CALLME: Call Graph Augmentation with Large Language Models for Javascript

Michael Wang
MIT CSAIL
Cambridge, MA 02139, USA
mi27950@mit.edu

Kexin Pei
Department of Computer Science
University of Chicago
Chicago, IL 60637, USA
kpei@uchicago.edu

Armando Solar-Lezama
MIT CSAIL
Cambridge, MA 02139, USA
asolar@csail.mit.edu

Abstract

Building precise call graphs for Javascript programs is a fundamental building block for many important software engineering and security applications such as bug detection, program repair, and refactoring. However, resolving dynamic calls using static analysis is challenging because it requires enumerating all possible values of both the object and the field. As a result, static call graph construction algorithms for Javascript ignore such dynamic calls, resulting in missed edges and a high false negative rate. We present a new approach, CALLME, that combines Language Models (LMs) with a custom static analyzer to address this challenge. Our key insight is in using LMs to incorporate additional modalities such as variable names, natural language documentation, and calling contexts, which are often sufficient to resolve dynamic property calls, but are difficult to incorporate in traditional static analysis. We implement our approach in CALLME and evaluate it on a dataset of call edges that are dependent on dynamic property accesses. CALLME achieves 80% accuracy and .79 F1, outperforming the state-of-the-art static analyzer by 30% and .60, respectively. To study the effectiveness of CALLME on downstream analysis tasks, we evaluate it on our manually curated dataset with 25 known Javascript vulnerabilities. CALLME can detect 24 vulnerabilities with only 3 false positives, whereas static analysis tools based on current call graph construction algorithms miss all of them.

1 Introduction

Call graph construction has been a prerequisite for many critical software analysis tasks, such as code optimization (Fink et al., 2008; Malavolta et al., 2023), bug detection (Brown et al., 2017; Cai et al., 2023a; 2021), taint analysis (Kang et al., 2023), and software maintenance and inspection (Feldthaus et al., 2013). However, dynamic languages such as Javascript present unique challenges for call graph construction. The dynamic nature of Javascript alongside the size of sophisticated frameworks such as React and AngularJS makes call graph construction very difficult. For example, a recent survey on Javascript call graph construction (Antal et al., 2023) has shown that even the best static analyzer only obtains a 0.43 detection F1 score, suffering from high false positives and false negatives. Moreover, the majority of the existing static analyzers tested simply ignore the method calls when they are across multiple files (Antal et al., 2023).

A key challenge for constructing call graphs statically is Javascript's flexible object model, which allows properties to be created and deleted at runtime. Specifically, dynamic property accesses, where the property being accessed depends on a runtime-computed string, are esti-

mated to cause 70% of missed edges in static call graph construction algorithms (Chakraborty et al., 2022). Recent static call graph construction algorithms specifically ignore reasoning about most dynamic property accesses due to their runtime-dependent behavior, and thus miss any calls that are computed dynamically from objects (Feldthaus et al., 2013; Nielsen et al., 2021a).

Figure 1 shows an example of a dynamic property access, where the property access call is made in line 13 to a function field on the object `o`. Analyzing the call requires a field-sensitive pointer analysis to determine the points-to set of `o["greet"+firstUser]`. However, the property names of `o` are computed dynamically in a loop in lines 3-8. Computing a field-sensitive pointer analysis with the presence of dynamic property accesses is thus a prohibitively expensive process. The presence of dynamic properties increase pointer analysis runtime from $O(N^3)$ to $O(N^4)$, where N is the size of the program (Sridharan et al., 2012). As a result, traditional field-sensitive analyses are unable to handle large Javascript frameworks such as *jQuery* and *react*.

```
1  var o = {};
2  function createGreetFunctions() {
3    for (user of this.getUsers()) {
4      var greetUser = "greet" + user;
5      o[greetUser] = new function(){
6        console.log(arguments[0] + \\
7                    " " + user);
8      }
9    }
10 }
11 createGreetFunctions();
12 var firstUser = this.getUsers()[0];
13 o["greet" + firstUser]("hello");
```

Figure 1: A basic example of a dynamic property access function call on line 13. The value of `firstUser` is determined at runtime, making it difficult for static analyzers.

In this paper, we introduce CALLME, an approach to specifically target dynamic property access calls for Javascript call graph construction. CALLME has two stages: statement selection and inference.

1. **Statement selection.** We develop JSelect, a custom static analyzer that efficiently selects the relevant statements needed to determine if a call site calls a specified function.
2. **Inference.** The output of JSelect is further processed by a Language Model (LM) which determines whether the call site can call the function.

Importantly, CALLME does not replace the need for traditional call graph construction algorithms. Rather, CALLME is intended to augment traditional call graph construction specifically for dynamic property accesses. To the best of our knowledge, CALLME is the first static analysis system capable of resolving dynamic property accesses without runtime information.

Reasoning about dynamic property accesses for Javascript makes a good target for LMs for several reasons. First, existing solutions are already unsound and incomplete, so a solution that works well in practice can be competitive even without theoretical guarantees – introducing an unsound LM does not worsen any existing guarantees. Second, LMs can take advantage of useful sources of information that are difficult to encode as traditional static analysis rules. In Figure 1, for example, it is fairly straightforward to infer that the function call made in line 13 likely refers to the function defined in line 5, given the variable names such as "greet" and "User". Similarly, inferring the relationship between words such as "greet" and "hello" often relies on knowledge of natural language. Additionally, LMs can infer high-level design patterns, such as visitors and builders, which can be challenging to incorporate into traditional static analysis.

We evaluate CALLME on a dataset of caller-callee edges collected by dynamic analysis (Chakraborty et al., 2022), where we specifically target dynamic property accesses. Importantly, the dataset consists only of calls that were missed by Approximate Call Graph (ACG) (Feldthaus et al., 2013), a recent call graph construction algorithm. CALLME is able to resolve 75% of calls with an F1 score of 0.79. Jelly, the recent open source implementation based off of the state-of-the-art static analysis framework JAM (Nielsen et al., 2021a), is only able to resolve 11% of these calls with an F1 score of .19 by handling dynamic property accesses that start or end with hard-coded strings, *e.g.*, `obj["foo" + y]()`.

