Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications

Jan Wichelmann
University of Lübeck
Lübeck, Germany
j.wichelmann@uni-luebeck.de

Anna Pätschke
University of Lübeck
Lübeck, Germany
a.paetschke@uni-luebeck.de

Florian Sieck
University of Lübeck
Lübeck, Germany
florian.sieck@uni-luebeck.de

Thomas Eisenbarth
University of Lübeck
Lübeck, Germany
thomas.eisenbarth@uni-luebeck.de

ABSTRACT
Secret-dependent timing behavior in cryptographic implementations has resulted in exploitable vulnerabilities, undermining their security. Over the years, numerous tools to automatically detect timing leakage or even to prove their absence have been proposed. However, a recent study at IEEE S&P 2022 showed that, while many developers are aware of one or more analysis tools, they have major difficulties integrating these into their workflow, as existing tools are tedious to use and mapping discovered leakages to their originating code segments requires expert knowledge. In addition, existing tools focus on compiled languages like C, or analyze binaries, while the industry and open-source community moved to interpreted languages, most notably JavaScript.

In this work, we introduce Microwalk-CI, a novel side-channel analysis framework for easy integration into a JavaScript development workflow. First, we extend existing dynamic approaches with a new analysis algorithm, that allows efficient localization and quantification of leakages, making it suitable for use in practical development. We then present a technique for generating execution traces from JavaScript applications, which can be further analyzed with our and other algorithms originally designed for binary analysis. Finally, we discuss how Microwalk-CI can be integrated into a continuous integration (CI) pipeline for efficient and ongoing monitoring. We evaluate our analysis framework by conducting a thorough evaluation of several popular JavaScript cryptographic libraries, and uncover a number of critical leakages.

CCS CONCEPTS
• Security and privacy → Software and application security; Cryptography; • Software and its engineering → Software verification and validation.

1 INTRODUCTION
Collection of sensitive data is common in today’s cloud and Internet of Things (IoT) environments, and affects everyone. Protecting this private and sensitive data is of utmost importance, therefore requiring secure cryptography routines and secrets. However, especially the cloud allows attackers to observe the execution of victim code using side-channels in co-located environments [26]. These attacks range from Last Level Cache (LLC) [35] and de-duplication attacks [34] to the observation of memory access patterns [64] or main memory access contention [44]. The spatial resolution depends on the granularity of the attacked buffer and the temporal resolution on the capabilities of the attacker, meaning either the ability to achieve a sufficiently high measurement frequency [65] or to interrupt and pause the victim code [14, 64].

To avoid side-channel vulnerabilities, programmers should write constant-time code, i.e., software which does not contain input or secret-dependent control flow or memory accesses. Depending on the problem at hand, this can be achieved by different means: For example, a secret-dependent data access may be replaced by accessing every element of the target array and then choosing the correct one with a mask. Conditionals can be adjusted by always executing both branches and then selecting the result.

However, for complex projects like large cryptographic libraries, finding such vulnerabilities is a difficult and time-intensive task. Thus, the research community has developed a number of analysis strategies [5, 13, 16, 17, 33, 46, 60, 62, 63], that aim at automating the detection of side-channel leakages in a given code base. However, a recent study [29] that conducted a survey between crypto library developers found that while most developers were aware of and welcome those tools, they had major difficulties using them due to bad usability, lack of availability and maintenance or high...
resource consumption. The authors worked out a number of recommendations for creators of analysis tools: The tools should be well-documented and easily usable, such that adoption requires low effort from the developer. Another focus is on compatibility: The analysis shouldn’t require use of special languages or language subsets. Finally, the tools should aid efficient development, i.e., quickly yield results with less focus on completeness, making them suitable for inclusion in a continuous integration (CI) workflow.

In this work, we study how these challenges can be addressed, and adapt the existing Microwalk [63] framework to fit the given objectives. Microwalk was originally designed for finding leakages in binary software, for which it generates a number of execution traces for a set of random inputs and then compares them. The dynamic analysis approach of Microwalk is quite fast, as it does only run the target program several times, and then compares the resulting execution traces with a simple linear algorithm. However, due to the simplistic leakage quantification, the resulting analysis reports contain a lot of potential vulnerabilities, with little or even misleading information about their cause. This makes it difficult to assess their severity and address them efficiently, especially for complex libraries. Finally, the initial setup can be time-consuming, as the different components need to be compiled from source.

We mitigate these issues by designing Microwalk-CI, which features a new leakage analysis algorithm that combines the performance benefits of dynamic analysis with an accurate leakage localization and quantification, easing the assessment and investigation of the reported leakages. In addition, we add support for running Microwalk-CI in an automated environment like a CI pipeline, and create a Docker image that contains Microwalk-CI and its dependencies for easy use. Finally, we create simple templates that allow quick adoption of Microwalk-CI’s analysis capabilities into a cryptographic library’s CI workflow with little effort by the developer.

During the research on leakage detection tools, as part of their evaluation, many vulnerabilities in popular cryptographic libraries have been uncovered and fixed. However, the developer community is moving away from compiled languages like C or C++ and instead embraces interpreted scripting languages like JavaScript or Python. In fact, the 2021 Stack Overflow developer survey and the January 2021 Redmonk programming language ranking found that those two languages are the most popular, both for private and professional contexts [45, 56]. JavaScript was originally designed as a client-side language for web browsers, but, with the arrival of Node.js [40], it has seen growing adoption for server-side software as well. Consequently, the community has come up with a number of cryptographic libraries written in pure JavaScript. However, due to the lack of appropriate tooling and attention of the research community, these libraries have never been vetted for their robustness against side-channel attacks, which is worrying given the fact that the servers using them may be hosted in IaaS cloud environments.

