UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Babbling Microwalk: A Novel JavaScript Plugin for Side-Channel Analysis using Microwalk

Implementierung einer neuen JavaScript Instrumentierung für die Seitenkanalanalyse mit MicroWalk

Bachelorarbeit

verfasst am
Institut für IT-Sicherheit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Anja Köhl

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Jan Wichelmann

Lübeck, den 19. Januar 2024

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____

Anja Köhl

Zusammenfassung

Im Jahr 2022 wurde ein JavaScript Trace Erzeugungs Plugin für MicroWalk veröffentlicht. MicroWalk ist ein Programm, um Code auf mikroarchitekturelle Seitenkanallecks zu überprüfen. Das veröffentlichte Plugin war durch die dynamische Instrumentierungsbibliothek Jalangi2, welche es als Basis verwendete, limitiert. Nach einer Beurteilung der notwendigen Änderungen, um Jalangi2 zu modernisieren, wurde die Entscheidung getroffen ein neues Plugin für JavaScript für MicroWalk als Ziel dieser Arbeit zu implementieren. Statt dynamischer Instrumentierung setzt das neue Plugin auf die direkte Manipulation von abstrakten Syntaxbäumen, um die Erzeugung eines Traces während der Laufzeit des JavaScript Programm zu ermöglichen. Dieser Ansatz ermöglicht es, die Instrumentierung für das Tracen zu minimieren sodass, der erzeugte Overhead reduziert wird.

Abstract

In 2022 a JavaScript trace generation plugin for the MicroWalk tool was introduced. MicroWalk is a framework for analyzing microarchitectural side-channel leakage in code. This plugin was limited by the dynamic instrumentation framework it used as a basis — Jalangi2. Following an assessment of the changes required to modernize Jalangi2, the decision was instead made to implement a novel MicroWalk plugin for JavaScript as the aim of this thesis. Rather than relying on dynamic instrumentation, the new plugin manipulates the AST of a given JavaScript program directly to enable trace generation during execution. This approach allows the instrumentation for trace generation to be minimal, which in turn reduces the overhead induced.

Acknowledgements

I would like to thank my parents, my sister, Sam, Chips, and my friends, Inni and David, for always supporting me and listening to my brainstorming. Additionally, I would like to thank my doctors and all the medical personnel that have accompanied me throughout the years. Without them this thesis would have never been possible.

Contents

1

Introduction

Side-channel attacks have long been a staple in the world of cybersecurity. These attacks are especially dangerous since the attack vector is, by design, outside the scope of the program. Basically, by monitoring information that is affected by the program execution deductions can be made. An analogy aside from technology would be the "Washington Pizza Index" as described in the Washington Post by Schafer [25]. Pizza delivery services noticed a correlation between the volume of pizza orders to the White House and Capitol Hill and political crises. Without knowing *what* was being discussed, delivery companies could still deduce major political discussions were taking place by monitoring the increase in demand. Back to the context of technological side-channels: attackers can leverage information such as the execution time or cache accesses (this would be akin to the increase in pizza deliveries of the previous example) to derive secrets. Since side-channels can be so incredibly varied and must always be considered during development these attacks pose a particular threat.

With rising popularity of web services the malicious interest in such services has also gained traction. The internet connects people all over the world, yet not all of these people have the best of intentions. Attackers utilizing websites to execute side-channel attacks forgo the classically required physical access to the victim system. Clearly this immediately increases the range of an attack by inordinate proportions.

One of the most popular client- and server-side programming languages for web development is JavaScript as illustrated by the somewhat tongue-in-cheek statement by Jeff Atwood "Any application that can be written in JavaScript, will eventually be written in JavaScript." Due to its ubiquitous nature the language is also of particular interest for IT security considerations. The ongoing challenge is to connect these two interests: i.e. writing secure code, for example in the context of web development, by those unfamiliar with IT security concepts in a simple and comprehensive way.

The MicroWalk framework [27] has been developed for this purpose and uses dynamic analysis to highlight potential microarchitectural side-channel leakage in a program. This is accomplished by assessing execution traces generated by executing custom test cases written by the user. The prerequisite for an execution to generate a trace is that the chosen code must be *instrumented*. This is where MicroWalk's plugins come into play: plugins take care of instrumenting code of a certain language in a way that upon execution of this code, a trace is generated, which can then be preprocessed to bring it in line

with the specific format required for further analysis by MicroWalk.

A MicroWalk plugin for instrumenting JavaScript was first introduced by Wichelmann et al. in 2022 — albeit with severe limitations. Due to constraints imposed by the choice of the underlying instrumentation library, the plugin cannot handle modern JavaScript language features. Additionally, the instrumentation substantially impacts execution time which makes it undesirable for developers looking to integrate MicroWalk analysis into their development workflow.

1.1 Contributions of this thesis

Given the constraints of the current MicroWalk plugin the aim of this thesis is to adjust the plugin to the needs of modern JavaScript. More specifically, the task is to evaluate and improve the underlying framework of the existing plugin, Jalangi2 [12].

Following the analysis of Jalangi2, a new plugin has been developed using packages from the Babel library [2]. This plugin is fully compatible with modern JavaScript. Over the course of the development a different approach was developed to modify the abstract syntax tree of a given code using Babel. The benefit of direct manipulation of the source code via the abstract syntax tree has the advantage that it results in minimal changes to the source. Additionally, MicroWalk plugins for any similar programming language can be written using the same novel approach under the condition that a framework for parsing source code into an abstract syntax tree exists.

1.2 Related Work

Specialized tools for assessing whether potential microarchitectural side-channel leakage is present in code exist, but they largely target binary executables or LLVM intermediate representation (IR) [14], which is used as a language independent representation of code by the LLVM compiler toolchain. A study conducted by Jančár et al. in 2022 among developers of cryptographic libraries showed that ctgrind [18] is the most widely used tool for this purpose. The ctgrind tool is an extension of Valgrind, an "instrumentation framework for building dynamic analysis tools" [26].

In contrast, not many frameworks for instrumentation of *JavaScript* exist and no specialized tools for securing against microarchitectural side-channel leakage in JavaScript code exist. Since JavaScript source code is only compiled during execution (a process known as just-in-time compilation), no binary is available for use with dynamic analysis tools. Furthermore, standard JavaScript engines are unable to generate LLVM IR, making tools that target it inaccessible for JavaScript developers.

Jalangi2 [12] is a framework for JavaScript that fully transforms a given source code to allow arbitrary dynamic instrumentation. Unfortunately Jalangi2 is no longer in active development and does not support modern JavaScript language features. A more detailed description of Jalangi2 is provided in Section 2.4.

Aran [4] is another framework for generic dynamic instrumentation. Again the entire source code is transformed to allow arbitrary analysis. Both of these frameworks are

designed as general purpose tools and as such there is a substantial performance loss from transforming the entire given code. Additionally, since the tools are mainly designed to enable dynamic analysis on JavaScript code in general, further work is subsequently required to then assess potential microarchitectural side-channel leakage of the code.

As outlined above in Section 1.1 on the preceding page the MicroWalk plugin developed by Wichelmann et al. fills the gap of missing tools for securing JavaScript code against microarchitectural side-channel leakage. The plugin generates compatible traces for MicroWalk, by instrumenting the given JavaScript source code using Jalangi2 to emit information required for the MicroWalk trace format. This work forms the basis for the work done for this thesis.

1.3 Structure of the thesis

The thesis is divided into five chapters. The second chapter explores the various concepts required to understand the new plugin that was developed for this thesis. Specifically, a basic understanding of microarchitectural side-channel attacks, instrumentation, the Jalangi2 and Babel libraries, abstract syntax trees and the JavaScript language are provided in this chapter. Additionally, the MicroWalk framework is explained in detail, as it forms the basis for the developed plugin.

The next two chapters detail the results and contributions of this thesis to ease development of secure JavaScript cryptographic libraries. First chapter three begins with the approach and analysis of the existing MicroWalk JavaScript plugin, using Jalangi2. Surmising the results of the analysis of the existing plugin, chapter three continues with the evaluation of the babel framework for the new plugin and the advantages of using Babel over Jalangi2.

In chapter four the implementation and the overarching concept of the plugin are illustrated with an in-depth explanation of the abstract syntax tree modifications and the trace format generated by the new plugin. Readers solely interested in the inner workings of the plugin may wish to skip directly to this chapter.

Finally, the last chapter concludes and summarizes the thesis work.

2

Background

Side-channel attacks are a field of major interest in IT security. Their primary target is not an underlying cryptographic protocol of a program but rather the (*side-*) *effects* caused by an executing program. For example, the first side-channel attacks described in research were the timing attacks on cryptographic systems using modular exponentiation [17]. This work showed that by observing the timing of operations, secrets can be extracted. The cryptographic protocols attacked were mathematically sound, yet the information exposed by the side-channel was sufficient to break secrecy.

Among the different side-channels, those that target microarchitecture show a high prevalence and relevance in current research. For example, the Spectre [15] attack introduced an entirely new class of microarchitectural side-channel attacks that continues to be relevant to this day [3]. Therefore, the first section is dedicated to defining microarchitectural side-channel attacks and setting the scope, which is of particular interest for this thesis. Afterwards, the MicroWalk framework is introduced in-depth as a tool for preventing microarchitectural side-channel leakage during development. Then, a short foray into dynamic and static instrumentation is undertaken, highlighting the differences between the two approaches and when each technique is used. Instrumentation is used extensively by MicroWalk in order to gather information about a program during execution. The focus then shifts towards syntactic analysis with an emphasis on abstract syntax trees for JavaScript. As we will see later when the new MicroWalk plugin is explained in greater depth, operating on these trees directly allows for finer control of the changes to the source. Finally, the last section provides a brief introduction to JavaScript — which is used for the MicroWalk plugin — and showcases some intricacies of the language, which are needed for a greater understanding of the plugin and the challenges encountered during development.