We show how CALLME can be applied downstream program analysis tasks such as vulnerability detection by resolving edges that are currently undetectable by call graph construction algorithms due to their use of dynamic property accesses. We manually searched multiple datasets of known Javascript bugs in real-world projects and identified 25 function calls across 23 projects resulting in bugs or vulnerabilities that are undetectable with current call graph construction due to dynamic property accesses. These security bugs include prototype pollution, cross-site scripting, command injection, and arbitrary file overwrites. We build a scanner using CALLME to search for these vulnerable function calls, which resolves 24 out of 25 vulnerable calls with only 3 false positives.

2 Motivation

As an example to motivate our approach, consider a simplified code snippet (Figure 2) from the jQuery framework (openjsf.org) containing a call that is ignored by static analyzers. In the following, we discuss several features of Javascript that make analyzing the call difficult.

Functions are objects. Functions are objects which can have assigned fields themselves. This is shown on line 9, where the initial jQuery object is a function but also has a field called `extend`. Functions can also be passed around as objects, as shown in the `each` function on line 28.

Dynamic additions/uses. Properties and fields can be added to objects dynamically, such as adding `extend` to the jQuery object on line 9, and then using `extend` to add the `each` and `show` properties. The properties can then be overwritten, as shown on line 30.

Arity mismatching. Functions can be called with any number of arguments, as shown in the `extend` function on lines 8 and 10.

Computed names. Properties can be read and written by computed names, as seen on line 29. Additionally, precise modeling of functions is necessary for an analyzer. In our example, an analyzer would need to model the functionality of `each` in order to understand what happens on line 28, and would need to model the functionality of `extend` to understand what happens on line 17. Without the explicit modeling of `extend`, there is no way for a static analyzer to know that the `showAll` and `each` functions are added to jQuery itself.

```
1  jQuery = function(selector, context); {
2    return new jQuery.prototype.init();
3  }
       1.Functions are objects.
4
5
6  2.Dynamic additions/uses.
7                          3.Arity mismatching.
8
9  jQuery.extend = function() {
10   for (i=0; i < arguments.length; i++ ) {
11     for ( name in arguments[i] ) {
12       this[name] = options[name];
13     }
14   }
15 };
16
17 jQuery.extend( {
18   each: function( obj, cb ) {
19     for ( i in obj ) {
20       cb.call(obj[i], i, obj[i]);
21     }
22   },
23   showAll: function() {
24     return showHide( this, true );
25   }
                                4.Computed names.
26 });
27
28 jQuery.each(["show"],function(i,name){
29   var cssFn = jQuery.fn[name + "All"];
30   jQuery[name] = function(speed){
31     return speed == null ?
32       cssFn.apply(this,arguments) :
33       this.animate(name, speed);
34   };
                  Dynamic property access call.
35 });
```

Figure 2: A modified code snippet taken from jQuery showing several features of Javascript that make static analysis difficult. The dynamic property access is on line 29, and the function call is on line 32. However, information such as the variable names allow an LM to successfully resolve the call.

However, for a human looking at the code snippet in Figure 2, there are several things that indicate that the function call `cssFn.apply(this,arguments)` on line 32 can refer to the function `showAll` on line 23. There are natural language hints and background knowledge of programming patterns that are helpful. For example, the anonymous function defined on line 28 takes two parameters, `i` and `name`, a common pattern where

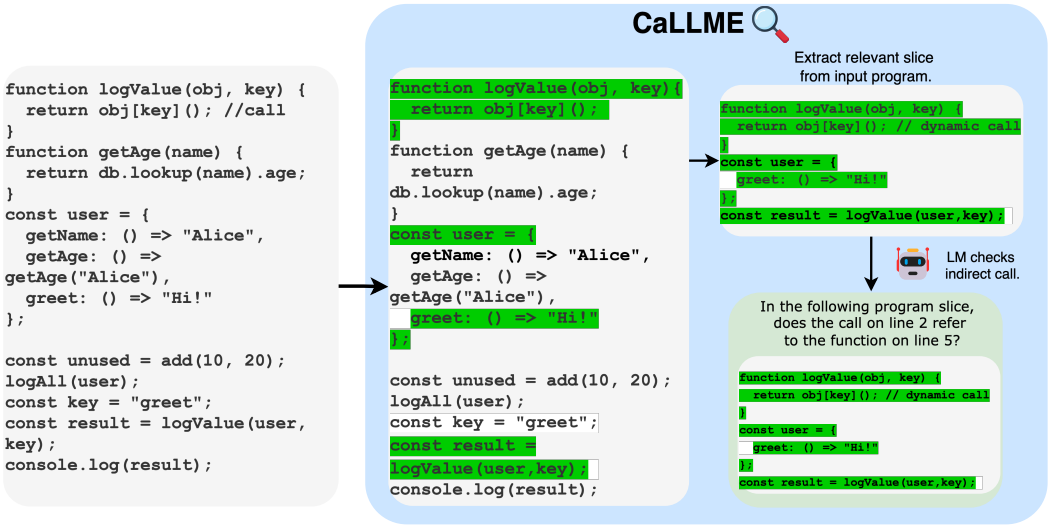

Figure 3: An overview of our system CALLME. JSelect selects the relevant statements from the original program, which then gets formulated into a prompt for an LM.

`i` refers to an index and `name` refers to the value of a list. `each` is a word commonly used in programming when iterating over lists, so it is fairly easy to infer that `jQuery.each` is iterating through each element in the list of its first argument, even without looking at the definition of `each` on line 18. While these types of heuristics are very difficult to encode in a traditional static analyzer, they can be incorporated into machine learning models and large language models. CodeLlama-34B-Instruct, a 34-billion parameter large language model optimized for code generation and understanding released by Meta AI in 2023 (Rozière et al., 2024a), successfully resolves the call on line 32.