To address this, Microwalk-CI offers a novel method for applying Microwalk’s original binary analysis algorithms to JavaScript libraries by using the Jalangi2 [48, 55] source code instrumentation library to generate compatible traces. The new tracing backend comes with a simple code template and supports full automation, such that the analysis can be easily added to the CI workflows of respective libraries. We evaluate several popular JavaScript cryptographic libraries, uncovering a number of high-severity leakages.

By supporting the analysis of JavaScript, we strive to improve the security of software and raise awareness for the importance of constant-time cryptographic code in the community of web and cloud developers. The underlying concepts of our source-based trace generator can be used for building analysis support for other programming languages as well, making side-channel leakage analysis available for all common platforms and at a low barrier.

1.1 Our Contribution
In summary, we make the following contributions:

- We introduce a novel call tree-based analysis method, which allows efficient and accurate localization and quantification of leakages.
- We show the first dynamic leakage analysis tool for JavaScript libraries.
- We propose a new approach for integrating a fully automated timing leakage analysis into the crypto library development workflow, which requires low effort from the developer and immediately reports newly introduced vulnerabilities.
- We evaluate the new analysis framework with several widely-used JavaScript libraries and uncover a significant number of previously unaddressed leakages.

The source code of Microwalk-CI is available at https://github.com/microwalk-project/Microwalk.

1.2 Disclosure
We contacted the authors of the affected libraries, informed them about our findings and offered to aid in fixing the vulnerabilities.

The author of elliptic acknowledged the discovered vulnerabilities, but noted that the package is no longer maintained and that fixing the vulnerabilities would require major changes, as side-channel resistance wasn’t part of the underlying design considerations. There was no response from the other library authors.

2 BACKGROUND
2.1 Microarchitectural Timing Attacks
Implementations of cryptographic algorithms are often run on hardware resources that are shared between different processes. If the code exhibits secret-dependent behavior, malicious processes can use the resulting information leakage to extract secrets like private keys through side-channel analysis. Cache attacks are a prominent example for exploiting resource contention with a victim process: By measuring the time it takes for repeatedly clearing and accessing a specific cache entry, the attacker can see whether the victim accessed a similar cache entry in the meantime [1, 10, 42, 53, 65]. Other attack vectors include the translation lookaside buffer (TLB) [22] and the branch prediction unit [2].

The most widely used software countermeasure against these attacks is writing constant-time code that does not contain secret-dependent memory accesses or branches, and that uses instructions that do not come along with operand-dependent runtime [12]. There exists a variety of tools [5, 13, 16, 17, 33, 46, 60, 62, 63], that feature different analysis approaches. Some of these tools are open-source, with varying performance and usability [29].
2.2 Microwalk

Microwalk [63] is a framework for checking the constant-time properties of software binaries in an automated fashion. It follows a dynamic analysis approach, i.e., it executes the target program with a number of random inputs and collects execution traces, which contain branch targets, memory allocations and memory accesses. This is done through a three-stage pipeline, where traces are generated, preprocessed, and analyzed. Each stage has various modules, which are chosen by the user depending on their application. Furthermore, Microwalk has a plugin architecture, that allows easy extension by loading custom modules.

Currently, Microwalk only has one trace generation module, which is based on Intel Pin [27] and produces traces for binary software. Correspondingly, there is a preprocessor module that converts the raw traces generated by Pin into Microwalk’s own format. Finally, these preprocessed traces can be fed into a number of analysis modules, e.g., for computing the mutual information between memory access patterns and inputs, or for dumping the preprocessed traces in a human-readable format.

2.3 Mutual Information and Guessing Entropy

Mutual information (MI) quantifies the interdependence of two random variables, i.e., it models how much information an attacker can learn about one variable on average by observing the other one [23]. It has been widely used for quantifying side-channel leakages [9, 28, 63, 67].

The mutual information of the random variables \( X : K \rightarrow \mathcal{X} \) and \( Y : L \rightarrow \mathcal{Y} \) is defined as

\[
I(X, Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} \Pr[X = x, Y = y] \cdot \log_2 \left( \frac{\Pr[X = x, Y = y]}{\Pr[X = x] \cdot \Pr[Y = y]} \right).
\]

The information is measured in bits. In our setting, the random variable \( X \) represents a secret and \( Y \) the information that can be gathered by observing the system state through a side-channel.

The guessing entropy (GE) of a random variable \( X : K \rightarrow \mathcal{X} \) quantifies the average number of guesses that have to be made in order to guess the value of \( X \) correctly [32]. If \( \mathcal{X} \) is indexed such that \( \Pr[X = x_i] \geq \Pr[X = x_j] \) for \( x_i, x_j \in \mathcal{X} \) and \( i \leq j \), the guessing entropy is defined as

\[
G(X) = \sum_{1 \leq i \leq |\mathcal{X}|} i \cdot \Pr[X = x_i].
\]

The conditional guessing entropy (conditional GE) \( G(X \mid Y) \) for random variables \( X \) and \( Y \) is defined as

\[
G(X \mid Y) = \sum_{y \in \mathcal{Y}} \Pr[Y = y] \cdot G(X \mid Y = y).
\]

\( G(X \mid Y) \) measures the expected number of guesses that are needed to determine the value of \( X \) for a known value of \( Y \).