2.1 Side-channel Attacks

As the name suggests *side-channel* attacks target information exposed by *secondary* information channels. This can be physical information such as the power consumption [16], electromagnetic radiation [10] or the previously mentioned timing [17] of a device, but it can also be information available on the microarchitectural level [22]. Here an attacker

takes advantage of resources shared with a victim, e.g. caches or a branch prediction unit, to extract information without consent. Monitoring the access time of a cache request allows an attacker to determine, whether a victim process accessed the same resource previously or not.

The foundation for this type of attack was laid by work of Osvik, Shamir, and Tromer [22] introducing the Prime+Probe technique with a later variation being the Flush+Reload technique [29]. A Prime+Probe attack can determine whether a cache set was used by a victim during a certain timeframe. First, a contiguous byte array is allocated filling the entire cache with the attacker's data. After a short delay, e.g. to wait for an encryption, the array is accessed again and the access time is measured. If a cache *miss* occurs, then the access time will consequently be higher and an attacker can deduce that the corresponding cache set was accessed by the victim, since the cache was previously filled with the attacker's data.

A Flush+Reload attack has an even higher spatial resolution of a singular cache line. In contrast to Prime+Probe, the monitored cache line is *flushed* from the cache. Again, after a short delay the cache line is accessed, and the access time is measured. If the victim accessed the cache line since it was flushed, then a cache *hit* will occur and the access time will be much faster than would be the case for a cache miss.

Alternatively an attacker may also track the control-flow of a program, which can result in information leaks in case of secret-dependent execution. In modern cloud-computing environments this vector is particularly perilous, as demonstrated by cross-VM prime+probe attacks by Liu et al. [19]. This work shows that malicious virtual machines (VMs) can leverage shared last-level caches to extract data from co-located VMs.

As mentioned briefly in the introduction, microarchitectural side-channel attacks using JavaScript are feasible and forgo the requirement of physical access or co-location of hardware. Oren et al. [21] detailed attacks using Prime+Probe that run entirely in the browser of the victim and simply require the victim to visit a malicious website. More recently, Ridder et al. [24] introduced a microarchitectural side-channel attack in JavaScript, which can compromise the Firefox browser. These attacks show that JavaScript sandboxing is not sufficient in preventing cache attacks.

Writing constant-time code is the generally accepted technique to avoid such side-channel leaks as each execution of the program results in the same memory accesses, identical control flow and identical timing behavior. To verify constant-time properties of code, its behavior must be compared in response to different inputs. A true constant-time code must have no discrepancies in memory accesses and a linear control flow regardless of input. This is in contrast to secret-dependent execution, where execution behavior changes *depending* on secrets. As an example of secret-dependent code, consider the simple if-statement detailed in Listing 2.1 on the next page. By monitoring the control flow during execution, the result of the test can be inferred. Since the result of the test is synonymous to one bit of information of the secret, the secret itself can be inferred.

To make Listing 2.1 on the following page secret-*independent*, essentially both cases are fused into a single expression which contains both results, but only stores the applicable one. This is shown in Listing 2.2 on the next page.

Listing 2.1: A simple secret-dependent if-statement in JavaScript with *secret* being either 1 or 0

```
if(secret == 1) {
    x = x * 10;
} else { // secret == 0
    x = x + 10;
}
// ...
```

Listing 2.2: A secret-independent transformation of Listing 2.1 in JavaScript with *secret* being either 1 or 0

```
x = (x * 10) * (secret) + (x + 10) * (1-secret);
```

When *secret* is 1, then the expression evaluates to the following:

$$x = (x \cdot 10) \cdot secret + (x + 10) \cdot (1 - secret)$$
$$x = (x \cdot 10) \cdot 1 + (x + 10) \cdot (1 - 1)$$
$$x = (x \cdot 10) + (x + 10) \cdot 0$$
$$x = x \cdot 10$$

In contrast, when *secret* is 0, then the expression evaluates as follows:

$$x = (x \cdot 10) \cdot (secret) + (x + 10) \cdot (1 - secret)$$
$$x = (x \cdot 10) \cdot 0 + (x + 10) \cdot (1 - 0)$$
$$x = (x + 10) \cdot 1$$
$$x = x + 10$$

Clearly, this results in x having the same value as when the two cases are split into an if-else-statement, yet with the code in Listing 2.2, there is no branching and both mathematical operations are always executed.

2.2 MicroWalk

Finding potential microarchitectural side-channel leakage in code is not a trivial task. Consequently, many tools have been developed to help developers find such leakage in their code. One of these tools, MicroWalk [27], is a framework for analyzing and finally classifying microarchitectural side-channel leakage in a given code.

Essentially, execution traces resulting from executing the program with various secret inputs are compared. If the traces differ, then these differences are a direct result from executing the program with different secret inputs. Consequently, the program has microarchitectural side-channel leakage.

The entire procedure can be seen as a pipeline consisting of discrete stages for *test case generation*, *creation of execution traces* and finally the *analysis* itself as visualized in Figure 2.3. Each stage is extensible by plugins — a quality that is leveraged for the work of this thesis. While the initial publication of 2018 focussed on binary analysis, later work by Wichelmann et al. in 2022 showcased MicroWalk's extensibility by introducing JavaScript analysis [28].

Figure 2.3: MicroWalk pipeline for analyzing and classifying microarchitectural side-channel leakage.

MicroWalk can generate test cases of variable length using cryptographically secure pseudorandom number generators. Alternatively, previously generated inputs can be specified. These test cases are used to observe, how the given program behaves with various inputs.

To capture the program state during execution, execution traces must be generated. These traces detail memory accesses and the control flow. Since MicroWalk originally focussed on binaries, memory allocation and stack pointer modifications are also of importance. However, given that JavaScript abstracts these details, they are not further considered in the context of this thesis. The main task of the thesis was to create a plugin, which extends the trace generation stage by instrumenting JavaScript code such that trace information can be emitted. To guarantee a uniform trace format, the raw execution traces must subsequently be preprocessed.

The uniform trace format is a prerequisite for the leakage analysis algorithm used by MicroWalk. Traces are merged into a *call tree* to keep track of trace divergences. Branches from nodes symbolize divergences in traces. The nodes of the tree are one of three types: *branches* i.e. function calls, returns from calls or jumps, *memory allocations* and *memory accesses* i.e. reading or writing an object. Each node contains the common trace entries for all traces that visit this node and a list of *split nodes*, which detail the divergences starting from this node. Since JavaScript lacks fine-grained control of memory allocation, such as `malloc` calls in C, which explicitly allocate memory, memory allocation nodes are not considered further.

After analyzing the microarchitectural leakage its significance is determined by the final analysis itself. Leakage can be quantified by MicroWalk using three different measures. These are: mutual information, conditional guessing entropy and minimal conditional guessing entropy.

Mutual information can be utilized to quantify the dependence of two variables upon each other. Basically, this measurement quantifies, how much information can be extracted regarding a pair of variables by observing only one of the two. In the context of microarchitectural side-channel leakage, a low mutual information score is desirable, because information gained by observing a different variable is indicative of side-channel leakage. A low score minimizes the risk that the observation of execution (for example by observing access patterns) will leak undesired information.

Conditional guessing entropy of two variables quantifies, how many guesses are expected in order to determine the value of one, when knowing the value of the other variable. Consequently, the *minimal* conditional guessing entropy is the lower bound of this number of expected guesses.

2.3 Dynamic and Static Instrumentation

The term "instrumenting" existing code, classically an existing binary, refers to an approach that entails inserting new code, which alters the behavior of a program. Such modifications either enable observation, e.g. by emitting information regarding the execution, or introduce new behavior, as is the case with modern compiler's stack protection mechanisms. An example of instrumentation as a stack protection mechanism is gcc's `fstack-protector` flag, which adds guard code to check for buffer overflows.

Dynamic instrumentation hooks into a program at runtime and inserts new instructions into the instruction stream directly. At this point, a plethora of information is available, such as memory used by the executing code and the system state that the execution induces. An example for a dynamic instrumentation tool for x86 binaries is Pin [23], which allows users to write program analyses specific to certain patterns. The advantage of dynamic instrumentation is the wealth of information available. The flip side is the potential negative impact that instrumentation can have on the performance of the program by injecting new instructions during runtime.

In contrast, static instrumentation operates solely on the *static* source code, without execution. This can be the actual source code before compilation, the decompiled code or decompiled machine instructions. The instrumentation is added directly to the source code, making it a persistent component of the program prior to compilation. Thus, static instrumentation is generally more complex, since it requires greater precautions to avoid breaking existing program logic or altering existing references. However, the advantage of static instrumentation is that the entire code can be taken into account, especially when the original source code is available. Additionally, no changes are made during runtime.

Since JavaScript is a just-in-time (JIT) compiled language, no binary is available for instrumentation but in contrast the source code is readily available, albeit potentially in minified form. This means that static instrumentation is more feasible since no decompilation is required.

2.4 Jalangi2

In its own words, Jalangi2 is "a framework for writing dynamic analyses for JavaScript" [12]. As a generic framework it is not explicitly specialized in instrumenting JavaScript to verify constant-time properties, but instead to write arbitrary dynamic "analyses". Code instrumented by Jalangi2 is first statically instrumented when it is loaded, replacing the entire source with an instrumented version of the code, which has been transformed to wrap each action in a corresponding wrapper function. For example, an assignment expression `x=1` would be wrapped as `Write("x",Literal(1))`. These wrappers are then used

by the callbacks that the Jalangi2 runtime exposes to allow users to write their analyses. Basically, before and after each action, the corresponding callback is first called, executing provided user code if applicable, and then the action itself is performed. Jalangi2 performs this wrapper transformation for *everything* regardless of whether a callback that uses a certain wrapper is utilized by the user analysis or not.

Additionally, all expressions are assigned IDs, which are used in conjunction with a source map to look up the corresponding expression in the original source code. This means that the "`x`" in the previous example would instead be encoded as an id that corresponds to the exact location of the "`x`" in the original source. The instrumented version exclusively uses these internal IDs to reference expressions.