3 Approach

CodeLlama-34b is able to identify the call in Figure 2 because of its small size at 35 lines. In a real-world program, the call site and target function could be several thousand lines apart, with most of the code being irrelevant to the function call we are trying to resolve. Thus, we need a way to automatically find the relevant lines to feed into the LM. We implement a custom statement selection algorithm, JSelect, which outputs the relevant lines to be formulated into a prompt for an LM. An overview of CALLME can be seen in Figure 3.

```
1  can.each(options, function(attr) {
2    options.attr = attr;
3    if ( Scanner.attributes [attr]) {
4      Scanner.attributes [attr](options, el);
5    };
6  });        Global variables are defined elsewhere.
```

Figure 4: We want to provide local information around the call site as well as information about the global variable `Scanner` whose definition is several hundred lines away from the call on line 4 in the snippet.

3.1 JSelect

Our statement selection algorithm, JSelect, is an intra-procedural def-use analysis combined with a simple window around the call site. We show an example of our statement selection in Appendix B in Figure 6 and Table 7 with other statement selection methods we tested.

Typically, static program slicing (Weiser, 1984) would be used to identify the relevant statements to the function call. The goal of static slicing is to identify all statements that can affect a specified variable, and has been used with machine learning algorithms to perform type inference for Python (Yan et al., 2023). Unfortunately, static program slicing for Javascript runs into the same problem as call graph reconstruction, as it requires a pointer analysis to collect the data flow graph. Sridharan *et al.* showed that a traditional field-sensitive pointer analysis has an $O(N^4)$ runtime for an N-statement Javascript program (Sridharan et al., 2012). In Section 4.1, we show that a field-sensitive pointer analysis is unable to finish on large Javascript libraries within 12 hours. To the best of our knowledge, there are no static slicing algorithms using field-sensitive pointer analysis for Javascript that scale to large frameworks like React. Instead, we opt for a cheap and scalable analysis in JSelect to identify the relevant statements for the call sites.

Call site statement selection. The intuition behind our approach is to collect the immediate context in which the call is invoked, as well as to capture references to global objects. For example, in Figure 4, we want to capture that the call on line 4 is happening inside of an `if` statement which is inside of a loop. However, we also want to provide information about the global variable `Scanner`.

The implementation of JSelect relies on parsing Javascript AST's to extract variable information. We build our implementation using Esprima (esp) and Esrefactor (Ariya), two static analysis tools for parsing Javascript AST's. First, JSelect performs a static scope analysis in order to each variable to its set of references and declaration. JSelect takes in a call site location and identifies what variables are being referenced in the call itself (in Figure 4, the variables are `Scanner.attributes, attr, options, el`). JSelect collects each of the statements that modifies or uses one of the variables. The final output of JSelect which is sent to the LM contains all of these statements along with the statements immediately surrounding the call site. This approach combines the local context with information about any global variables that are being referenced. JSelect is not path or flow-sensitive, and is not inter-procedural. An inter-procedural analysis would require a pre-existing call graph, leading to a chicken-and-egg problem.

Target function statement selection. There is often useful information around the function definition for the target function (If we are trying to determine whether call site `A` calls function `B`, the target function is `B`). For example, the target function may be assigned as a field to an object. In this case, it would be very helpful to include the entire object in the input to the LM. To obtain the relevant information, we use Esprima (esp) to obtain the AST node corresponding to the target function. We then provide the statements of the parent of the target function node to the model. If the parent node is too big (sometimes the parent node is the entire program), we use a window of 50 lines before and after the target function. We set the limit of the parent node at 200 lines.

3.2 Prompt Formulation

Once we have filtered the relevant statements from the programs, we form the input to the language model. We base our prompting strategy off of how a human might resolve an indirect call. Our prompting has three steps: program understanding, call site intention inference, and final prediction. Our final prompting template can be seen in Figure 5 in Appendix A.

Program understanding. First, the model should gain an understanding of what each statement in the program is doing. We ask the model to interpret the slice from JSelect line-by-line to get a general understanding of the code. Section 4.2 shows the performance of CALLME with different methods of interpreting the code line-by-line, such as asking the model to explain each line or asking the model to simulate the execution line-by-line.

Call site intention inference. Next, we ask the model to reason about the call site itself. We provide the call, the line number, as well as information on the variables referenced in the call site. This provides additional information about what the call is being used for and helps determine whether it is likely to be a match with the target function provided later.

Final prediction. Finally, we ask the model to predict whether the call site in question refers to the specified function. By this point, the model has reasoned about the program snippet as a whole as well as how the call site of interest is used.

4 Evaluation

Our primary baseline is Jelly, an open-source call graph construction algorithm written by the authors of JAM (Nielsen et al., 2021a), which handles dynamic property accesses with string prefixes and postfixes as well as indirect calls. The static analysis for Jelly is based off of three static analysis tools for Javascript – JAM (Nielsen et al., 2021a), ACG (Feldthaus et al., 2013), and Tapir (Møller et al., 2020a). First, we analyze how well CALLME performs in resolving dynamic property calls. Next, we explore how the design decisions in CALLME affect the performance. Finally, we show how CALLME can be used with downstream program analysis tasks such as bug detection. We conducted the experiments on a Linux server with two AMD EPYC 7763 64-Core Processors, 128 cores, 1024GB RAM, and 4 NVIDIA RTX 6000 Ada Generation GPUs.

Dataset. We use the dataset from Chakraborty et al. (2022) root cause analysis of Javascript call graphs (Chakraborty et al., 2022), consisting of caller-callee pairs generated by performing dynamic analysis on the popular TodoMVC Suite (Chakraborty et al., 2022). We identified 660 caller-callee pairs that were missed by ACG due to a dynamic property access. For each caller-callee pair, we generate a negative example by selecting a random function for the same caller for a total of 1,320 total caller-callee pairs evenly split between positive and negative samples. Statistics for each framework in the TodoMVC Suite can be found in Table 1.

Program	#Lines	Inter-File	Total
AngularJS	12,091	41	207
Backbone	9,003	22	46
KnockoutJs	1,044	5	24
KnockbackJs	15,836	24	85
CanJs	11,371	34	91
React	24,855	11	57
Mithril	1,433	25	27
Vue	7,667	15	61
VanillaJs	751	0	14
jQuery	9,526	7	48
Total	**93,557**	184	660

Table 1: Dataset statistics. **Inter-file** refers to the number of caller-callee edges that are in different files, and **Total** refers to the total number of calls.