A variant of the conditional GE, the minimal conditional guessing entropy (minimal GE), determines the lower bound of expected guesses. It is defined as

\[
\hat{G}(X \mid Y) = \min_{y \in \mathcal{Y}} G(X \mid Y = y),
\]

i.e., it outputs the minimal number of guesses that are needed to find out one of the possible values of \( X \).

2.4 JavaScript Instrumentation

JavaScript code can be instrumented in different ways, each coming with their own benefits and drawbacks.

FoxHound [49] modifies Firefox’s JavaScript engine. While this allows many optimizations, it comes with the downside of being constrained to one specific JavaScript engine and requiring constant maintenance to keep up with the upstream project. OpenTelemetry [41] and Google’s tracing framework [21] create program traces to monitor and profile software, but require the developer to insert instrumentation calls into their source code manually. While being very specific and thus only introducing the necessary overhead, they are not generally applicable without a lot of manual effort.

Lastly, the JavaScript code can be dynamically instrumented in a source-to-source fashion. Jalangi2 [48, 52, 55] wraps the loading process of JavaScript files and injects instrumentation code into the source code. The user of the instrumentation framework can write and register custom callback routines, which are supplied with the current execution state. This approach comes with a certain overhead, but it is flexible and works with arbitrary JavaScript code without manual adjustments.

3 A FAST LEAKAGE ANALYSIS ALGORITHM

We propose a new leakage analysis algorithm that is optimized for quickly delivering detailed leakage information, aiding developers in efficiently locating and fixing issues. Before we dive into the algorithm, we define the leakage model and discuss the objectives a thorough leakage analysis must meet. Then, we describe how the traces are processed to build a call tree, which in a final step is broken down to compute leakage metrics for specific instructions.

3.1 Leakage Model

To ensure that we detect all leakages which may be exploited by current and future attack methods, we choose a strong leakage model: An attacker tries to extract secret inputs from an implementation through a side-channel attack, which allows them to get a trace of all executed instructions and all accessed memory addresses. They also have access to all public inputs and outputs.

Under certain conditions, a hypervisor/OS-level adversary can single step instructions [4, 38, 54], or have below cache-line resolution [37, 66]. However, for more relaxed adversarial scenarios like cross-VM attacks, granularities of 32 or 64 bytes and hundreds of instructions may be more appropriate. Adjusting the processing of the leakage accordingly allows an analysis under such a leakage model as well, but, we believe that the most conservative approach should be applied, i.e., assuming a maximum resolution attacker. Attacks exploiting speculative execution are considered off-scope, as we focus on leakages caused by actual secret-dependent control flow or memory accesses, i.e., code paths that are reached architecturally.

This leakage model and the following analysis approach are consistent with the models used by Microwalk [63] and DATA [62].

3.1.1 Analysis approach. The leakage model can be turned into a dynamic analysis approach by making the following observation: Since the attacker tries to infer a secret solely by looking at an execution trace and public inputs/outputs, they can only succeed if the trace depends on the secret. I.e., if changing the secret does
3.2 Objectives

For an efficient and useful dynamic leakage analysis, we identified three major objectives: Accurate localization of leakages, a quantification of leakage severity, and performance.

Localization. While varying address traces for a memory read instruction are a clear sign that there is leakage, which can be extracted by monitoring that particular instruction [63], they do not indicate where the leakage is actually caused. E.g., a non-constant-time function may be called two times, once with a secret-dependent parameter, and once varying number of times in a loop, but with a constant parameter (Figure 1). A correct analysis should distinguish the two invocations of lookup and mark the table access in line 12 as leaking for the first invocation (line 2); for the second invocation (line 6), the secret-dependent branch in line 5 should be reported, as the table access in line 12 itself does not add any leakage.

Quantification. In addition to an accurate localization, there is a need for a rough quantification of the severity of leakages. For example, a chain of nested if statements may only leak a few bits of the secret each, but the leakage aggregates up to a point which allows an attacker to easily distinguish different secrets just by looking at the resulting sequence of branch instructions. At the same time, a lone if statement which merely handles a special case during key file parsing (e.g., whether a parameter has some additional byte) does not necessarily pose an urgent problem. The analysis should assign each leakage with a score allowing the attacker to easily distinguish different secrets.

Performance. Finally, for integrating the leakage analysis into a development workflow, performance is important: When checking whether a proposed change impacts security, or whether a given patch fixes a previously discovered leakage, the developer should not need to wait several ten minutes or hours until analysis results are available. The analysis should be efficient enough to run it both on a standard developer machine and in a hosted CI environment.

3.3 Algorithm Idea

In order to find leakages, we need to compare the generated traces, and find sections where they diverge and, later, merge again. However, due to the performance requirements and the immense size of traces, especially for asymmetric primitives, we cannot afford running a traditional diff or trace alignment algorithm, which usually have quadratic complexity. At the same time, we do not want to lose information, as we want to accurately pinpoint the detected leakages. Thus, we opt for a data structure that preserves all necessary information in an efficient way, and which allows to conduct a thorough leakage analysis which can discover and quantify trace divergences in linear time.

For that, we merge the traces into a call tree, where each function call and a few other trace entries form the nodes, and where subsequent function calls and trace divergences generate branches. Each node holds the IDs of the traces which reach that node. The tree can be built on-the-fly while the traces are processed, so it can be integrated into a leakage analysis pipeline like the one offered by Microwalk. After the traces have been processed, a final step traverses the tree and evaluates for each instruction in each call stack, whether it caused a divergence and how severe that divergence is. In the following sections, we elaborate on the respective steps.