Listing 2.4: A minimal assignment expression in JavaScript.

```
1 var x;
2 x = 1;
```

Table 2.5: Jalangi2 internal IDs of Listing 2.6 matched to source objects in Listing 2.4. Source locations have the format [`start line, start column, end line, end column`].

Id	Source Location	Source object
9, 17	[2, 5, 2, 6]	`x`
25	[2, 1, 2, 7]	`x = 1;`
33, 41, 49, 57	[1, 1, 2, 7]	`var x;x = 1;`

Listing 2.4 first shows the minimal JavaScript code for a valid variable assignment. Then, Listing 2.6 on the next page shows the instrumented version of the same assignment. The IDs defined in lines 2-4 correspond to expressions in the source as detailed previously and illustrated in Table 2.5. All function calls that are called as members of the `J$` Object are wrappers. For example `J$.W()` in line 15 of Listing 2.6 on the next page wraps a write operation, while `J$.N()` in the same line wraps the literal "`1`".

An analysis in the context of Jalangi2 is a JavaScript file that implements some or all of the available callbacks. As previously mentioned, these callbacks are then called before (or in some cases after) wrapped code is executed. The aforementioned example of `J$.W()` would execute the `write()` callback before executing a write operation. Thus, users can utilize the callbacks to further instrument code. Both, the repeated instrumentation at loadtime and the extensive transformation result in a slowdown of 3-100 times the original, according to Jalangi2 developers.

Regarding ongoing support, Jalangi2 supports Node.js version 12 and additionally uses Python 2.7. Node.js v12 reached end-of-life status in April 2022 and Python v2.7 has been marked end-of-life since January 2020. Jalangi2 officially supports ECMAScript version 5.1 which was released 2011 [5] and superseded by version 6 released 2015 [6]. The last version update of Jalangi2 was on July 2, 2020. Thus, the tool is no longer being actively developed.

Listing 2.6: The minimal assignment expression from Listing 2.4 as instrumented by Jalangi2.

```
J$.iIDs = {
    "9":[2,5,2,6],"17":[2,5,2,6],
    "25":[2,1,2,7],
    "33":[1,1,2,7],"41":[1,1,2,7],"49":[1,1,2,7],"57":[1,1,2,7],
    "nBranches":0,
    "originalCodeFileName":"id.js",
    "instrumentedCodeFileName":"id_jalangi_.js"
};
jalangiLabel0:
    while (true) {
        try {
            J$.Se(33, 'id_jalangi_.js', 'id.js');
            J$.N(41, 'x', x, 0);
            var x;
            J$.X1(25, x = J$.W(17, 'x', J$.T(9, 1, 22, false), x, 2));
        } catch (J$e) {
            J$.Ex(49, J$e);
        } finally {
            if (J$.Sr(57)) {
                J$.L();
                continue jalangiLabel0;
            } else {
                J$.L();
                break jalangiLabel0;
            }
        }
    }
```

2.5 Babel

Given these constraints of Jalangi2 as a foundation for the MicroWalk plugin, different supporting tools for the development of the new MicroWalk plugin were considered. As outlined above, Babel [2] is well-suited for this purpose. Babel is not primarily a framework intended for custom instrumentation of JavaScript code. Rather, it is a "transpiler" or "source-to-source" compiler capable of transforming arbitrary JavaScript code into compatible legacy code according to a previous ECMAScript Language specification. For example code written using modern language features, such as the nullish coalescing operator (`??`) introduced in ES2020, can be *transpiled* by Babel into ES5 code as demonstrated by the listings in Figure 2.7.

```
// in
function greet(input) {
    return input ?? "Hello world";
}
```

```
// out
function greet(input) {
    return input != null ? input : "Hello world";
}
```

Figure 2.7: Code snippets from the Babel GitHub [2] showcasing a transformation of modern (ES2020) JavaScript using the nullish coalescing operator (`??`) to code compatible with prior (<ES2020) specifications.

For the transformation, the source code is parsed into an abstract syntax tree (AST) and then modifications are made to the nodes of the tree. Babel consists of multiple packages that are used in conjunction for the source-to-source compilation. The *core* packages are described in Table 2.8.

Table 2.8: The packages that make up the core of the Babel library.

Package Name	Description
`@babel/core`	Babel source-to-source compiler itself that transforms code compliant with one specification to another
`@babel/parser`	parser that parses source code and creates a corresponding abstract syntax tree object
`@babel/traverse`	library for traversing and manipulating the AST object
`@babel/generator`	library for generating source code from an AST object

For the new MicroWalk plugin, the parser, traverse and generator packages are used *without* the core package. The parser library parses a given JavaScript or TypeScript source code and creates a corresponding AST object that can be manipulated using further babel libraries. The tree object represents the complete syntactic structure of the given program

by breaking each statement, expression or other syntactic structure down into terminals. Abstract syntax trees are explained in more detail in Section 2.6.

The traverse library in particular focuses on utilities for traversing the tree and manipulating nodes, for example by replacing, removing or modifying them. The main entry point for tree traversal is the `traverse(ast,visitors)` method that takes an AST object and visitors for AST nodes. Visitors are (function) objects that define code that should be executed when a node of the corresponding type is encountered. This means that for each node of the tree, the node type is matched against the types accepted by each visitor passed to the `traverse` call and, in case of a match, the node is visited, causing the visitor code to be executed. The pattern matching done for visitors allows the new MicroWalk plugin to perform very specific transformations to the AST.

The final *core* Babel package that is used, is the generator package. This package is responsible for generating JavaScript source code from a given AST object.

2.6 Abstract Syntax Trees

As mentioned in the previous section, abstract syntax trees are used to represent the syntactic structure of a program. They symbolize the derivation of statements and expressions down to terminal words according to a JavaScript AST specification. The derivations are represented by nodes of the tree with each node encompassing all of its children. For example, the root node represents the *entire* program, since every node is a child of the root node. Terminals are words that can no longer be broken down further. Consequently, in the tree they are leaf nodes. In the context of this thesis, the Babel AST specification is used to define trees, as the MicroWalk plugin developed for this thesis uses the Babel library to parse and instrument JavaScript source files.

According to the Babel specification, the root node for an AST is a program node, which contains all the statements of the program as its children. Statements can in turn consist of expressions or further statements. Figure 2.9 on the next page shows an example of how source code is represented by a Babel AST.

2.7 JavaScript

In terms of computer science, JavaScript is an ancient programming language. First described in 1995, it has gone through many shifts of direction and iterations, especially since its transition from a solely client-side scripting language, to a client and server-side language with the advent of *Node.js*. Standardization began in 1997, when it was submitted to Ecma International which has overseen the specification of ECMAScript (ES) since [7]. JavaScript is formally a language that conforms to the ECMAScript standard. Both terms (JavaScript and ECMAScript) are used interchangeably in this thesis.

When the language was first specified in 1995, it was intended to only be used for scripting within web browsers. The introduction of Node.js, a runtime environment for JavaScript outside a browser in 2009, expanded the reach of JavaScript substantially. The language could now be used for server-side programming, since it was no longer bound to

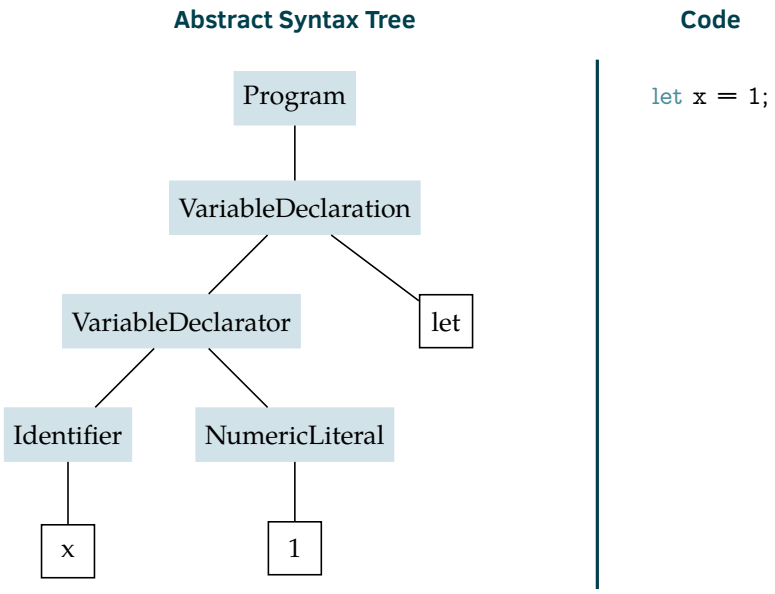

Figure 2.9: Abstract syntax tree and code of a variable declaration. Filled blocks are nodes of the tree and outlined blocks are properties of the node object that are included for clarification.

execution within a browser. Consequently, the capabilities of the language also increased. The biggest change in the specification was with publication of ECMAScript 6 (ES6) [6], seeing additions such as class declarations, a module system, iterators, different variable declarators (`let`, `const`) and arrow function expressions.

Like any programming language, JavaScript has its quirks and novelties. While humorous misunderstandings such as the ones often caused by the complexities of JavaScript type coercion are of little relevance for the new MicroWalk plugin, certain language features are referenced extensively during the explanation of the inner workings of the plugin in Chapter 4 on page 19. Such features are detailed in the following paragraphs.

Object Property Access

In JavaScript almost everything is an object, including functions. Objects have *properties* which can be accessed in one of two ways: either using the dot notation (i.e. `o.property`) or with the bracket notation (i.e. `object[expression]`). The bracket notation is special in that a property can be accessed by passing any *expression* that evaluates to a valid property name of the object. This means that to ascertain which property is being accessed, the expression must first be evaluated. For both notations either a *member expression* [11] or a *call expression* that returns an object form the basis. Since a member expression is either an object, reference to an object or a member expression with a property access, these expressions can also be chained when another object is returned by prior access. Listing 2.10 on the next page shows an example for such a chain.