4.1 Performance on TodoMVC

Table 2 presents the results of CALLME while using different LM backends. We test the CodeLlama models (Rozière et al., 2024b), Llama-3.3-70B-Instruct (Grattafiori et al., 2024), and GPT-4 (Achiam et al., 2023). CALLME with CodeLlama-34b achieves an F1 score of .79 and is able to detect almost 7x more calls as Jelly while maintaining a tolerable false positive rate. Additionally, as shown in Table 9 in Appendix D, 39/71 of the calls detected by Jelly are in Knockback.js, due to Knockback using more dynamic property accesses with string concatenations. Excluding Knockbackjs, Jelly can only detect 5.5% of calls, while CALLME is able to detect 75% of all calls in our dataset. Additionally, as shown in Table 8 in Appendix D, results remain stable even when calls are resolved across different files. GPT-4 and Llama-3.3's false positive rates are much lower than everything else, with almost

Model	Detect	Miss	FP	Acc.	F1	Prec.	Recall
Jelly	71	589	19	.501	.19	.79	.11
CodeLlama-7B	490	170	339	.61	.71	.59	.74
CodeLlama-13B	435	225	230	.66	.66	.65	.66
CodeLlama-34B	497	163	102	.80	.79	.83	.79
Llama-3.3-70B	280	380	18	.70	.59	.94	.42
GPT-4	292	368	13	.71	.61	.96	.44

Table 2: Final results on the TodoMVC benchmark using different models for the final inference. FP stands for False Positive.

9x fewer false positives than the next lowest model, but suffers from lower recall. CALLME is configurable with different LM backends, so users can use CodeLlama-34b or GPT-4 based on their precision requirements.

Runtime. As noted in prior work (Sridharan et al., 2012), a field-sensitive pointer analysis is intractably slow for large programs. To test, we ran a pointer analysis using two existing static tools on the 5 largest files in the TodoMVC benchmark. WALA performs a standard Andersen's alias analysis with call site abstractions. TAJS is a static analysis tool based on abstract interpretation. TAJS errored out on all of the programs except for React due to unsupported Javascript features. We set the time-out threshold at 12 hours. As CALLME is meant to augment existing static analyzers, we use an analogous setup for our experiment. We run Jelly on each program in Table 3 and find all calls that do not have a target function. We then run CALLME once for each call. This is likely an overapproximation of CALLME's runtime in practice, as we do not anticipate CALLME being used to construct an entire call graph. However, as seen in Table 3, CALLME is still able to scale to much larger programs than a traditional field-sensitive pointer analysis.

Program	#Lines	#Calls	Total Runtime (Hours)		
			CALLME	WALA	TAJS
AngularJS	28,363	1,349	2.6	T.O (12+)	N/A
React	21,641	2,307	4.5	T.O (12+)	T.O (12+)
Jquery	9,205	728	1.4	T.O (12+)	N/A
Ractive	9,133	900	1.7	T.O (12+)	N/A
Blocks	14,724	1,031	2	T.O (12+)	N/A
Average	16,613	1,263	2.5	N/A	N/A

Table 3: Runtime performance of CALLME compared to a field-sensitive pointer analysis. TAJS errored out on everything except for React. #Calls refers to the number of call sites which Jelly does not return any target functions.

4.2 Ablations

We tested multiple statement selection methodologies and prompting formats to achieve the best tradeoff between scalability and accuracy. We perform the same experiment as in Section 4.1 on the TodoMVC dataset.

Statement selection. We evaluate the effect of different statement selection methodologies in CALLME's performance. Our goal is to find the fastest analysis that still gets robust performance. We show an example program in Figure 6, as well as the output of various slicers in Table 7. We test using a simple window around the caller and callee, a program slicer tracking the flow of function values (Feldthaus et al., 2013), thin slicing (Sridharan et al., 2007), and following def-use information of variables used in the call sites. Additional information on each of the statement selection methodologies can be found in Appendix B.

Slicing Method	Detect	Miss	FP	Improve (+/-)		
				Detect	Miss	FP
Def-Use + Window	**497**	**163**	**102**	**0%**	**0%**	**0%**
Window Only	531	129	299	+7.1%	-25.6%	+193.1%
Def-Use Only	491	169	195	-1.0%	+4.3%	+91.1%
Thin-Slicing	452	208	170	-9.7%	+27.8%	66.7%
Full-Slicing	438	222	163	-13.2%	+36.4%	+59.8%

Table 4: Ablation on various slicing methods. We treat the first row as the baseline and then compute the improvement of other slicing methods. Green = Performance improvement. Red = Performance decrease. Using a window slightly improves the detection and miss rates, but greatly increases the number of false positives.

The results of the slicing ablation study can be found in Table 4, where we find that combining Def-Use chains and a simple window performs best. The improvement seems to come primarily in the false negative and false positive rates, where the next best slicing method still has 60% more false positives. We find evidence that incorporating static analysis leads to

dramatic improvements over only using LMs. Simply using the 50 statements before and after the call sites almost triple the false positive rate. This is likely due to the fact that simply using a window includes the context in which a call is made, but likely does not include information about the relevant variables used in the call itself.

Prompting. We tested multiple prompting strategies, such as asking the model to perform abstract interpretation, asking the model to analyze the code in English, and directly prompting the model. Details on all our prompting strategies can be found in Appendix C.

The results of the study can be found in Table 5, where we find that asking the model to perform abstract interpretation has the best results. We can see that giving the model instructions as well as generating a high level summary through abstract interpretation or English interpretation of the code significantly improves performance over the *direct* and *explain* approaches. Interestingly, abstract interpretation performs better than English. This is likely due to it being slightly more granular and providing better information about the individual variables.

Prompt Method	Detect	Miss	FP	Improve (+/-) Detect	Miss	FP
Abstract Interpret.	**495**	**162**	**102**	**0%**	**0%**	**0%**
English Interpret.	450	209	94	-10%	+29%	-8.5%
Instruct	490	169	165	-1.0%	+4.3%	+59%
Two-Step	304	171	119	-62.8%	+5.6%	+16.7%
Direct	525	135	277	+6.1%	-20%	+171.6%
Explain	471	189	225	-5.1%	+16.7%	+120.1%

Table 5: Ablation on various prompts. We treat the first row of each design as the baseline and compute the improvement of other alternatives. Green = Performance improvement. Red = Performance decrease.