3.4 Step 1: Building the Call Tree

We merge the traces into a tree in a greedy way, i.e., we simultaneously iterate over a trace and the current tree entries, and add the trace entries to the tree. In order to save memory and get a readable representation of the traces with little tree depth, we can exploit the fact that traces of constant-time implementations tend to have long shared sequences without any differences, and thus use a radix trie instead of a plain tree, such that each node holds an as long as possible sequence of consecutive trace entries.

3.4.1 Types of trace entries. In order to address the leakage model, the execution traces used by Microwalk contain information about branches, memory allocations and memory accesses.

Branches cover call, return and jump instructions. A branch trace entry has a source address, a target address and a taken bit that indicates whether the branch was taken or skipped (e.g., due to a failed comparison). The source and target addresses consist each of an image ID (i.e., the binary which contains the corresponding instruction) and an offset.

Memory allocations are used to keep track of memory blocks on the heap and stack. Each time the analyzed program calls malloc or a similar function, a new allocation block is registered with a unique ID and the allocation block size.

Memory accesses contain the image ID and offset of the corresponding instruction and the allocation block ID and offset of the accessed address. This relative addressing allows to compare traces
While traces 1 to 4 share the entries in the left node, they differ at the jump statement at location A: Traces 1 and 3 jump to location B, while traces 2 and 4 jump to C. Here, each case gets its own split node, and processing of trace entries is resumed there.

**Figure 2:** A generic trace divergence with two split nodes. While traces 1 to 4 share the entries in the left node, they differ at the jump statement at location A: Traces 1 and 3 jump to location B, while traces 2 and 4 jump to C. Here, each case gets its own split node, and processing of trace entries is resumed there.

The handling of equal and conflicting trace entries depends on the respective type. Split nodes are only created when a function call or a jump targets a different instruction than the already existing trace entry, as the resulting sub tree may be fairly different. Other differences like varying memory access offsets are only recorded in the respective trace entry, as they don’t affect control flow and the current entry is thus likely followed by other, non-conflicting entries.

A function call is handled by creating a new tree node at the current position in the list of consecutive trace entries of the current tree node. Afterwards, the current node is pushed onto a stack and the new call node is set as the current node, such that subsequent trace entries are stored in the new node. When encountering a return statement, the last node is popped from the stack, and insertion of trace entries is resumed after the earlier created call node. If the target address of the current call entry does not match the target address recorded in an existing call node, a split is triggered.

If a conflict between an existing and a new trace entry is detected, the algorithm generates two new split nodes: One node receives the original conflicting trace entry, the remaining consecutive trace entries and the split node list of the current node; the other node is initialized with the new conflicting trace entry and an empty split node list. The branches to both nodes are annotated with the corresponding trace IDs. The current node is then set to the new split node, such that the new trace entries end up in the new node. The call node stack is not updated, i.e., the next return statement ends the divergence and restores the state before the last call node. This way, we can recover from a trace divergence and discover additional leakages in other function calls.

Cases where there are more than two possible targets for an instruction (e.g., an indirect jump) are handled appropriately, by generating further split nodes at the same level.

### 3.5 Step 2: Leakage Analysis

After trace processing has concluded, we have a call tree that encodes the similarities and differences of all traces. We now perform a final step that collects this information and computes leakage measures, such that we can assign leakage information to each instruction, meeting our localization and quantification objectives.

#### 3.5.1 Building call stacks with trace ID trees per instruction

First, we consolidate the call tree into a number of call stacks, and store the trace split information for each instruction in the corresponding call stack. The split information consists of trace ID trees, which encode how multiple executions of the given instruction for a certain function invocation led to trace divergence. This greatly simplifies the computation of leakage measures for individual instructions, and allows to display expressive information about the leakage behavior of a given instruction to the developer. If a function is called multiple times (i.e., the same call stack occurs repeatedly), additional trace ID trees are created (no merging).

When a split is encountered, new child nodes for each edge of the split are added to the trace ID tree for the responsible instruction. Figure 4 illustrates the resulting trace ID tree for a simple program counting bits in a secret variable: When the jump instruction in question is encountered first, all traces are identical (tree level 0). At that point, execution diverges for traces with an even versus an odd secret. After the second iteration, traces are again split depending on the second bit of the secret. In the end, there are four different possible traces for the given function call.
The MI of the trace ID $X$ and the observed trace $Y$ is defined as

$$I(X, Y) = \sum_{k=1}^{T} \sum_{i=1}^{n} \log_2 \left( \frac{Pr[X = k, Y = i]}{Pr[X = k] \cdot Pr[Y = i]} \right).$$

With

$$Pr[X = k, Y = i] = \begin{cases} \frac{1}{n} & \text{if } k \not\in L_i \\ \frac{1}{n} & \text{if } k \in L_i \end{cases}$$

we get

$$I(X, Y) = \frac{|L_i|}{n} \cdot \log_2 \left( \frac{1}{\frac{1}{n} \cdot |L_i| / n} \right) \approx \frac{1}{n} \sum_{i=1}^{T} \frac{|L_i| \cdot \log_2 \left( \frac{n}{|L_i|} \right)}{n}.$$

The value of $I(X, Y)$ can be interpreted as bits: In the best case, there is only one leaf containing all trace IDs, such that the attacker learns nothing (0 bits). In the worst case, with one leaf for each trace ID, the attacker learns $\log_2(n)$ bits. The MI of the example in Figure 4 is $\frac{1}{2} \cdot 2 \cdot \log_2(3) + 2 \cdot 1 \cdot \log_2(6) \approx 1.33$ bits.