Listing 2.10: An example of a member expression chain.

```
// 'o' is an object
const o = {
    // 'a' is a property of 'o' and an object
    a: {
        // 'x' is a property of 'a'
        x: 5
    }
};

o.a.x; // 5
```

Destructuring Assignment

Object properties can also be accessed indirectly when assigning variables with a process called *destructuring assignment*. This means that values from objects and arrays can be unpacked and assigned to variables without the need for intermediate variables. Listing 2.11 show examples of destructuring assignment.

Listing 2.11: An example of destructuring assignment in JavaScript.

```
// 'o' is an object
const o = {
    a: 1,
    b: 2
};

let { a, b: c } = o;
// a = 1, c = 2

// 'arr' is an array
const arr = [1, 2, 3];
let [x, y] = arr;
// x = 1, y = 2;
// 3rd element is ignored
```

The properties of an object can be assigned to variables that share a name with the property or the property key can be specified, and its value assigned to a variable with a different name. For arrays, the elements are unpacked in ascending order, from the first to last element. Not all values have to be used during destructuring and superfluous values can be stored as an array in a *rest element*.

JavaScript Modules

Several custom solutions for JavaScript have been established that offer desired functionality as the official language standard could not always keep pace with the demand of growing applications. The module system is one of them. Since the 6th edition of ECMAScript (ES6) importing of modules has been standardized, by using either static import declarations or dynamic `import()` calls. Prior to ES6, CommonJS modules were the de facto standard, since they were supported by the package management system of Node.js: *npm*. To this day CommonJS and ES6 modules are published side-by-side on the npm registry, reflecting the fact that the transition from one system to the other has not been realized by all developers.

Given that modern JavaScript code continues to run with CommonJS modules, *both* types of module imports have to be supported when instrumenting JavaScript code. When using Node.js there is no native support for easily changing *both* types of module imports.

To ensure that the instrumented version of a module is executed, so-called *hooks* are utilized for the module loading process. In the case of ES6 modules this is done by registering a hook for the corresponding load function. For CommonJS modules the `require()` call, which is used to load external modules, must be hooked. However, Node.js does not natively support hooking this call hence, to hook into "require", the *pirates* library is used.

3

Preliminary Assessment: A Tale of Two Frameworks

The goal of the thesis was to improve the existing JavaScript MicroWalk plugin [28] as a prerequisite for modern JavaScript support. This chapter details the state of Jalangi2 [12], the underlying framework that is used by the preexisting plugin, the changes required to adapt it to modern JavaScript and the considerable challenges associated with such a major refactoring. In the next step, the usage of the Babel library [2] and its advantages in comparison to Jalangi2 are detailed.

3.1 Jalangi2

The 2022 JavaScript MicroWalk plugin uses *Jalangi2* as a dynamic instrumentation framework to generate execution traces for further analysis by MicroWalk. *Jalangi2* enables dynamic instrumentation by wrapping every expression in calls specific to the type of expression or statement. These wrappers later call the functions that are exposed by the "analysis" callback template, letting users execute custom code for specific hooks. To allow arbitrary instrumentation, the entire given source code is transformed. First, the source is parsed using the *acorn* library [1], which generates a corresponding abstract syntax tree (AST) according to the ESTree Spec [9]. The AST then undergoes a series of transformations.

AST traversal and modification are part of *Jalangi2* itself. Similarly to tree traversal in the *babel/traverse* library, the visitor design pattern is used to "visit" each node of the AST. Starting from the root node, each child is visited in the order of the properties of the node object as given by a `for...in` iteration of the node properties. Upon entering and upon exiting a node, visitors are executed to transform the node. Only node types up to the ESTree Spec for ES5 are considered by *Jalangi2* for traversing and manipulating the AST, since *Jalangi2* only supports ES5. Consequently, to modernize the framework, the visitors must be adapted and expanded to consider all new language features and changes to the ESTree Spec since ES5.

After the transformations of the AST are completed, JavaScript code is generated from the new AST using the *esotope* library [20]. *Esotope* is a fork of the *Escodegen* [8] library

for generating JavaScript from ESTree ASTs. The latest release of *esotope* was in December 2014. Thus, it was last updated prior to the publication of the ES6 standard, which was released June 2015 and has had no further support for almost ten years. Importantly, the base library *Escodegen* was last updated in December 2020, so that it is no longer in active development and will continue to lack features of newer ECMAScript specifications. As a consequence, to modernize the code generation, a different library that supports the current ECMAScript standard is evidently necessary. Furthermore, it needs to be in active development to ensure continuous support.

The first step in adding support for modern JavaScript to *Jalangi2* is expanding the supported node types for AST traversal to also consider new nodes added by ES6. The AST generated by the *acorn* library already contains these new nodes, they simply aren't being considered by Jalangi2 during instrumentation. Additionally, as detailed above, it is necessary to use a new library for code generation from ASTs: Babel. Babel is a project which encompasses libraries for parsing JavaScript into an AST, traversing and modifying ASTs and once again generating code from these ASTs.

Since Babel is an active project with ongoing support, its libraries for AST traversal and modification work with *modern* JavaScript. Hence, the AST traversal could be handled entirely be the *babel/traverse* library. Yet, while *babel/parse* is capable of parsing JavaScript source code into ESTree Spec compliant ASTs (the AST spec *Jalangi2* supports), the *babel/traverse* and *babel/generate* libraries require ASTs according to the Babel AST spec. Consequently, the entire code pertaining to ASTs, such as the visitors that perform the instrumentation of the source, has to be refactored to consider the Babel AST spec.

Given the constraints and challenges outlined above, it becomes clear that the modernization of *Jalangi2* is not a trivial task. The entire instrumentation code, i.e. 2558 lines of code spread over three files, has to be refactored first to use the Babel packages. Subsequently, new instrumentation code would be required to support all new language features since ES5.

An alternative to a complete refactoring of *Jalangi2* is using *Babel* to directly instrument given source code for trace generation. By cutting out the intermediary dynamic instrumentation framework — *Jalangi2* — it is possible to reduce the modifications to a minimum.

3.2 Babel

The Babel project originated from the need to transpile ES6 to ES5 JavaScript code in 2015. Since then, it has grown substantially with 1070 contributors listed on the project GitHub page [2]. Constant activity and an active community have resulted in regular and ongoing releases. The latest update is Babel 7.23.6 released on December 11, 2023. Babel is also supported by companies such as *Airbnb*, *AMP Project* and *Salesforce*. In summary, Babel is a well-rounded project with ongoing support, which makes it an ideal candidate for the foundation of the new MicroWalk plugin.

As briefly mentioned in the previous chapter, the *babel/traverse* library is at the heart of the new MicroWalk plugin. It enables traversal and manipulation of an AST object

created by parsing source code with the *babel/parser* library. Tree traversal means that starting from the root node of the program, each node is first *entered* and afterwards each child node is entered before being *exited* when moving to a sibling node or returning to the parent. In short, the tree is traversed using a depth-first search. An example AST and corresponding tree traversal are illustrated in Figure 3.1.

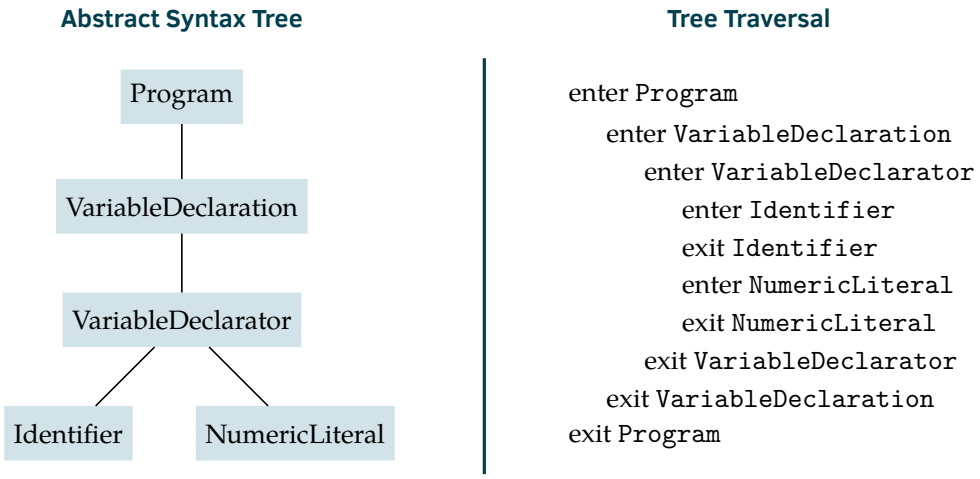

Figure 3.1: Abstract syntax tree and visualized tree traversal of the tree of a variable declaration. Blue blocks are nodes of the tree.

Whenever a node is entered or exited, *visitors* for that specific node type and action (either enter or exit) are called. A *visitor* object can contain methods, which execute upon entering and exiting nodes of a specific type. These methods are passed a path object, which encompasses the path to the current node from the root (program) node. The parent of the current node, which the path is associated with, can be accessed either via the `path.parent` key (for the parent *node*) or the `path.parentPath` key (for the parent *path*). In summary, akin to a linked list, each path holds a reference to its parent path, allowing backtracking all the way to the program node. Thus, the AST can be inspected or modified using the path object. The *babel/traverse* library also entails multiple methods on paths explicitly for this purpose such as `path.insertBefore()`, `path.insertAfter()`, `path.replaceWith()` and `path.remove()`, which provide safe ways of manipulating the AST.

In contrast, *Jalangi2* uses its own AST traversal utilities, which would need to be considered, whenever AST traversal is modified. As a result, there is a higher risk of breaking functionality and a higher complexity in total.

By using Babel, the new MicroWalk plugin utilizes static instrumentation techniques instead of dynamic instrumentation as was the case when using Jalangi2. Without Jalangi2 as an intermediary between the original source and the instrumentation for trace generation, the entire initial code can be taken into account. The major advantage is taht the new MicroWalk plugin focuses solely on instrumenting code for MicroWalk trace generation with pinpoint precision, unlike Jalangi2 which enables generic universal further instrumentation. In particular, only information pertaining to possible side channel leakage is necessary.