4.3 Bug detection.

In order to determine CALLME's effectiveness in a downstream program analysis task, we use CALLME to help identify bugs which rely on resolving dynamic property accesses. We simulate a taint analysis scenario, where the analysis needs to discover all paths from any tainted sources to vulnerable sinks. For example, a user might want to locate all call sites to library function which writes information to the filesystem to ensure that it is free from unsafe user input. An example of a real-world security vulnerability that requires resolving a dynamic property call can be found in Appendix E in Figure 7.

Our dataset includes multiple examples of security bugs that require resolving dynamic property access calls such as Cross Site Scripting, Arbitrary File Overwrites, Prototype Pollution, Improper Access Control, and Remote Code Execution.

Case study methodology. We manually searched through several hundred examples from three datasets of known Javascript bugs, SecBench.js, BugAid, and Vulnerable Functions in the Wild for bugs which required resolving dynamic property accesses and found 24 separate bugs. For each, we manually identified the target buggy function in question. Next, we built an automated detection tool which performs the following steps:

1. Scans the code for dynamic property access calls with Esprima (esp) and EsRefactor (Ariya), which returns all the locations in the program where dynamic property access calls occur.
2. For each call discovered by our scanner, ran CALLME to obtain the relevant statements to the call site.
3. Formulates the statements into a prompt as described in Section 3 with the vulnerable function to determine whether the call could refer to the vulnerable function or not and query CodeLlama-34b.

Our final dataset has 25 dynamic property access calls which resolve to vulnerable functions and 66 which do not.

Code obfuscation. Language models trained on code have been shown to be brittle to semantics-preserving program transformations (Miceli-Barone et al., 2023; Zeng et al., 2022). To test our CALLME's robustness, we obfuscate our samples using UglifyJS (Mishoo) and repeat our experiment. We replace all variable names with single letters, such as `a` and `b`, and compress the AST with UglifyJS's built-in compressor. It is important to note that analyzing obfuscated code is a *significantly* harder task. Prior work shows that language models are very brittle to name changes (Miceli-Barone et al., 2023), as replacing random variable names causes *an average of 81% performance decrease in BLEU-4 on code summarization (Zeng et al., 2022)*.

Case study results. As shown in Table 6, CALLME successfully resolves 24 true positives with only three false positives and a single false negative. Closer introspection into the failures show that the false negative uses function parameters as part of the indirect call, which are not handled by JSelect as discussed in Section 3. Additionally, performance remains relatively robust on an obfuscated dataset, where CALLME can still identify almost 67% of true positives.

Prediction	Normal		Obfuscated	
	True Pos.	True Neg.	True Pos.	True Neg.
Pred. True	24	3	17	9
Pred. False	1	63	8	57

Table 6: Results of case study on obfuscated and un-obfuscated samples.

5 Related Work

Program analysis for Javascript. There have been several approaches to static analysis for Javascript (Lee et al., 2012; Jensen et al., 2009; Sridharan et al., 2012; Møller et al., 2020b; Nielsen et al., 2021b; Li et al., 2022; Kang et al., 2023). Several tools also explicitly target call graph construction (Feldthaus et al., 2013; Nielsen et al., 2021a; Madsen et al., 2015). Additionally, these systems have difficulty scaling up to large programs, as shown in Section 4.1. Madsen *et al.* generates a call graph using static analysis, but only handles event listeners. ACG (Feldthaus et al., 2013) and JAM (Nielsen et al., 2021a) generate call graphs for Javascript programs that can handle multiple files and are scalable, but ignore dynamic property accesses.

There have been several analysis techniques to apply additional analysis specifically for certain features such as dynamic reads and writes (Park et al., 2013; Ko et al., 2019; Stein et al., 2019; Kim et al., 2014; Madsen & Andreasen, 2014; Park et al., 2016). One popular technique is to use dynamic information to focus a static analyzer on dynamic structures collected at runtime, and has led to many improvements in reasoning about dynamic data (Wei et al., 2016b; Wei & Ryder, 2014b; Wei et al., 2016a; Wei & Ryder, 2013; 2014a; 2012; Wei, 2012; Chakraborty et al., 2024; Laursen et al., 2024). However, these approaches are not purely static like CALLME, as they require the program to be executed to collect the runtime information.

LMs for program analysis. LMs have been used for many program analysis task such as type inference (Peng et al., 2023; Wei et al., 2023; Wang et al., 2023b), fuzzing (Xia et al., 2024; Yang et al., 2023b;a; Deng et al., 2023), vulnerability detection (Mathews et al., 2024; Liu et al., 2023), resource leak detection (Wang et al., 2023a; Mohajer et al., 2023), code summarization (Cai et al., 2023b; Geng et al., 2024; Ahmed et al., 2024; Wang et al., 2022), and fault localisation (Wu et al., 2023). DLInfer (Yan et al., 2023) and TypeGen (Peng et al., 2023) both use static program slicing as well as a machine learning model for type inference on Python. CALLME differs from the prior work as it is the first to perform call graph construction. Unlike many prior works, CALLME is meant to augment current static analysis tools on certain edge cases, rather than replacing them. To the best of our knowledge, CALLME is the first work to explicitly tackle an inter-procedural analysis task like call graph construction using LMs.

6 Limitations

Single link prediction. CALLME is designed to predict a single edge between a specified call site and a function. Thus, it is not scalable to build an entire call graph using CALLME, and is not designed for a task such as IDE support like ACG (Feldthaus et al., 2013) on its own. However, as many program analysis tasks only require discovering a single link, CALLME can still be useful in many cases as demonstrated in Section 4.3. Additionally, CALLME is meant to be used alongside existing algorithms to handle some difficult edge cases rather than replacing them.

7 Conclusion

We present a new approach, CALLME, which combined static program analysis with LLMs to perform call graph construction for Javascript. Given a full program, CALLME first runs the static analysis, JSelect, to select the relevant statements. Next, CALLME formulates a prompt with the output of JSelect and queries an LLM to determine whether a specified call site calls a target function. We evaluate CALLME on the TodoMVC benchmark (Chakraborty et al., 2022) and find that CALLME is able to resolve 75% of all calls with an F1 score of .79. The next best tool, Jelly, is only able to resolve 11% of these calls with an F1 score of .19. Additionally, we show how CALLME can find vulnerabilities that are currently undetectable by static analyzers, where we are able to find 24 out of 25 vulnerabilities with only 3 false positives.