This metric has a few drawbacks: Due to its logarithmic nature, with an increasing number of traces it only grows slowly. Another shortcoming is the averaging, i.e., a high leakage in a few cases may get suppressed by the smaller leakage of all other cases. Finally, it may be mistakenly interpreted as additive due to its "bits" unit (i.e., 10 instructions leaking 3 bits each does not mean that there is a leakage of 30 bits). However, it does perform well for small and balanced leakages, e.g., when an instruction constantly divides the traces into two groups of similar size.

**Conditional guessing entropy** measures the expected number of guesses an attacker needs for associating a given trace with a secret input. The conditional GE $G(X | Y)$ for determining a trace ID, modeled as random variable $X$, for a known value of an observed trace (random variable $Y$) is calculated as

$$G(X | Y) = \sum_{i=1}^{T} \Pr[Y = i] \cdot G(X | Y = i)$$

$$= \sum_{i=1}^{T} \Pr[Y = i] \cdot \sum_{k=1}^{T} k \cdot \Pr[X = k | Y = i].$$

Since

$$\Pr[X = k | Y = i] = \begin{cases} 0 & \text{if } k \not\in L_i \\ \frac{1}{|L_i|} & \text{if } k \in L_i, \end{cases}$$

we can simplify (1) to

$$G(X | Y) = \frac{1}{|L_i|} \sum_{i=1}^{T} \Pr[Y = i] \cdot \frac{1}{|L_i|} \sum_{k=1}^{T} k = \frac{1}{2n} \sum_{i=1}^{T} |L_i| \cdot (|L_i| + 1).$$

Note that the value of $G(X | Y)$ is upper-bounded by $\frac{T^2}{4n}$, which is the best case where there is only one leaf which contains all trace IDs, i.e., all traces are identical. For the example in Figure 4, we get $G(X | Y) = \frac{1}{2n} (6 + 2 + 6 + 2) = 1.33$ guesses.

Small values for the conditional GE convey that an instruction sequence leads to almost unique traces, implying that there is widespread leakage affecting most to all traces. On the other side, a high value means that most traces are similar and do not leak much information. However, this being an average measure just like MI, there may well be special cases where there is a very high leakage.
Those risk being obscured by this metric, thus we add an additional worst-case metric designed for catching these cases.

Minimal conditional guessing entropy measures the minimal number of guesses an attacker needs for associating a given trace with a secret input. It is calculated similarly to the conditional GE, but takes the minimum of all individual outcomes instead of weighting them:

$$\hat{G}(X | Y) = \min_{i=1,...,|Y|} G(X | Y = i) = \min_{i=1,...,r} |L_i| + 1 \over 2.$$

For the example in Figure 4, we get $\hat{G}(X | Y) = \min\{1.5, 1, 1.5, 1\} = 1$ guesses, i.e., there is at least one trace that is unique.

Minimal GE is the most definite leakage measure; it gives the number of guesses needed for the trace which leaks most. A high value for the minimal GE affirms that there is no outlier with high leakage. We thus recommend using this metric when evaluating the severity of a detected leakage.

3.5.3 Leakage severity and score. While the full analysis report provides detailed information about each leakage, we also seek to condense this information into a single, uniform score, such that the developer can quickly prioritize. That score should require little context: The developer should not need to be familiar with entropy, nor know analysis details like the particular number of test cases, which determines the upper bounds for the various metrics. Additionally, providing a single score allows easy integration of the leakage report into the user interface of modern development platforms like GitLab. The platform can then use that score for sorting and assigning a severity to the leakage.

We chose minimal GE for computing the leakage score, as it represents the worst-case leakage. Instead of reporting the minimal GE value directly, we map it onto a linear scale of 0 to 100, where 0 corresponds to a minimal GE of $\frac{|L|}{2}$ (i.e., no leakage), and 100 corresponds to a minimal GE of 1 (i.e., maximum leakage). If there are multiple trace ID trees for a given instruction (see Section 3.5.1), we show the mean and the standard deviation over the individual minimal GE values.

3.6 Implementation

We implemented the described algorithm as a new analysis module in Microwalk-CI’s source tree. It integrates directly into the leakage analysis pipeline, i.e., it receives and handles preprocessed traces from the previous pipeline stage, the trace preprocessor. The tree is implemented as a recursive data structure, where each node holds a list of successor and split nodes. We do not store the consecutive non-diverging trace entries as a plain ITraceEntry list (as is suggested in the algorithm description), but as full-featured tree nodes as well. Apart from making the code more readable, this simplifies adding new divergences and storing temporary data for the final leakage analysis step, at the cost of additional memory overhead (we discuss this trade-off in Section 6.2).

Our implementation offers functionality for generating leakage reports and other detailed analysis result files optimized for readability, including an optional full call tree dump for debugging purposes. All features can be controlled via the Microwalk-CI configuration file infrastructure, allowing easy adoption of the new analysis module. In total, the module has 1,363 lines of C# code.

4 JAVASCRIPT LEAKAGE ANALYSIS

We now show how we can apply Microwalk-CI’s generic analysis methods to JavaScript libraries, despite them being originally designed for binary analysis. First, we present a simple trace generator relying on the Jalangi2 instrumentation framework. Then, we show how these traces can be preprocessed such that they use the generic trace format from Microwalk-CI.