4

Implementation: Babbling Microwalk

Due to the multiple constraints imposed by Jalangi2, the decision was made to move away from a dynamic instrumentation framework and instead favor static instrumentation. In particular, an approach was developed that targets the abstract syntax tree (AST) representation of a JavaScript source code. The advantage of this approach is that AST representations exist for most programming languages, since compilers usually parse source code into an AST during compilation, whereas suitable dynamic instrumentation frameworks don't necessarily exist for all languages. Thus, the same steps and concepts of the AST modifications, which allow tracing of potential microarchitectural side-channel leakage in JavaScript, can be used for other programming languages to easily develop MicroWalk plugins.

While the rest of this chapter will focus entirely on JavaScript specific AST modifications, similar syntactic structures can be found in other programming languages. Therefore, similar methods could be used to develop tracing plugins for such languages.

For trace generation, the original JavaScript source code is parsed using the *babel/-parser* library, which generates an AST object representing the program. Next, the AST object undergoes a series of transformations with the help of the *babel/traverse* library. As previously explained in Section 2.5 on page 11, the traversal library traverses the tree in depth-first order, executing visitor code whenever a matching node is encountered. In total, the entire program AST is traversed three times. Twice during the preliminary setup and once to add the trace generation modifications. Eventually, the modified AST object is finally used to generate JavaScript code again using the *babel/generate* library.

The new plugin consists of 2120 lines of code in JavaScript and 46 snippets (usually only consisting of a handful of lines) for testing.

4.1 Preliminary Transformations

The raw original source code given to the plugin is not necessarily suited for all transformations required to trace the necessary information. Thus, certain "setup" modifications are needed to enable tracing. Oftentimes, tracing requires new statements to be injected before or after expressions. Since statements cannot be added *into* expressions, certain syntactic structures must therefore be modified to ensure tracing is feasible.

In the following, each section details the modifications necessary, to enable tracing of a certain statement or expression and why the transformation is required.

Member Expression (Optional) Chains

As detailed in Section 2.7 on page 13, property accessors in JavaScript can be chained into member expression chains. Since trace information is injected around each property access, these chains must be split during preparation. Listing 4.1 shows a "regular" chain of property accessors and Listing 4.2 shows, how the same chain is split during the preparation stage.

Listing 4.1: Regular chain of member expressions in JavaScript.

```
const obj = {
    innerObj1: {
        innerObj2: {
            x: 1
}}};
obj.innerObj1.innerObj2.x;
```

Listing 4.2: Split chain of member expressions of Listing 4.1 after preparation.

```
1 {
2     let $$tmp0 = obj;
3     let $$tmp1 = $$tmp0.innerObj1;
4     let $$tmp2 = $$tmp1.innerObj2;
5     let $$tmp3 = $$tmp2.x;
6     $$chain = $$tmp3;
7 }
8 $$chain;
```

In a first step, each object is assigned its own temporary variable. Then, the result of the last property access is assigned to the `chain` variable which replaces the member expression chain. The temporary variables used to split the chain are scoped to a new block to avoid conflicts with any preexisting variable names.

Call Expressions

Just like member expressions, calls can also be chained as long as the preceding call returned a function object. Similarly to member expression chains, call chains must also be split, since each call is traced individually. Again, a new block is inserted immediately prior to the call, which prepares the arguments of the call. This is required, since each argument may necessitate individual tracing, which isn't possible if it is already in a call expression. Anonymous functions such as the one returned in the example in Listing 4.3 on page 22 are called directly since the `this` value is undefined.

Due to the problem that call arguments could require individual tracing, every call is prepared in such a way that all argument values are first individually pushed into a temporary argument queue, and then removed for the call according to the first-in-first-out principle. If the function object is the result of a property access, then the call is executed via the `call` method of the function object (see line 17 and line 22 in Listing 4.4 on the next page). This allows the explicit assignment of the `this` keyword to match the original object used to call the function.

Switch Statements

Switch-case statements are transformed into equivalent `if` chains to ensure that branches can be properly traced. An example of such a transformation is illustrated in Listing 4.6 on page 23, which is a transformation of the `switch` statement in Listing 4.5 on page 23. Each case creates a new `if` statement with the same condition as the case. Empty cases, i.e. when multiple cases share a consequent block, are merged into one condition by using boolean ORs. The default case is mapped to an `if` statement whose condition is true, if all other conditions are false. A peculiarity of `switch` statements, is that once a case has been matched, all following cases regardless of the condition are executed until a `break` statement is encountered. Hence, a "fall through" variable is added as an override to all conditions to mimic this behavior. Finally, to emulate the `break` statement behavior, the `if` statement chain is placed in a labeled block, which can be broken out of with a `break` statement.

For Statements

All types of for statements undergo changes, since the loop variable must be traced separately. Specifically, the initialization of the variable must be traced once at the start of the loop and the update of the variable must be traced at the beginning of every loop iteration. To enable tracing, the initialization is moved immediately prior to the `for` statement.

Regular `for` statements consist of three components: initialization, condition and afterthought. Each of these three components is an *expression* [7]. Thus, regular `for` statements are transformed into equivalent `while` loops with the initialization prepended and the afterthought added to the loop body as the last statement and before any `continue` statements.

Since `const` variable declarations must be initialized *upon declaration*, these are changed to `let` declarations, which allow the initialization after declaration. Only `for…of` and `for…in` statements that declare a new variable, either explicitly or implicitly, have their initialization moved outside the loop as detailed above.

Listing 4.7 on page 24 and Listing 4.8 on page 24 show the transformation of `for` (lines 1-7 in 4.8), `for…of` and `for…in` (lines 8-13 in 4.8) statements. Since the transformation of `for…of` statements is identical to that of `for…in` statements, only `for…of` is shown for brevity.

Listing 4.3: Regular chains of call expressions in JavaScript.

```
const f = (x) => {
    console.log(x);
    return (y) => {
        console.log(y);
        return;
}};
const o = {
    a: f
};
o.a(1)(2);

const arr = [["inner"]]
arr.pop().pop();
```

Listing 4.4: Split chains of call expressions of Listing 4.3 after preparation showcasing both anonymous calls and calls resulting from a property access.

```
1  //o.a(1)(2)
2  {
3      {
4          $$args.push(1);
5      }
6      $$call = o.a($$this, $$args.shift());
7      {
8          $$args.push(2);
9      }
10     $$call = $$call($$args.shift());
11 }
12 $$call;
13
14 // arr.pop().pop()
15 {
16     let $$tmp0 = o;
17     let $$tmp1 = $$tmp0.pop;
18     let $$tmp2 = $$tmp1.call($$tmp0);
19     let $$tmp3 = $$tmp2.pop;
20     $$chain = $$tmp3;
21     $$this = $$tmp2;
22 }
23 $$chain.call($$this);
```

Listing 4.5: `switch` statement with a `break` statement, fall through and a default case.

```
switch(n) {
    case 0:
        console.log(n);
        break;
    case 1:
    case 2:
        console.log(`combined ${n}`);
    default:
        console.log(`default ${n}`);
}
```

Listing 4.6: Resulting `if` statement chain from transforming the `switch` statement in Listing 4.5. Only the transformation from `switch` to `if` is shown for simplicity.

```
$$switchlabel0: {
    $$switchfallthrough = false;
    // case 0:
    if (n === 0 || $$switchfallthrough) {
        $$switchfallthrough = true;
        console.log(n);
        break $$switchlabel0;
    }
    // case 1:
    // case 2:
    if (n === 1 || n === 2 || $$switchfallthrough) {
        $$switchfallthrough = true;
        console.log(`combined ${n}`);
    }
    // default:
    if (!(n === 0 || n === 1 || n === 2) || $$switchfallthrough) {
        console.log(`default ${n}`);
    }
}
```

Listing 4.7: Example for and for…of statements in JavaScript.

```
for (let i=0; i< 10; i++) {
    console.log(i);
}
for (let e of [1,2,3]) {
    e;
}
```

Listing 4.8: Resulting while and for statements from transforming statements in Listing 4.7. Only the for modifications are shown for brevity.

```
{
  let i = 0;
  while (i < 10) {
    console.log(i);
    i++;
  }
}
{
  let e;
  for (e of [1, 2, 3]) {
    e;
  }
}
```

Do While Statements

As we will later see when discussing the transformations performed for trace generation, branch tracing of conditional statements is injected as the first statement of the body of each conditional branch. Since `do…while` statements execute once prior to evaluation of the test, this would result in an erroneous branch emission. The first iteration of the `do…while` body is *always* executed. To solve this problem, `do…while` statements are transformed into `while` statements, with the loop body prepended once. Thus, the body is executed once without evaluation of the test and a branch trace message is only output, when the condition is evaluated and a branch takes place. Listing 4.9 shows a simple `do…while` loop, while Listing 4.10 shows the corresponding transformation into a `while` loop with prepended body.

Listing 4.9: Simple `do…while` statement in JavaScript.

```
let x = 0;
do {
    x++;
} while (x < 1)
```

Listing 4.10: Resulting `while` statement from transforming the `do…while` statement in Listing 4.9. Only the transformation to a `while` loop is shown for brevity.

```
let x = 0;
{
    x++;
    while (x < 1) {
        x++;
    }
}
```

Ternary Expressions

The ternary operator in JavaScript is essentially a condensed `if` statement. It takes a condition and returns one of two values, depending on whether the condition evaluates to true or false. These ternary or conditional expressions are also a form of conditional branching. Thus, to enable tracing, such expressions are transformed into `if` statements.

It is necessary to transform ternary *expressions* into `if` statements, since the injected tracing code can consist of multiple *statements* and statements can't be children of expressions. An example of this transformation is shown in Listing 4.12 on the following page. Again, if a ternary expression is used to declare a `const` variable, it is changed into a `let` variable, since `const` variable declarations require initialization upon declaration.

