Learning Type Inference for Enhanced Dataflow Analysis

Lukas Seidel .... *Qwiet AI and TU Berlin*
Sedick David Baker Effendi .... *Stellenbosch University and Whirly Labs*
Xavier Pinho .... *Qwiet AI*
Konrad Rieck .... *TU Berlin*
Brink van der Merwe .... *Stellenbosch University*
Fabian Yamaguchi .... *Qwiet AI, WhirlyLabs, and Stellenbosch University*

28th European Symposium on Research in Computer Security (ESORICS), 2023

Our Goal

Practical Application of LLMs in Security Research

… Specifically, Static Analysis

Motivation

Why type inference?

Type information affects the **precision** of downstream static analysis, with a knock-on effect… *Call graphs, field accesses, taint tracking, etc.*

This effect is made worse when we perform **partial program analysis**[1].

[1] We do not see the whole program, e.g., code snippets, dependencies are excluded.

Motivation

Why large language models?

If we have the whole program, we see **type definitions** and **object instantiations**.

If we don't, we may need heuristics.

Where type annotations are excluded, developers may use **descriptive identifiers**.

How to reason about data like a human

From this snippet alone, can we identify the attack surface and sensitive sinks?

- **`req`** refers to some HTTP request body
- No library in particular
- reads a payload from **`req.body.params`**.

- **`documentClient`** is fetched from some global database module
- appears to be a *DynamoDB DocumentClient* object
  - **`query`** is invoked from it.

```
const db = require("db.js");
const documentClient = db.documentClient;

const handler = (req, res) => {
  const params = req.body.params;
  documentClient.query(params, function(err, data) {
    if (err) console.log(err);
    else console.log(data);
  });
};

export default handler;
```

Fig. 1: JavaScript request handler with a query to a database.

How to reason about data like a ~~human~~ machine

`req` = parameter of type **`any`**

`documentClient` = field access of type **`any`**

We do not see the internals of **`db.js`**, so we cannot tag **`documentClient`** using type information.

We could use a code heuristic for the handler:
`const handler = ($SOURCE, res)`

But how many cases do we need to consider? How flexible is our matcher?

```
const db = require("db.js");
const documentClient = db.documentClient;

const handler = (req, res) => {
  const params = req.body.params;
  documentClient.query(params, function(err, data) {
    if (err) console.log(err);
    else console.log(data);
  });
};

export default handler;
```

Fig. 1: JavaScript request handler with a query to a database.

Can machines reason like humans?

Code-LLMs appear to come close…

What should the machine see?

Can we just give the source code?

TypeBert achieved pretty decent results with this and a large GitHub database, but falls short on user-defined types…

GraphCodeBert-ManyTypes4TypeScript achieved better results for a bigger model & dataset.

Can we do better?

- Use LLM foundation model pretrained on code and documentation (CodeT5+)
- Integrate tokens into the code to precisely tag variables that need inference
  => *usage slice*
- No token classification head: encoder-decoder architecture

```
{
    "tgt": ["documentClient", "DocumentClient"],
    "def": ["db.documentClient", "require(\"db.js\")"],
    "calls": [{"name": "query", "index": 0}]
 }
```

Fig. 2: A usage slice for **`documentClient`**.

```
/* Infer types for Javascript: */
const db = require('db.js');
const documentClient: <extra_id_0>  = db.documentClient;

const handler = (req: <extra_id_1>, res) => {
  const params: <extra_id_2> = req.body.params;
  documentClient.query(params, function(err: <extra_id_3>, data: <extra_id_4>) {
    if (err) console.log(err);
    else console.log(data);
  });
};

export default handler;
```

Fig. 3: Annotated code with tokens on target variables.

Implementation Target

The open-source static analysis platform Joern:

- Language agnostic
- Supports partial programs
- Rich intermediate code representation for number of downstream tasks

Joern uses the **code property graph** (CPG) IR.

The CPG is the combination of **abstract syntax**, **control-flow**, and **data-dependence** information.

We perform usage slicing on the CPG.