4.1 Instrumenting JavaScript code

Microwalk-CI expects multiple execution traces with varying secret input for the analyzed target function. These execution traces are then fed into various analysis modules for finding non-constant-time behavior, i.e., control flow or data flow dependencies from secret input. A trace needs to contain the following information:

- Address and size of all loaded program modules (e.g., binaries or source files, called “images” internally);
- the control-flow of the analyzed program, encoded as a sequence of branch source and target addresses;
- address and size of all heap memory objects; and
- the instructions and target addresses of all memory accesses.

We translate this to JavaScript by collecting a trace of all executed code lines, and recording access offsets to any object or array. For instrumentation, we use Jalangi2 [48]. Jalangi2 instruments the code at load time by inserting callbacks before and after certain source tokens, e.g., conditionals, expressions or return statements.

First, we register the provided SMemory analysis module, which assigns a shadow object to each object, that contains a unique ID and the object value, allowing us to map accesses to known objects. We then create our own analysis front-end, called tracer, which registers some callbacks to record the necessary information and write it to a file for further processing. The tracer has 252 lines of code, and is chained after the SMemory analysis, which supplies the means for memory access tracking.

4.2 Trace File Structure

Figure 5 illustrates the structure of the trace files for a simple toy example. The example has an input-dependent branch in line 10 and a secret-dependent memory access in line 11, which should be detected by our analysis toolchain.

Each trace is structured as follows: The first element defines the type of the trace event, e.g. Call or Expr (for Expression). This is followed by the exact source location of the event, meaning the script file name with start/end line and column number. For a Call, the first location describing the source of the call is followed by a second location describing the target, which in turn is followed by the name of the called function. Expr entries log the locations of all executed expressions; this information is only needed for reconstructing control flow edges. Similarly, Ret1 records the occurrence of a return-statement, which must be tracked due to not being covered by an expression. Ret2 is generated after a function has returned, and records the entire ranges of the function call and the executed function; however, the associated callback does not know where the control flow originated from, thus the necessity of tracking expressions and return statements. The same is true for Cond entries, which mark the execution of a conditional and thus the begin of a control-flow edge.
To illustrate this, Figure 5b shows the case of a taken else-branch with the assignment \( \text{ret} = 0 \) in line 14 of the trace. If we compare this trace to the Figures 5c and 5d, which both show a taken if-branch, it becomes apparent that the control-flow deviation only shows up due to the differences in lines 14 and 15; everything else is identical. Thus, only tracking all expressions and read/write operations allows us to reconstruct the entire control flow.

Comparing line 14 of Figures 5c and 5d demonstrates how the traces enable us to discover secret-dependent memory accesses. The last two elements of the Get entry represent the ID of the shadow object, and the accessed property or offset. Both elements differ between the traces: The object IDs are assigned by Jalangi2 and thus vary for subsequent invocations of processTestcase, and the accessed offset depends on the input. The analysis conducted by Microwalk-CI will later match the object IDs belonging to identical objects between traces, such that it can compare the offsets.

This example shows a very short excerpt of a trace for a toy program. Analyzing real-world code may result in traces with millions of events, resulting in huge files. To reduce the storage overhead, we compress the trace by shortening strings and encoding repeats with the assignment

```
Microwalk-CI
```

These raw traces are not yet suitable for use by arbitrary JavaScript software with the existing and new generic analysis programs, generates raw execution traces and converts them into a format suitable for use by actual memory addresses and offsets. In fact, this proves valuable for later analysis, as this simplifies encoding objects between traces, such that it can compare the offsets.

### 4.3 Trace Preprocessing

These raw traces are not yet suitable for use by Microwalk-CI; we need to translate the sequence of executed lines to branch entries, generate allocation information for the objects showing up in the traces, and finally produce compatible binary traces, which can be fed into analysis modules like the one described in Section 3. For this, we implemented a new preprocessor module, which has 702 lines of code and resides in a plugin. The module iterates through each entry of the raw trace, generating a preprocessed trace on-the-fly. It recognizes branches by waiting for the next code location that is outside the corresponding conditional; if an access to a previously unknown object is detected, an allocation is created.

Note that our analysis module is designed for binary analysis, i.e., it works with actual memory addresses and offsets. In fact, this proves valuable for later analysis, as this simplifies encoding trace entries and gives clear identifiers for referring to certain instructions. Thus, we chose to generate a mapping of observed source locations to dummy addresses, by encoding the line and column numbers onto a base address belonging to the respective source file. This mapping is stored in a special map file, such that it can be mapped back to a human-readable source line after analysis.

In summary, we now have a tool chain that instruments JavaScript programs, generates raw execution traces and converts them into the Microwalk-CI binary trace format, allowing us to analyze arbitrary JavaScript software with the existing and new generic analysis algorithms, without having to create a dedicated analysis tool.

### 5 INTEGRATION INTO DEVELOPMENT WORKFLOW

In this section, we show how one can simplify usage of Microwalk-CI to a degree that it only needs a one-time effort by the developer to set it up and register the functions that need to be analyzed. From that point, the tool is part of the CI pipeline of the respective library, and runs each time a new commit is submitted. The developer is then able to easily verify whether a code change introduces new leakages, without requiring any manual intervention.