Listing 4.11: Simple ternary expression used in an assignment in JavaScript.

```
const x = true ? 0 : 1;
```

Listing 4.12: Resulting `if` statement from transforming the ternary expression in Listing 4.11. Only the transformation to an `if` statement is shown for brevity.

```
let x;
if (true) {
  x = 0;
} else {
  x = 1;
}
```

Computed Properties

When accessing properties of objects using bracket notation, as explained in Section 2.7 on page 13, any *expression* may be used to specify a named property. To enable tracing of the access expression components without side effects from repeated evaluation, the expression is unpacked into its own statement and assigned to a variable prior to the property access. The expression used for the access is then replaced by the variable. Listing 4.14 shows an example of the transformation. The variable `x` in the Listing is incremented postfix as a side effect to the array access. If the access expression `x++` were evaluated twice, i.e. once for tracing and once for the access, `x` would have an incorrect value.

Listing 4.13: Example of a computed property access. The variable `x` is evaluated to access element 0 of the array.

```
const array = [1, 2, 3], x = 0;

let y = array[x++];
```

Listing 4.14: Computed property access from Listing 4.13 after setup instrumentation.

```
const array = [1, 2, 3], x = 0;

$$computed.push(x++);
let y = array[$$computed.pop()];
```

Ensuring Block Bodies

In JavaScript, a *block* refers to multiple statements enclosed by braces. Certain statements and declarations *allow* but don't necessarily *require* braces to enclose their body. For ex-

ample, if and for statements as well as function declarations don't require their body to be enclosed by braces, when the body only consists of a single statement.

Listing 4.15: Example if statements in JavaScript with only one statement in its body, meaning the braces are not syntactically required. Both if statements are semantically equivalent.

```
// no block body
if (true)
    x = 1;

// block body
if (true) {
    x = 1;
}
```

Since instrumentation for trace generation can possibly add to the body, it is ensured during setup that all of these statements and expressions have a block body. Listing 4.15 shows an example if statement with and without a block body.

Summary

The setup stage consists of two distinct tree traversals, since the splitting of member expression chains must be completed prior to further modifications of member expressions, such as call expression chain splitting and computed property unwrapping. Following the setup, the source code is now ready to be instrumented for trace generation.

4.2 Instrumentation for Trace Generation

To classify and quantify potential microarchitectural side-channel leakage, three areas are of particular importance: memory allocations, memory accesses and branches. Since JavaScript abstracts memory management, memory allocations can't be analyzed when only the JavaScript source code is given. This leaves memory accesses and branches. For trace generation, the original source code is modified in such a way that any action pertaining to either memory accesses or branches results in a trace message being output.

First, as a mandatory preface, the common boilerplate code is injected at the start. This boilerplate code declares various variables that are used for tracing, such as the `$$chain` variable seen in Listing 4.2 on page 20. Additionally, a function for handling unique IDs of objects and the object in charge of handling the formatting of the logging is declared. Unique IDs for each object are needed to understand which object is being accessed and whether this exact object has been accessed before. This is done by defining an "uid" property on each object the first time the `getUid` function is called for an object and otherwise returning the value of its "uid" property. The key of the property is set using `Symbol.for('uid')` to alleviate the risk of a conflicting property existing already.

After the common code, tracing code is selectively added with specific visitors for the node types that are relevant for either memory accesses or branches.

Memory Accesses

Whenever an object is *read* or *written*, a memory access is required. For microarchitectural side-channel leakage, only objects but not primitives are considered threats, since primitives (i.e. numeric literals, boolean literals, etc.) are guaranteed to be constant-time. Table 4.16 shows an overview of all nodes that are targeted by the visitor responsible for logging *read* operations and a description of the condition required for the read, while Table 4.17 on the next page shows of an overview for write operations.

Table 4.16: AST nodes targeted for logging memory reads.

Node	Condition for Read
AssignmentExpression, AssignmentPattern	Assignments that first perform an arithmetic (`*=`, `/=`, `%=`, `+=`, `-=`, `**=`), bitwise (`<<=`, `>>=`, `>>>=`, `&=`, `|=`, `^=`) or logical operation (`&&=`, `||=`, `??=`) and then assign the result, perform a read on the pattern on the left-hand side. The simple assignment operator (`=`) does not trigger a read operation of the left-hand side.
VariableDeclarator, (Assignment)	Destructuring assignment reads the values of the object on the right-hand side.
MemberExpression, OptionalMemberExpression	When a property of an object is accessed the object is read.
CallExpression, OptionalCallExpression	When a function is called the function object is read.
ThisExpression	When the `this` keyword is used the value of `this` is read.
Identifier	Potentially any occurrence of an identifier results in a read of the referenced object. Identifiers in call and catch clause parameters, identifiers used for declarations and identifiers used for labels do *not* result in read operations.

Note that not all nodes that can be linked to read or write operations are explicitly targeted, since multiple nodes cover the same read or write, yet logging is only required once. Additionally, due to general transformations described in Section 4.1 on page 19, not all nodes (e.g. `for` statements) exist after the preliminary setup.

Memory accesses are traced with a combination of the ID of the object that is being accessed along with its potential offset and the location of the access in the original source. With this information, it is possible to determine exactly which objects are accessed when.

To trace read operations, logging code is injected immediately *prior* to the statement that results in a reading memory access. Write operations are logged directly *after* the triggering statement, since for example in the case of a variable declaration, the object can't be accessed prior to declaration.

Table 4.17: AST nodes targeted for logging memory writes.

Node	Condition for Read
AssignmentExpression, AssignmentPattern	Assignments write to the object on the left-hand side.
VariableDeclarator	When a variable is declared with an initial value, a write operation is triggered.
UpdateExpression	Post- and prefix increment and decrement operators write to the referenced object.
ForOfStatement, ForInStatement	Each loop iteration of the `for…of`/`for…in` results in a write operation for the loop variable.

Branches

The second major area of interest — branches — is more complex than memory accesses. Generally, to track the control flow, the source and destination of each jump is required. For conditional branches, this means that each block body following a test is expanded by a trace statement that outputs the location information for the corresponding jump. This instrumentation is illustrated in Listing 4.18.

Listing 4.18: Example `if` statement in JavaScript with a consequence and alternate block. Tracing code is added at the commented lines.

```
if (b) {
    // trace consequence branch
    x = 1;
} else {
    // trace alternate branch
    x = 0;
}
```

Since each call is essentially also a branch in the execution path, these must be traced as well. Consequently, function definitions, i.e. function declarations and function expressions, are instrumented in such a way that the first statement in the function body outputs the location of the function for tracing. This location is the destination of the branch that a corresponding call results in. Additionally, since the `return` statement is also a jump, each function body is guaranteed to end with a `return` statement to allow tracing.

Finally, the calls themselves are traced by adding code immediately *prior* to the call, to log the jump to the function, and immediately *after*, to log the jump back to the context. Listing 4.20 on the following page shows, how a simple function definition and a call are instrumented. Generators are a somewhat special case, since each `yield` expression serves not only as the return to the previous context when the generator is stopped but also as

Listing 4.19: Example function definition and call in JavaScript. The function is defined using function expression syntax.

```
const f = function (x) {
    x++;
}

f(1);
```

Listing 4.20: Example function definition and call from Listing 4.19 after instrumentation. Tracing code is injected into the commented lines.

```
const f = function (x) {
    // trace function location
    x++;
    // trace return
    return; // new (empty) return statement
}

// trace call
f(1);
// trace implicit return from call
```

the starting point of a new context when the generator is resumed. Therefore, tracing code is injected before a `yield` expression for the return and after the `yield`, to log the location of the resumption, akin to the first statement in an instrumented function body.

The last type of branch happens when a `continue` or `break` statement is encountered. In JavaScript switch cases, loops and labeled blocks may be broken out of using the `break` statement. When such a statement is encountered, execution either continues directly after the innermost loop or after the specified labeled block. Similarly, a `continue` statement may be used to jump to the next iteration of a loop.

In summary, both statements result in branches in the control flow. To trace these branches, logging code is injected immediately prior to the `continue` or `break` statement. The destination of the jump is found in one of two ways: when a label is given, the parent nodes of the statement are searched, until the matching label is found; otherwise, the parent nodes are searched for the nearest loop.

The node types that are targeted for branch tracing are detailed in Table 4.21 on the next page.

Summary

Taken together, the primary instrumentation transforms the JavaScript source code in such a way that information regarding the control flow and all memory accesses of objects are logged during execution. The modifications to the code are kept to a minimum to reduce overhead resulting from the instrumentation.

Table 4.21: AST nodes targeted for logging branches.

Node	Explanation
IfStatement, WhileStatement, ForInStatement, ForOfStatement, ForStatement	After the test of the statement is evaluated, a (loop) block is either branched into or not.
ContinueStatement, BreakStatement	When a `break` or `continue` statement is encountered, execution jumps out of the current block.
FunctionDeclaration, FunctionExpression, ArrowFunctionExpression, ObjectMethod, ClassMethod	Function definitions are instrumented to emit location information and guarantee a return statement.
CallExpression	Every call is a guaranteed branch to the function definition.
ReturnStatement, YieldExpression	Both `return` statements and `yield` expressions result in a jump back to the previous context.

After instrumentation, each action that is relevant for potential microarchitectural side-channel leakage — i.e. memory accesses and branches — now generates a trace message. The exact format of these messages will be explained in the next section.

4.3 Trace Format

A trace encompasses all trace messages generated by an execution. This trace should be complete, as in all information that is required to assess whether the code has potential microarchitectural side-channel leakage should be contained, and it should be concise, to ensure that the resulting trace files are small.

Each trace message has two main components: a shorthand — usually a letter — denoting the action, which is being traced, and supplemental information specific to that action. Table 4.22 on the following page shows an overview of all shorthands used by the plugin and their descriptions.

Generally, most trace lines contain at least one location string, which always has the same format as visualized in Figure 4.23 on the next page.