The Architecture

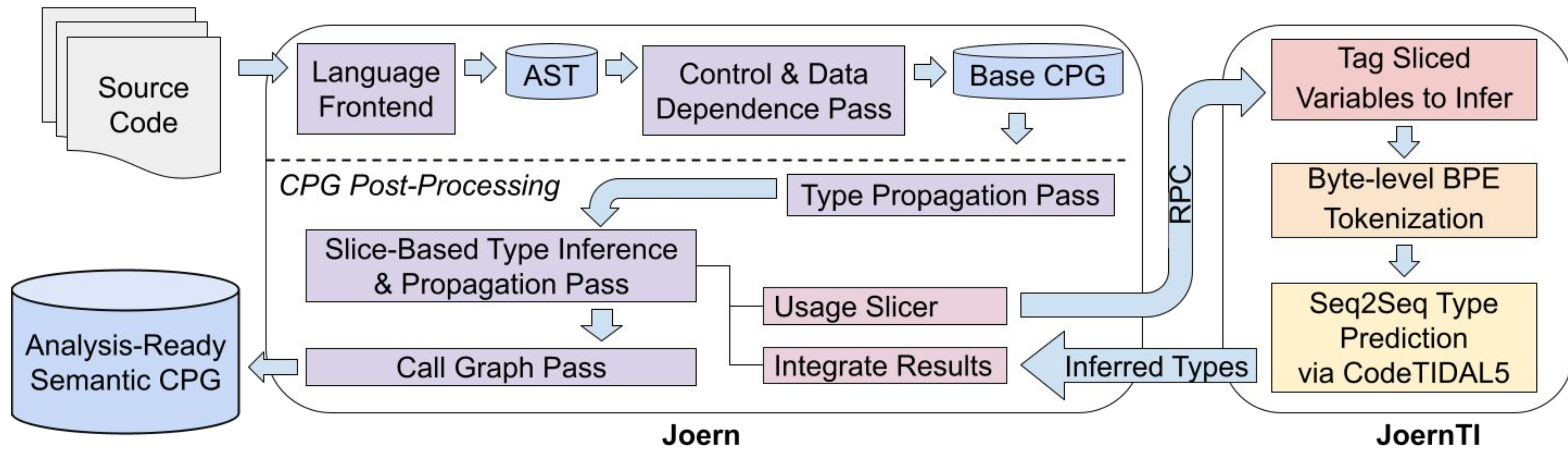


Fig. 3: End-to-end pipeline of Joern's code property graph construction with the JoernTI neural type inference server.

Performance & Evaluation

Compared neural inference models:

- LambdaNet
- TypeBert
- GraphCodeBert-MT4TS

Datasets

- LambdaNet (LN)
- ManyTypes4TypeScript (MT4TS)

*Excluding: **`Function`** and **`void`** types, and situations where constructors are present.*

Model	Top-1 Acc %			Size
	User-Defined	Top-100	Overall	
LambdaNet	46.89	64.79	61.18	N/A
TypeBERT	51.50	73.30	68.90	360M
GCBert-4TS	46.89	**80.62**	**73.81**	162M
CodeTIDAL5	**53.20**	79.77	73.61	220M

Table 1: Performance comparison of ML-based on LambdaNet dataset. Size in number of trainable parameters.

Model	Top-1 Acc %	
	Top-100	Overall
TypeBERT	48.92	28.07
GCBert-4TS	87.22	63.42
CodeTIDAL5	**90.03**	**71.27**

Table 2: Performance comparison on the ManyTypes4TypeScript dataset.

Manual Labelling & Real-World Testing

Does the model generalize outside of benchmarks?

1093 type inferences from 10 open-source JavaScript projects are manually labelled.

Hardware: M1 MacBook Pro 16GB RAM.

Category	Manual Labelling Results %			
	Correct	Partial	Useful	Incorrect
Built-in	76.60	0.00	1.25	22.14
User-Defined	63.63	10.43	6.15	19.79
Overall	72.10	3.57	2.93	21.41

```
const db = require("db.js");
const documentClient = db.documentClient;

const handler = (req, res) => {
  const params = req.body.params;
  documentClient.query(params, function(err, data) {
    if (err) console.log(err);
    else console.log(data);
  });
};

export default handler;
```

documentClient	DocumentClient
req	NextApiRequest
params	Record
err	ecma.Error
data	DocumentClient.QueryInput

Querying the Resulting Graph

The additional types allow us to keep and make use of CPG queries that **match type information**.

Queries of this kind are superior to code-matching, as they are **robust to syntactic differences.**

Also allows us to provide **meaningful feedback** to the user with API or library specific recommendations.

```
def src = cpg.identifier
  .typeFullName(".*(express.|NextApi|__ecma.)Request")
  .inFieldAccess.code(".*\\.body\\..*")
def sink = cpg.identifier
  .where(_.and(_.typeFullName(".*DocumentClient"),
               _.argumentIndex(0)) // Receivers are at 0
  ).inCall.name("query")
sink.reachableBy(src)
```

nodeType	tracked	lineNumber	method	file
Call	const params = req.body.params	5	handler	handler.js
Identifier	const params = req.body.params	5	handler	handler.js
Identifier	documentClient.query(params...	6	handler	handler.js
Identifier	documentClient.query(params...	6	handler	handler.js
Call	documentClient.query(params...	6	handler	handler.js

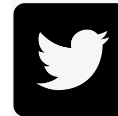
Fig. 4: CPG query tracking flow from an HTTP request parameter to a `DocumentClient` query call.

Conclusion

- CodeTIDAL5 offers SOTA type inference, especially for **user-defined types**
- JoernTI integrates this into practical static analysis workflows on **developer hardware**
- Access to **more type information** during downstream static analysis such as data-flow analysis
- Ready-to-use **integration of LLM** to assist in real-world security analysis

Future Work

- Extend to multiple languages
- Improve detection of incorrect/invalid inferences



@SDBakerEffendi
@pr0me

Backup Slides

Machine Learning Model

CodeTIDAL5

Saleforce's CodeT5 220M parameter model base.

Features to learn

Semantic relationships, i.e, variable naming conventions and class names.

Motivation

CodeT5 achieves SOTA for NLP-PL tasks where *understanding semantics* are required.

salesforce/**CodeT5**

Home of CodeT5: Open Code LLMs for Code Understanding and Generation

2 Contributors | 43 Issues | 2k Stars | 342 Forks

Mitigating Errors & Hallucination

Can we detect invalid type inferences?

The user has the ability to specify **TypeScript declaration files** to compare the usage slice against for inconsistent properties. These are filtered out for fewer false positives.

Example:

`lib.es5.d.ts` tells us `String` does not have a `.body` property, so `req` should not be a `String` *(in most cases)*

Do we infer everything?

No, as **not everything has a usage** (or needs it).

The user can specify the **minimum number of usages** a variable requires to be sent for inference. This increases the known context and may boost accuracy.

If, for example, a constructor is in the file, Joern's type propagation would likely solve the missing type information for that variable.