#### 5.1 Dockerizing the Analysis Framework

In order to use the analysis framework in an automated environment, we must ensure that all its dependencies are present and the environment is configured correctly. For this task, common CI systems allow the use of Docker containers. When a job starts, a new container is started from a predefined Docker image. The CI system checks the current source code and then executes a user-defined script within the container. This has the advantage of being independent of the host system: The analysis job can run on the developer’s private server, but also on cloud infrastructure administrated by external providers. We thus create a pre-configured Docker image
The individual microwalk/target-*.js

The image is uploaded to a Docker registry. Microwalk-CI
Jalangi2 runtime, the analysis script and the developer if they wish to use other analysis modules or options care of by the existing scripts. We implemented a similar template generating a number of random test cases; everything else is taken focus on supplying simple wrappers for their library interface and than the preconfigured ones.

The template features a script file index.js, which serves as analysis entry point and is responsible for loading test cases and executing the target implementations. A target is any independently testable code unit, e.g., a single primitive in a cryptographic library. The individual microwalk/target-*.js script files consist of a single function, that receives the current test case data buffer and is expected to call the associated library code. Each target also needs a number of test cases, which may have a custom format and thus need to be generated once by the developer. The test cases are stored in the microwalk/testcases/ subdirectory. Finally, the microwalk folder has a bash script analyze.sh, that is called by the CI. The analysis script iterates through the target files, and runs the Microwalk-CI pipeline. The Microwalk-CI configuration is located in two generic YAML files, which can be adjusted by the developer if they wish to use other analysis modules or options than the preconfigured ones.

The abstractions offered by our template allows the developer to focus on supplying simple wrappers for their library interface and generating a number of random test cases; everything else is taken care of by the existing scripts. We implemented a similar template for compiled software, so the approach is the same for C libraries.

Figure 6: Generic source tree of a JavaScript project containing our analysis template.
themselves are executed through a Docker-based GitLab Runner on a separate machine (build server), which has an AMD EPYC 7763 processor with 128 GB DDR4 RAM. We configured the Microwalk-CI trace preprocessor step to use up to 4 CPU cores. After all CI jobs had completed, we collected the performance statistics generated by GitLab and the CI jobs, and went through the leakage reports. The results are visualized in Table 1.

6.2 Performance

6.2.1 Computation time. The CPU time spent for trace generation, preprocessing and analysis mostly depends on two factors. First, it correlates with the complexity of the analyzed targets: For the investigated libraries, symmetric algorithms and utility functions performed very well, while asymmetric primitives took significantly longer, which is expected. Second, the CPU time scales linearly with the number of test cases. We discuss the corresponding trade-off between accuracy and performance in Section 6.4.

The computational cost for the trace generation step mainly stems from the instrumentation itself, as our tracer script is already quite minimal. Significant optimizations would thus need to target the Jalangi2 implementation.

The CPU time spent for the preprocessing step correlates with the size of the raw traces. The implementation is parallelized, so each trace can be processed independently. Profiling shows a slight bottleneck in the string parsing code, so switching to a binary trace format may further improve preprocessing performance, at the cost of higher code complexity in the trace generation.

The analysis step took less than one CPU minute for every investigated target; this underlines the efficiency of the presented analysis algorithm, and that it is fast enough to be used in a productive setting. The time spent for the analysis mostly depends on the trace size, as when building the call tree, each trace entry is converted into a tree node or embedded into an existing one. Another factor is the number of leakages, as is apparent when comparing the analysis times of the various ed25519 implementations.

The measured overall duration heavily depends on where most CPU time is spent: While the trace generation and the analysis are mostly sequential, the trace preprocessing is heavily parallelized. Thus, a high CPU time for preprocessing does contribute less to the overall duration. Apart from one outlier, elliptic’s p384, the measured times stayed well within a few minutes, which can be considered acceptable for productive use in a CI pipeline.

6.2.2 Memory usage. The inherently different pipeline steps also reflect in different memory requirements.

The trace generation step has a negligible memory footprint, which mostly depends on the size of the array that is used for buffering trace entries before writing them to the output file.

The memory consumption of the preprocessing step is mainly caused by loading chunks of the trace file into memory and decompressing them. Parallelization of the preprocessing step means that several trace files are being held in memory simultaneously. The memory usage of the preprocessing can be reduced by decreasing the number of parallel threads (4 in our experiment).

In the call tree analysis step, the memory demand is driven by the size of the preprocessed traces and, most notably, their level of divergence. If the target is constant-time and thus all traces are identical, the tree does not have any split nodes, so all traces end up in the same nodes. Adding a trace ID to an existing node does not involve any significant memory cost, as the trace IDs assigned to a call tree node are stored as a bitfield.

However, if the traces heavily diverge, the analysis produces many split nodes with partially redundant subtrees. This distinction becomes apparent by the implementations of ed25519 in elliptic and in tweetnacl. While using comparable tracing and preprocessing time, the constant-time implementation in tweetnacl requires much less memory than the implementation in elliptic, which relies on the leaking bn.js and hash.js libraries. Through continuously applying Microwalk-CI and mitigating non-constant-time behavior such that only small leakages pop up during analysis, the peak memory usage of the analysis step can be kept within the bounds of a typical CI environment.

Overall, the peak memory usage of Microwalk-CI is on an acceptable level. The highest memory consumption was observed when analyzing elliptic’s p384. This is certainly a worst case example, as large parts of its code are non-constant time, while Microwalk-CI is optimized for finding mid-level leakages in an otherwise fairly constant-time software. However, most of p384’s code is shared with the other curve implementations, which contain the same leakages, but can be analyzed more efficiently. Also, a significant part of the identified leakages reside in the the SHA-512 implementation of hash.js, which should be analyzed separately.