Thus, the location string (1,1,0,6;main.js) would describe the location of the variable declaration `let x` in Listing 4.24 on the following page, whereas (3,5,0,1;main.js) would describe the entire function definition of $foo(y)$.

Store and Load Messages

The most varied trace lines are those of memory stores and loads. While the difference between a store and load is only denoted by a differing letter at the start of the line, the

Table 4.22: Letters used to denote words in the trace format

Shorthand	Action	Description
l	load	memory read
s	store	memory write
c	call	function call
r	return (1)	return from inside a called function
R	return (2)	implicit return immediately following a call
yR	yield (resume)	resumption of a generator
j	jump	jump (e.g. to or from a label)
f	function location	location information of the function being executed

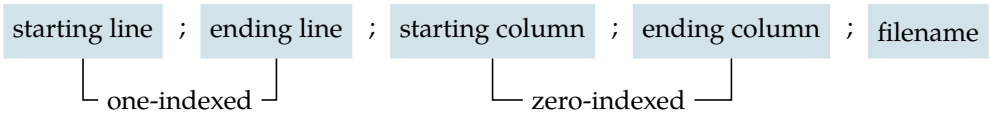

Figure 4.23: Visualization of the format location strings have. Lines are one-indexed and columns are zero-indexed.

Listing 4.24: Minimal example for a location description.

```
1 let x;
2
3 function foo(y) {
4         return y;
5 }
```

information regarding which object was accessed can take on three different appearances. This is due to the fact that an object may be accessed directly or a *property* of the object can be accessed. As described in 2.7 on page 13, JavaScript allows property access by means of two notations: dot notation and bracket notation.

Property accesses are distinguished by a "+" following the unique ID of the accessed object. If dot notation is being described, then the corresponding identifier of the property is appended to the object ID with a plus sign (see line 3 of Listing 4.25). Should the property be accessed using bracket notation, then the *computed value* in brackets is again, appended with a plus sign (see line 4 of Listing 4.25).

Since bracket notation allows property access by evaluating the expression enclosed in brackets, the computed *value* representing the property's name is of interest, instead of the expression prior to evaluation. For example, when accessing an array — which is simply a special type of object — one could use `array[0]`, `array[0*5]` or `array[0-0]` to access the same element, since `0`, `0*5` and `0-0` all evaluate to `0`. Evidently, the computed *value* is easily comparable, unlike the different expressions prior to evaluation.

Listing 4.25: Example trace messages for various memory stores and loads.

```
1 s;5;53,53,6,11;index.js // store uid 5
2 l;5;56,56,7,12;index.js // load uid 5
3 l;1+log;26,26,0,6;index.js // load property 'log' of uid 1
4 l;5+[1];59,59,21,29;index.js // load property '1' of uid 5
```

Following the unique ID, is the location string of the access in the original source code. In summary, a memory access message has the format as illustrated in Figure 4.26.

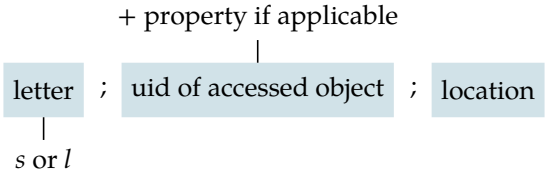

Figure 4.26: Visualization of the format each memory access trace message has.

Call Messages

As detailed in Section 4.2 on page 29, calls result in jumps. To properly trace the control flow and allow a reconstruction of the call tree, multiple types of trace messages are generated during execution. The main message type is for calls. These messages output location information regarding the call and its name as defined by the function `name` property. Specifically, the source location of the call in the original source and the destination location — if possible — are output. Since functions can be imported or defined within objects or classes, it is not always feasible to determine the location of the function that is being called. In this case, the function name, as inferred by the `name` property, is output.

The next type of message exists to supply location information, when the function being called is defined within the same file, yet perhaps as a nested structure, so no destination location could be supplied by the *call* trace message. Function location messages start with an *f* for *function* and log the location of the function definition that is being executed.

Finally, returning from a function generates up to two messages: when a return statement is executed and the implicit return directly after execution returns to the context of the caller. The explicit return message is denoted by a lowercase *r* for *return* and outputs the location of the return statement in the original source. Yield expressions also result in explicit return messages when yielding from a generator. The implicit return directly after the call is marked by an uppercase *R* also for *return* and outputs the location of the call expression that was returned from directly prior. When a generator is resumed by a `generator.next()` call, a yield resumption message is logged, which starts with a *yR* for *yield resumption* and outputs the location of the yield expression immediately prior to mark the destination of the jump from one context to another.

The four different types of trace messages for calls and corresponding information are visualized in Figure 4.27.

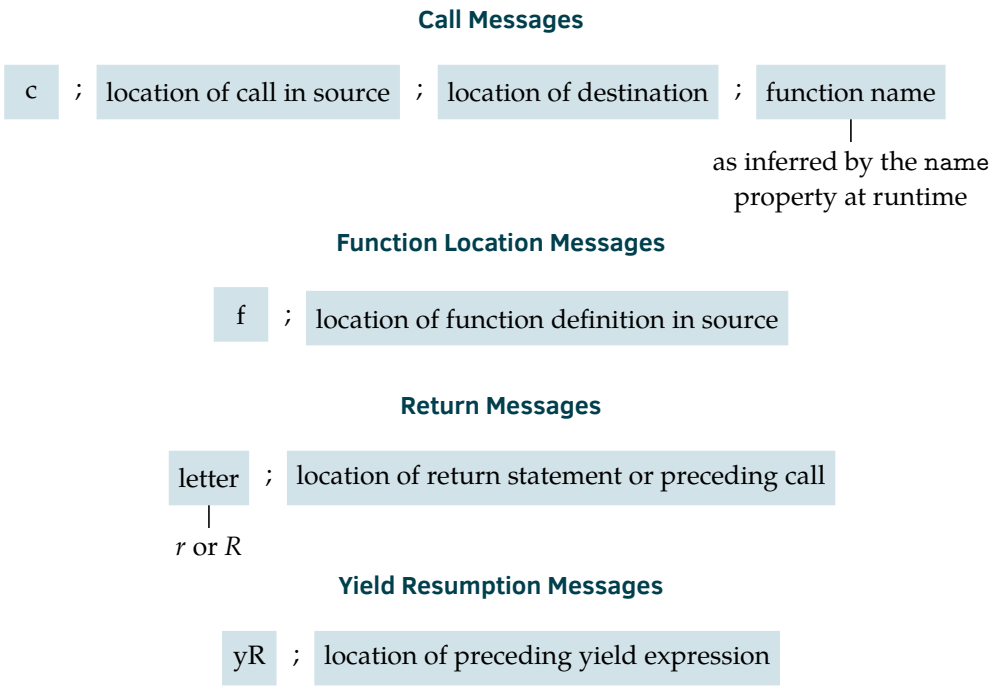

Figure 4.27: Visualization of the format of different call trace messages.

Conditional Branch and Jump Messages

The final two types of messages are structured very similarly. They describe the source and destination locations of a branch. Conditional branches, such as the ones resulting from an `if` statement, start with a *b* for *branch* and log the location of the test, which resulted in the branch (source) and the location of the block that is executed as a result

(destination). Jumps following a `break` or `continue` statement cause a jump trace message, which begins with a *j* for *jump* and logs the location of the `break` or `continue` statement in the original source and the location of the next statement after the block being broken out of (in case of a `break`) or the location of the loop or labeled statement (in case of a `continue`) that will be executed. Both trace message formats are visualized in Figure 4.28.

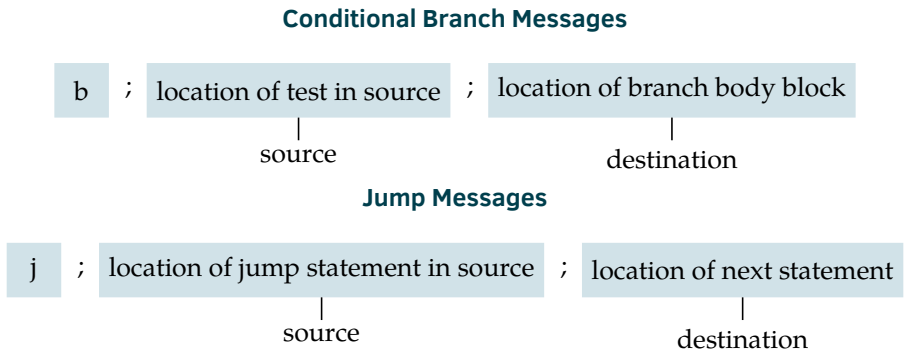

Figure 4.28: Visualization of the format of conditional branch and jump trace messages.

4.4 Implementation Challenges

As is to be expected of any implementation, certain challenges were encountered while creating the new MicroWalk plugin. Some minor issues and their solutions were already previously described, such as transforming `const` into `let` under certain circumstances. Yet, other problems have not been detailed thus far.

First, the preliminary modifications required to allow trace generation posed particular challenges during the implementation of the plugin. Splitting member expression chains was one such challenge. Since it is vital to know, which parent object the property that is being accessed belongs to, it was necessary, to split chains where the object part of the member expression was another member expression. When the split variables were then used in calls, it became apparent that the `this` value was incorrect. Thus, the `call` method of the function object is used to explicitly pass the correct `this` value. The test snippets starting with `member-expression` all showcase how member expression chains are split whereas the test snippets `this.js` and `yield.js` show the correct `this` value being used.

Secondly, when splitting chains and generally injecting new nodes into the AST there is always the danger of creating infinite cycles during traversal. The methods supplied by *babel/traverse* to manipulate paths, i.e. inserting or replacing nodes, add each new node to the traversal queue. Thus, if a visitor inserts a new node that then again matches against a visitor that adds a new node, the traversal queue will clearly grow indefinitely. To avoid this behavior, while still being able to use the *babel/traverse* methods, whenever new nodes should *not* be queued for traversal, these nodes are simply removed from the queue after the corresponding path manipulation.