References

URL https://esprima.org/doc/index.html.

URL https://www.npmjs.com/package/find-process.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. Automatic semantic augmentation of language model prompts (for code summarization), 2024.

Gábor Antal, Péter Hegedűs, Zoltán Herczeg, Gábor Lóki, and Rudolf Ferenc. Is javascript call graph extraction solved yet? a comparative study of static and dynamic tools. *IEEE Access*, 11:25266–25284, 2023. doi: 10.1109/ACCESS.2023.3255984.

Ariya. Ariya/esrefactor. URL https://github.com/ariya/esrefactor.

Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in javascript bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 559–578, 2017. doi: 10.1109/SP.2017.68.

Yuandao Cai, Peisen Yao, and Charles Zhang. Canary: practical static detection of inter-thread value-flow bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pp. 1126–1140, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454099. URL https://doi.org/10.1145/3453483.3454099.

Yuandao Cai, Peisen Yao, Chengfeng Ye, and Charles Zhang. Place your locks well: Understanding and detecting lock misuse bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 3727–3744, Anaheim, CA, August 2023a. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/cai-yuandao.

Yufan Cai, Yun Lin, Chenyan Liu, Jinglian Wu, Yifan Zhang, Yiming Liu, Yeyun Gong, and Jin Song Dong. On-the-fly adapting code summarization on trainable cost-effective language models. In A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 56660–56672. Curran Associates, Inc., 2023b. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/b16e6de5fbbdcb2df237aa66b302bc17-Paper-Conference.pdf.

Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs. In Karim Ali and Jan Vitek (eds.), *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 3:1–3:28, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-225-9. doi: 10.4230/LIPIcs.ECOOP.2022.3. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2022.3.

Madhurima Chakraborty, Aakash Gnanakumar, Manu Sridharan, and Anders Møller. Indirection-bounded call graph analysis. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, pp. 10–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models, 2023.

Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 752–761, 2013. doi: 10.1109/ICSE.2013.6606621.

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), may 2008. ISSN 1049-331X. doi: 10.1145/1348250.1348255. URL https://doi.org/10.1145/1348250.1348255.

Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3608134. URL https://doi.org/10.1145/3597503.3608134.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su (eds.), *Static Analysis*, pp. 238–255, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03237-0.

Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, V. N. Venkatakrishnan, and Yinzhi Cao. Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability. In *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 1059–1076, 2023. doi: 10.1109/SP46215.2023.10179352.

Se-Won Kim, Wooyoung Chin, Jimin Park, Jeongmin Kim, and Sukyoung Ryu. Inferring grammatical summaries of string values. In Jacques Garrigue (ed.), *Programming Languages and Systems*, pp. 372–391, Cham, 2014. Springer International Publishing. ISBN 978-3-319-12736-1.

Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. Weakly sensitive analysis for javascript object-manipulating programs. *Software: Practice and Experience*, 49(5):840–884, 2019.

doi: https://doi.org/10.1002/spe.2676. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2676.

Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. Reducing static analysis unsoundness with approximate interpretation. *Proceedings of the ACM on Programming Languages*, 8 (PLDI):1165–1188, 2024.

Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. 2012. URL https://api.semanticscholar.org/CorpusID:14633517.

Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 143–160, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/li-song.

Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhi Li, and Limin Sun. Harnessing the power of llm to support binary taint analysis, 2023.

Magnus Madsen and Esben Andreasen. String analysis for dynamic field access. In Albert Cohen (ed.), *Compiler Construction*, pp. 197–217, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-54807-9.

Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node.js javascript applications. *SIGPLAN Not.*, 50(10):505–519, oct 2015. ISSN 0362-1340. doi: 10.1145/2858965.2814272. URL https://doi.org/10.1145/2858965.2814272.

Ivano Malavolta, Kishan Nirghin, Gian Luca Scoccia, Simone Romano, Salvatore Lombardi, Giuseppe Scanniello, and Patricia Lago. Javascript dead code identification, elimination, and empirical assessment. *IEEE Transactions on Software Engineering*, 49(7):3692–3714, 2023. doi: 10.1109/TSE.2023.3267848.

Noble Saji Mathews, Yelizaveta Brus, Yousra Aafer, Meiyappan Nagappan, and Shane McIntosh. Llbezpeky: Leveraging large language models for vulnerability detection, 2024.

Antonio Valerio Miceli-Barone, Fazl Barez, Ioannis Konstas, and Shay B. Cohen. The larger they are, the harder they fail: Language models do not recognize identifier swaps in python, 2023.

Mishoo. Mishoo/uglifyjs: Javascript parser / mangler / compressor / beautifier toolkit. URL https://github.com/mishoo/UglifyJS.

Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. Skipanalyzer: A tool for static code analysis with large language models, 2023.

Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. Detecting locations in javascript programs affected by breaking library changes. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020a. doi: 10.1145/3428255. URL https://doi.org/10.1145/3428255.

Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. Detecting locations in javascript programs affected by breaking library changes. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020b. doi: 10.1145/3428255. URL https://doi.org/10.1145/3428255.

Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of node.js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, pp. 29–41, New York, NY, USA, 2021a. Association for Computing Machinery. ISBN 9781450384599. doi: 10.1145/3460319.3464836. URL https://doi.org/10.1145/3460319.3464836.

Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Semantic patches for adaptation of javascript programs to evolving libraries. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 74–85, 2021b. doi: 10.1109/ICSE43902.2021.00020.

OpenJS Foundation openjsf.org. URL https://jquery.com/.

Changhee Park, Hongki Lee, and Sukyoung Ryu. All about the with statement in javascript: removing with statements in javascript applications. In *Dynamic Languages Symposium*, 2013. URL https://api.semanticscholar.org/CorpusID:11618912.

