json
    ├── src
    │   ├── index.ts
    │   └── rules
    │       └── example.ts          ← example usage of stadt
    ├── tsconfig.json
    └── unittest.sh
```

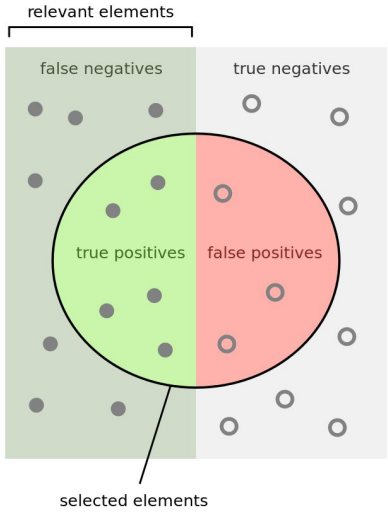
For Analyzers 2 and 3: we have test cases!

Max score: 20 true positive / 20 true negative
on *r2c-buffer-js-easy*

Max score: 60 true positive / 20 true negative
on *r2c-buffer-js-hard*

Real-world dataset (bonus): *r2c-buffer-js-cves*

(Max score: as good as you can!)



Our online scoring is based on the F-1 score:
$F_1 = 2 \times (\text{precision} \times \text{recall}) / (\text{precision} + \text{recall})$

(en.wikipedia.org/wiki/F1_score)

part 5/5

is your analyzer...web-scale?

Analyzers can be collaborated on!

Both by sharing the source, and by depending on them in the r2c registry

Collaborate work with your workshop partners & please share your results

(chat with us if you’re working on something more heavyweight, we’d love to help!)

What are the available analyzers?

What are the available analyzers?

github/

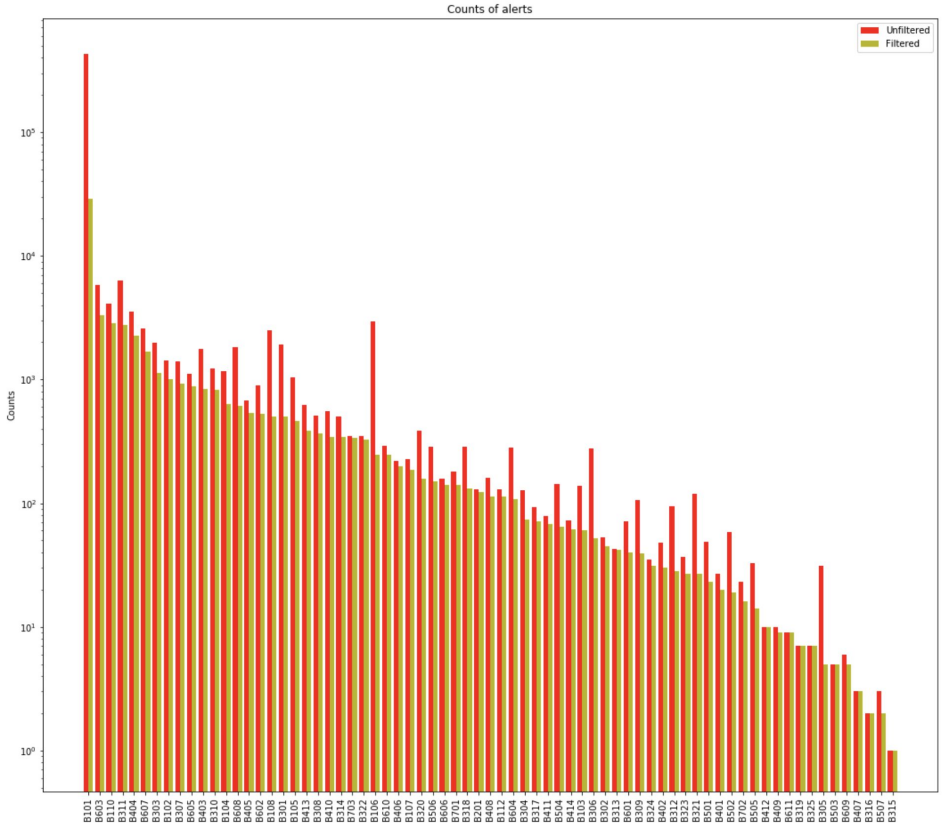
semantic

Language support

Priority	Language	Parse	Assign	Diff	ToC	Symbols	Import graph	Call graph	Control flow graph
1	Ruby	✅	✅	✅	✅	✅	✅	🚧	
2	JavaScript	✅	✅	✅	✅	✅	✅	🚧	
3	TypeScript	✅	✅	✅	✅	✅	✅	🚧	
4	Python	✅	✅	✅	✅	✅	✅	🚧	
5	Go	✅	✅	✅	✅	✅	✅	🚧	
	PHP	✅	✅	✅	✅	✅			
	Java	✅	✅	✅	🔶	✅			
	JSON	✅	✅	✅	N/A	N/A	N/A	N/A	
	JSX	✅	✅	✅	🔶				
	Haskell	✅	✅	✅	🔶	✅			
	Markdown	✅	✅	✅	🔶	N/A	N/A	N/A	

What are the available analyzers?

r2c/bandit
on PyPi



What are the available analyzers?

r2c/js-permissions

What are the available analyzers?

r2c/js-test-finder*

*worked on by several groups; currently invitation-only

What are the available analyzers?

collab/js-callgraph*

*worked on by several groups; currently invitation-only

Why all the collaborative + large scale features?

1) Build an “app store” for program analysis
2) Use it to find large numbers of vulnerabilities (where others might not be looking!)
3) Profit! Report the vulns!
4) Give feedback to the authors of the analyzers

Hunting time!

Outputs from some of the analyzers just shown are cached on 100K-1M repos/packages on r2c infrastructure: r2c.dev/triager

Ask an instructor for internal datasets about the F1 score of their various checks. Only ask: please credit r2c.dev if you file an issue/bug bounty!



r2c

Stay in touch!

Lots of people writing analysis and some of them hanging out on r2c-community.slack.com

Email **hello@r2c.dev** for an invite!