The final major challenge during the implementation was destructuring assignment.

As explained in Section 2.7 on page 13, destructuring assignment *unwraps* objects during assignment. This means that for an assignment or variable declaration, the *properties* referenced on the left-hand side are in relation to the *object*(s) on the right-hand side. For destructuring assignment without nesting, the properties are correctly resolved against the object on the right-hand side. A limitation of the new plugin is that it only considers the outermost level of a destructuring assignment. In other words, the properties are not completely resolved when a nested object is unwrapped with destructuring assignment. Consequently, the trace is incomplete, since nested properties are not correctly resolved.

Next Steps

The plugin for trace generation is complete, yet as previously shown in Figure 2.3 on page 7, a preprocessor is required to prepare traces for analysis by MicroWalk. The previous preprocessor used by the plugin relying on Jalangi2 is not compatible with the traces generated by the new plugin. This is due to the fact, that the static instrumentation of the AST provides more — especially more *specific* — information than Jalangi2's dynamic instrumentation. Thus, the existing preprocessor must be adapted accordingly.

5

Conclusion and Outlook

5.1 Summary

In this thesis we analyzed the current state of Jalangi2 as the basis for the existing MicroWalk JavaScript plugin and provide a new approach for developing MicroWalk plugins for trace generation using AST manipulation. This new approach is used for a novel MicroWalk JavaScript plugin based on Babel.

Since Jalangi2 is built using libraries that are no longer in active development, the decision was made to implement a new plugin for JavaScript trace generation with a focus on static instrumentation using AST manipulation. Focussing on static JavaScript instrumentation with ASTs meant assessing which nodes are relevant for microarchitectural side-channel leakage analysis. More specifically, nodes which are elements of actions that are susceptible to microarchitectural side-channel leakage; namely memory accesses and branches. This analysis and the exact details of how these actions are traced by the new plugin were explained at length in Chapter 4 on page 19.

During the implementation, more and more syntactic structures that required modifications to enable tracing were identified. This led to certain preliminary transformations of the source code becoming necessary. The trace generation itself, in summary, ensures that each statement or expression resulting in either a memory access or a branch generates a trace message with information required for further analysis by MicroWalk.

5.2 Outlook

In correspondence with the MicroWalk pipeline, the next step towards using the plugin with MicroWalk, is writing a preprocessor which transforms the generated traces into a format that MicroWalk understands. As mentioned at the end of Chapter 4, this preprocessor is left for future work. Once a preprocessor has been implemented, it will be possible to analyze modern cryptographic JavaScript libraries which use language features of ES6 and upwards.

Furthermore, the technique described in Chapter 4 on page 19 can be used to create further trace generation plugins for other languages, such as Python.

Bibliography

[1] *Acorn library source code*. https://github.com/acornjs/acorn. Accessed: 2023-12-09.

[2] *Babel Source*. https://github.com/babel/babel. Accessed: 2023-09-16.

[3] Barberis, E., Frigo, P., Muench, M., Bos, H., and Giuffrida, C. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 971–988. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/barberis.

[4] Christophe, L., De Roover, C., and De Meuter, W. Poster: Dynamic analysis using JavaScript proxies. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE. 2015, pp. 813–814.

[5] *ECMA-262, ECMAScript Language Specification, 5th Edition*. Ecma International. Rue du Rhône 114, 1204 Geneva, Switzerland, 2009.

[6] *ECMA-262, ECMAScript Language Specification, 6th Edition*. Ecma International. Rue du Rhône 114, 1204 Geneva, Switzerland, 2015.

[7] *ECMA-262, ECMAScript Language Specification Overview*. Ecma International. Rue du Rhône 114, 1204 Geneva, Switzerland, 2023.

[8] *Escodegen library source code*. https://github.com/estools/escodegen. Accessed: 2023-12-09.

[9] *ESTree Specification*. https://github.com/estree/estree. Accessed: 2023-12-09.

[10] Gandolfi, K., Mourtel, C., and Olivier, F. Electromagnetic analysis: Concrete results. In: *Cryptographic Hardware and Embedded Systems—CHES 2001: Third International Workshop Paris, France, May 14–16, 2001 Proceedings 3*. Springer. 2001, pp. 251–261.

[11] Guo, S.-y., Ficarra, M., Gibbons, K., and community, E. *ECMAScript® 2024 Language Specification - 13.3 Left-Hand-Side Expressions*. https://tc39.es/ecma262/multipage/ecmascript-language-expressions.html#sec-left-hand-side-expressions. Accessed: 2023-12-20.

[12] *Jalangi2 Source*. https://github.com/Samsung/jalangi2. Accessed: 2022-11-16. Samsung.

[13] Jančár, J., Fourné, M., Braga, D. D. A., Sabt, M., Schwabe, P., Barthe, G., Fouque, P.-A., and Acar, Y. "They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 632–649.

[14] Jančár, J. and Kario, H. *List of constant-time verification tools*. https://crocs-muni.github.io/ct-tools/. Accessed: 2023-11-21.

[15] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre Attacks: Exploiting Speculative Execution. In: *CoRR* abs/1801.01203, 2018. URL: http://arxiv.org/abs/1801.01203.

[16] Kocher, P., Jaffe, J., and Jun, B. Differential power analysis. In: *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*. Springer. 1999, pp. 388–397.

[17] Kocher, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*. Springer. 1996, pp. 104–113.

[18] Langley, A. *ctgrind Source Code*. https://github.com/agl/ctgrind. Accessed: 2023-11-21.

[19] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-level cache side-channel attacks are practical. In: *2015 IEEE symposium on security and privacy*. IEEE. 2015, pp. 605–622.

[20] Nikulin, I. and Sridharan, M. *Esotope library source code*. https://github.com/inikulin/esotope. Accessed: 2023-12-09.

[21] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2015, pp. 1406–1418.

[22] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: the case of AES. In: *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*. Springer. 2006, pp. 1–20.

[23] *Pin 3.28 User Guide*. Accessed: 2023-12-09. Intel Corporation. Mission College Boulevard 2200, 95052-8119 Santa Clara California, USA.

[24] Ridder, F. de, Frigo, P., Vannacci, E., Bos, H., Giuffrida, C., and Razavi, K. SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1001–1018. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/ridder.

[25] Schafer, S. With Capital in Panic, Pizza Deliveries Soar. In: *Washington Post*:D1, Dec. 1998.

[26] *Valgrind Website*. https://valgrind.org/. Accessed: 2023-11-21.

[27] Wichelmann, J., Moghimi, A., Eisenbarth, T., and Sunar, B. MicroWalk: A framework for finding side channels in binaries. In: *Proceedings of the 34th Annual Computer Security Applications Conference*. 2018, pp. 161–173.

[28] Wichelmann, J., Sieck, F., Pätschke, A., and Eisenbarth, T. Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 2915–2929.

[29] Yarom, Y. and Falkner, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. URL: https://

www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

A

Appendix

A.1 Babel AST Node Types

As defined by the Babel AST specification document and *babel/types*. See also the specification document[1] and the API of *babel/types*[2].

General

- Identifier
- PrivateName
- Programs

Literals

- RegExpLiteral
- NullLiteral
- StringLiteral
- BooleanLiteral
- NumericLiteral
- BigIntLiteral
- DecimalLiteral

Functions

Covers function declarations and (arrow) function expressions. See `https://babeljs.io/docs/babel-types#function` for further details.

Statements

- ExpressionStatement
- BlockStatement

[1] `https://github.com/babel/babel/blob/main/packages/babel-parser/ast/spec.md`
[2] `https://babeljs.io/docs/babel-types#api`

- EmptyStatement
- DebuggerStatement
- WithStatement
- **Control flow**
  - ReturnStatement
  - LabeledStatement
  - BreakStatement
  - ContinueStatement
- **Choice**
  - IfStatement
  - SwitchStatement
  - SwitchCase
- **Exceptions**
  - ThrowStatement
  - TryStatement
  - CatchClause
- **Loops**
  - WhileStatement
  - DoWhileStatement
  - ForStatement
  - ForInStatement
  - ForOfStatement

Declarations

- FunctionDeclaration
- **VariableDeclaration**
  - VariableDeclarator

Misc

- Decorator
- Directive
- DirectiveLiteral
- InterpreterDirective

Expressions

- Super
- Import
- ThisExpression
- ArrowFunctionExpression
- YieldExpression

- AwaitExpression
- ArrayExpression
- **ObjectExpression**
  - ObjectMember
  - ObjectProperty
  - ObjectMethod
- RecordExpression
- TupleExpression
- FunctionExpression
- **Unary operations**
  - UnaryExpression
  - UnaryOperator
  - UpdateExpression
  - UpdateOperator
- **Binary operations**
  - BinaryExpression
  - BinaryOperator
  - AssignmentExpression
  - AssignmentOperator
  - LogicalExpression
  - LogicalOperator
  - SpreadElement
  - ArgumentPlaceholder
  - MemberExpression
  - OptionalMemberExpression
- BindExpression
- ConditionalExpression
- CallExpression
- OptionalCallExpression
- NewExpression
- SequenceExpression
- ParenthesizedExpression
- DoExpression
- ModuleExpression

Template Literals

- TemplateLiteral
- TaggedTemplateExpression
- TemplateElement

Patterns

- ObjectPattern

- ArrayPattern
- RestElement
- AssignmentPattern

Classes

- ClassBody
- ClassMethod
- ClassPrivateMethod
- ClassProperty
- ClassPrivateProperty
- StaticBlock
- ClassDeclaration
- ClassExpression
- MetaProperty

Modules

- ModuleSpecifier
- **Imports**
  - ImportDeclaration
  - ImportSpecifier
  - ImportDefaultSpecifier
  - ImportNamespaceSpecifier
  - ImportAttribute
- **Exports**
  - ExportDeclaration
  - ExportNamedDeclaration
  - ExportSpecifier
  - ExportNamespaceSpecifier
  - ExportDefaultDeclaration
  - ExportAllDeclaration