Changhee Park, Hyeonseung Im, and Sukyoung Ryu. Precise and scalable static analysis of jquery using a regular expression domain. *SIGPLAN Not.*, 52(2):25–36, nov 2016. ISSN 0362-1340. doi: 10.1145/3093334.2989228. URL https://doi.org/10.1145/3093334.2989228.

Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. Generative type inference for python, 2023.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024b.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024a. URL https://arxiv.org/abs/2308.12950.

Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pp. 112–122, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1250734.1250748. URL https://doi.org/10.1145/1250734.1250748.

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In James Noble (ed.), *ECOOP 2012 – Object-Oriented Programming*, pp. 435–458, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31057-7.

Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. Static analysis with demand-driven value refinement. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi: 10.1145/3360566. URL https://doi.org/10.1145/3360566.

Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pp. 382–394, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549113. URL https://doi.org/10.1145/3540250.3549113.

Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. Llm-based resource-oriented intention inference for static resource leak detection, 2023a.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation, 2023b.

Jiayi Wei, Greg Durrett, and Isil Dillig. Typet5: Seq2seq type inference using static analysis, 2023.

Shiyi Wei. Blended analysis for javascript: a practical framework to analyze dynamic features. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pp. 101–102, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315630. doi: 10.1145/2384716.2384758. URL https://doi.org/10.1145/2384716.2384758.

Shiyi Wei and Barbara Ryder. A practical blended analysis for dynamic features in javascript. 08 2012.

Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pp. 336–346, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321594. doi: 10.1145/2483760.2483788. URL https://doi.org/10.1145/2483760.2483788.

Shiyi Wei and Barbara G. Ryder. State-sensitive points-to analysis for the dynamic behavior of javascript objects. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, pp. 1–26, Berlin, Heidelberg, 2014a. Springer-Verlag. ISBN 9783662442012. doi: 10.1007/978-3-662-44202-9_1. URL https://doi.org/10.1007/978-3-662-44202-9_1.

Shiyi Wei and Barbara G. Ryder. Taming the dynamic behavior of javascript. In *Proceedings of the Companion Publication of the 2014 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '14, pp. 61–62, New York, NY, USA, 2014b. Association for Computing Machinery. ISBN 9781450332088. doi: 10.1145/2660252.2660393. URL https://doi.org/10.1145/2660252.2660393.

Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. Revamping javascript static analysis via localization and remediation of root causes of imprecision. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pp. 487–498, New York, NY, USA, 2016a. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950338. URL https://doi.org/10.1145/2950290.2950338.

Shiyi Wei, Franceska Xhakaj, and Barbara G. Ryder. Empirical study of the dynamic behavior of javascript objects. *Softw. Pract. Exper.*, 46(7):867–889, jul 2016b. ISSN 0038-0644. doi: 10.1002/spe.2334. URL https://doi.org/10.1002/spe.2334.

Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984. doi: 10.1109/TSE.1984.5010248.

Yonghao Wu, Zheng Li, Jie M. Zhang, Mike Papadakis, Mark Harman, and Yong Liu. Large language models in fault localisation, 2023.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models, 2024.

Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu. Dlinfer: Deep learning with static slicing for python type inference. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 2009–2021, 2023. doi: 10.1109/ICSE48619.2023.00170.

Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. White-box compiler fuzzing empowered by large language models, 2023a.

Chenyuan Yang, Zijie Zhao, and Lingming Zhang. Kernelgpt: Enhanced kernel fuzzing via large language models, 2023b.

Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, pp. 39–51, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393799. doi: 10.1145/3533767.3534390. URL https://doi.org/10.1145/3533767.3534390.

```
    [CALLER SNIPPET from JSelect]
Please perform abstract interpretation on the previous piece of code line by line.
After each line you interpret, provide the output from abstract interpretation.
Focus specifically on the call `[CALL]` that is on line [LINE]. Please note that
this is not a complete piece of code, and is a snippet from a larger body.

            [MODEL RESPONSE]

What does the `[CALL]` call on line [LINE] do? Below is some information about the
variables that are used.
Var1, which is defined on line X of the previous snippet.
Var2, which is defined on line Y of the previous snippet.
...
            [MODEL RESPONSE]

Does the `[CALL]` call refer to the function `[TARGET FUNCTION]` on line [LINE] in
the following program snippet? Please use the information provided previously and
answer yes or no.
    [TARGET FUNCTION SNIPPET from JSelect]

            [MODEL RESPONSE]
```

Figure 5: The final prompting template for CALLME.

```
1  var obj = {};
2  var callUserString = true;
3  obj["call_func1"] = function() {
4    console.log("func1");
5  }
6
7  obj["call_func2"] = function() {
8    console.log("func2");
9  }
10
11 function callFunc(userString) {
12   if (userString == "func1" ||
13     userString == "func2") {
14     if (callUserString === true) {
15       var a = "call_" + userString;
16       obj[a]();                    Seed statement
17     }
18   } else {
19     obj[userString] = "foo";
20   }
21 };
22
23 function foo() {
24   callFunc("func1");
25 }
26 foo();
```

Figure 6: An example of a program to demonstrate slicing.

A Appendix

B Statement Selection Details

Simple Windowing: Our baseline approach is to use a simple window around the call site and callee function without any further analysis. This determines whether LLMs can perform call graph reconstruction without being augmented by traditional static analysis tools. In our experiments, we found a window size of 50 to work best.

Full Slicing with ACG: As CALLME is meant to be used alongside existing algorithms such as ACG, we use the output of WALA's ACG implementation to construct our slices. We

Table 7: The output of different slicing methods on Figure 6.

Slicing Option	Slice

Full Slicing	`var obj = {};`
	`obj["call_func1"] = function() {`
	`  console.log("func1");`
	`}`
	`obj["call_func2"] = function() {`
	`  console.log("func2");`
	`}`
	`function callFunc(userString) {`
	`  if (userString == "func1" ||`
	`  userString == "func2") {`
	`  if (callUserString === true) {`
	`  var a = "call_" + userString;`
	`  obj[a]();`
	`  }`
	`  }`
	`}`
	`callFunc("func1");`

Thin Slicing	`var obj = {};`
	`function callFunc(userString) {`
	`  a = "call_" + userString;`
	`  obj[a]();`
	`}`
	`callFunc("func1");`

Def-Use	`var obj = {};`
	`obj["call_func1"] = function() {`
	`  console.log("func1");`
	`}`
	
	`obj["call_func2"] = function() {`
	`  console.log("func2");`
	`}`
	`var a = "call_" + userString;`
	`obj[a]();`
	`obj[userString] = "foo";`

choose ACG rather than Jelly as its implementation inside of WALA allows for querying of the points-to graph generated during call graph construction. ACG generates a simple flow analysis (Feldthaus et al., 2013), which only tracks the flow of function values rather than all objects. This allows it to scale to large frameworks such as *React* and *jQuery*. We use the output from the flow analysis to build a system dependence graph for our slicer. WALA constructs an Intermediate Representation (IR) from the source code. We use the built in source mapping to find all IR statements that correspond with the caller line. Each statement is a seed statement. Full slicing returns the set of all statements that can influence the value of one or more specified seed statements by tracing all data and control dependencies from any variables in the seed statement, and is repeated recursively to find all statements. Table 7 shows the output of our full slicer on the code from Figure 6. Note that ACG only tracks function values rather than all objects, leading to the definition of `callUserString` on line 2 not being included in the slice.

Thin Slicing with ACG: Full program slices often return more information than is necessary. To remedy this, Sridharan *et al.* developed an approach called thin slicing (Sridharan et al., 2007), which helps limit the size of the slices. Thin slicing ignores value flow of base pointers as well as control dependencies. In contrast, if one of the seed statements contains a value read from a global object, a full slice would include *all* writes to that object, whereas a thin slice would ignore all writes to the object. In Table 7, we can see that the thin slice ignores all control dependencies and all other writes to `obj`.

Model	Detect	Miss	FP	Acc.	Prec.	Recall
Jelly	7	177	0	52%	1.0	.04
CodeLlama-7B	108	76	49	66%	.69	.59
CodeLlama-13B	133	51	35	76%	.79	.72
CodeLlama-34B	121	63	21	77%	.85	.66
GPT-4	98	86	3	76%	.97	.53

Table 8: Results to determine CALLME's ability to resolve calls from different files.

Def-Use. We follow the protocol described in Section 3. The Def-Use slice only contains statements that reference the one of the variables at the seed statement. In the code snippet in Figure 6, these are any statements that refer `obj` or `a`. We test both using only the Def-Use information as well as combining it with a naive window, as seen in Table 4.

C Prompting details.

Program understanding with Abstract Interpretation. The approach described in Section 3. We ask the model to perform abstract interpretation line-by-line in the first prompt. We then ask the model to analyze the purpose of the call. Finally, we ask the model whether the call refers to the target function.

Program understanding with English Interpretation. This serves as a baseline to determine how much asking the model to perform abstract interpretation helps. We use the same prompts as the abstract interpretation approach, except we ask the model to create textual descriptions of each line without mentioning abstract interpretation.

Instruct. We guide the model step by step through resolving a dynamic property access call using the same methodology that a person would. The prompts in order are as follows:

1. We ask the model to analyze the variables used in the call, and provide the variable information that is provided by JSelect as well as the statements.
2. We ask the model what the purpose of the call is for and how it is used.
3. We ask the model whether the call refers to the target function and provide the statements for the target function output by JSelect.

Two-step. We first provide the model with the relevant statements for the call site, and ask the model to analyze what is happening at the call site. After the model responds, we then provide the model with the callee function and ask whether the call site refers to the callee function. The purpose of the two-step prompt is to see if the model is able to determine what the important pieces of information are in resolving a dynamic property access.

Direct. In our direct prediction approach, we provide the model with the relevant statements for the call site as well as the callee function, and ask the model to predict whether the call site can refer to the callee function.

Explain. We perform the same test as in the direct prompts, but also ask the model to explain its reasoning before providing an answer.

D Additional Results

E Vulnerability Example

Motivating example. To show how resolving dynamic property accesses can be crucial to vulnerability detection, consider the following code snippet from the NPM package `find-process` in Figure 7, which has over 1.2 million weekly downloads (fin).

Benchmarks	Detect		Miss		FP	
	CALLME	Jelly	CALLME	Jelly	CALLME	Jelly
AngularJS	136	12	71	195	25	11
Backbone	30	2	16	44	4	1
KnockoutJs	24	2	0	22	6	0
KnockbackJs	70	39	15	46	15	4
CanJs	71	6	20	85	18	2
React	45	2	12	55	15	0
Mithril	27	1	0	26	4	0
Vue	48	4	13	57	3	1
VanillaJs	13	0	1	14	2	0
jQuery	32	3	16	45	10	0

Table 9: Results separated by program and compared to Jelly, the state of the art call graph construction algorithm for Javascript.

```
1  const finders = {
2    darwin: function(cond) {
3      ...
4      ...
5      ...
6      exec(cond.cmd);          Command injection!
7    },
8    android: function(cond) {
9      let cmd = 'ps';
10     utils.exec(cmd);
11   },
12   'linux': darwin
13 }
14
15 function findProcess (cond) {
16   let platform = process.platform;
17   let find = finders[platform];
18   if (typeof find === 'string') {
19     find = finders[find];
20   }
21   find(cond).then(resolve, reject);
22 }
23
24 module.exports = findProcess
```

Figure 7: A security vulnerability from findProcess. Because `cond` is an argument to the `darwin` function, whether there is command injection on line 6 depends on whether `cond.pid` was sanitized before the call to `darwin`. Resolving the call on line 21 requires resolving the dynamic property access on line 17.

There is a potential command injection vulnerability in the function `darwin` on line 6. If an attacker can control `cond.pid`, they can execute arbitrary commands as `cond.pid` is passed into `exec`. However, `cond` is a function argument, so whether or not there is a vulnerability depends on whether it was sanitized before the function call. The function call is made on line 21, where if `finders[platform]` is `darwin`, then the `darwin` function is called with unsanitized input. In order to discover the vulnerability, a static analyzer would need to determine whether the dynamic property access on line 17 can refer to the `darwin` function on line 2. As a result, current static call graph algorithms such as JAM (Nielsen et al., 2021a) and ACG (Feldthaus et al., 2013) miss this dynamic property access and the vulnerability is undiscovered.