As expected, more complex algorithms like asymmetric cryptography require more memory in the analysis. But, even those only require an amount of memory which, today, is commonly available.

6.3 Vulnerabilities

Our leakage analysis identified many leakages in the given libraries. We evaluated whether those are in fact actual vulnerabilities, and discuss a few examples in the following. In general, the leakages were correctly assigned to the respective leaking code lines, and we did not encounter any false positives (i.e., code lines that don’t leak by themselves). In addition to the report shown in the user interface (Figure 7), a detailed leakage report is generated, which provides the full calling context for each leakage and shows how the different test cases contributed to tree divergences.

6.3.1 Leakages in AES. All investigated implementations of AES use table lookups into S-boxes or precomputed T-tables, making those highly susceptible to timing attacks. The exploitability of such lookups was previously shown in other work [10]. All leakages found in aes–js by Microwalk-CI have a maximum leakage score.

Additionally, Microwalk-CI discovers input-dependent behavior in the AES-GCM encryption of node-forge. Manual inspection shows that these leakages in the tableMultiply function in the file cipherModes.js occur during the computation of the GHASH which is used for the final computation of the authentication tag. The tableMultiply function uses a table precomputed from the hash key and multiplies by accessing this table with an index which is an intermediate value computed from the current ciphertext block and the previous hash value. Learning this intermediate value potentially allows to gain information about the GHASH key, compromising the authentication property. The implementation in node–forge uses 4-bit tables. Whether this implementation
Table 1: Targets analyzed with JavaScript Microwalk-CI, performance metrics and the number of detected leakages (total and unique code lines). “Tr. CPU” shows the CPU time for generating the raw traces. "Prep. CPU" for trace preprocessing, and “An. CPU” for the analysis step. "Duration" denotes the wall clock time spent for the entire CI job (including setup and cleanup). Finally, “Prep. RAM” and "An. RAM" show the peak memory usage for the preprocessing and analysis steps, respectively.

| Target     | Type        | Tr. CPU | Prep. CPU | An. CPU | Duration | Prep. RAM | An. RAM | # Leakages | # Unique |
|------------|-------------|---------|-----------|---------|----------|-----------|---------|------------|----------|
| aes-js [3] | 3.1.2, ≈ 800k weekly downloads |          |           |         |          |           |         |            |          |
| AES-ECB    | cipher      | 1 sec   | < 1 sec   | < 1 sec | 7 sec    | 294 MB   | 180 MB  | 16         | 16       |
| base64-js  | [7] 1.5.1, ≈ 28M weekly downloads |          |           |         |          |           |         |            |          |
| base64-encode | utility     | < 1 sec | < 1 sec   | < 1 sec | 6 sec    | 283 MB   | 173 MB  | 7          | 7        |
| base64-decode | utility     | < 1 sec | < 1 sec   | < 1 sec | 6 sec    | 291 MB   | 189 MB  | 7          | 7        |
| crypto-js  | [15] 4.1.1, ≈ 4M weekly downloads |          |           |         |          |           |         |            |          |
| AES-ECB    | cipher      | 2 sec   | < 1 sec   | < 1 sec | 8 sec    | 289 MB   | 191 MB  | 44         | 44       |
| Rabbit     | cipher      | 2 sec   | < 1 sec   | < 1 sec | 8 sec    | 304 MB   | 182 MB  | 0          | 0        |
| base64-encode | utility     | 4 sec   | 1 sec    | 1 sec   | 10 sec   | 349 MB   | 250 MB  | 0          | 0        |
| base64-decode | utility     | 3 sec   | < 1 sec   | < 1 sec | 9 sec    | 339 MB   | 225 MB  | 2          | 2        |
| pbkdf2     | utility     | 5 sec   | 1 sec    | 1 sec   | 11 sec   | 384 MB   | 221 MB  | 0          | 0        |
| elliptic   | [18] 6.5.4, ≈ 13M weekly downloads |          |           |         |          |           |         |            |          |
| secp256k1  | signature   | 110 sec | 26 sec   | 21 sec  | 139 sec  | 853 MB   | 3,123 MB | 58         | 45       |
| p192       | signature   | 237 sec | 35 sec   | 12 sec  | 261 sec  | 2,112 MB | 1,835 MB | 98         | 57       |
| p224       | signature   | 303 sec | 45 sec   | 14 sec  | 334 sec  | 1,700 MB | 2,357 MB | 76         | 58       |
| p256       | signature   | 469 sec | 84 sec   | 45 sec  | 545 sec  | 3,347 MB | 6,674 MB | 78         | 50       |
| p384       | signature   | 977 sec | 145 sec  | 45 sec  | 1,063 sec| 3,383 MB | 7,522 MB | 391        | 53       |
| ed25519    | signature   | 175 sec | 46 sec   | 32 sec  | 222 sec  | 2,884 MB | 4,607 MB | 111        | 40       |
| js-base64  | [8] 3.7.2, ≈ 6M weekly downloads |          |           |         |          |           |         |            |          |
| base64-encode | utility     | < 1 sec | < 1 sec   | < 1 sec | 6 sec    | 290 MB   | 187 MB  | 0          | 0        |
| base64-decode | utility     | < 1 sec | < 1 sec   | < 1 sec | 6 sec    | 290 MB   | 155 MB  | 0          | 0        |
| node-forge | [19] 1.2.1, ≈ 17M weekly downloads |          |           |         |          |           |         |            |          |
| AES-ECB    | cipher      | 5 sec   | < 1 sec   | < 1 sec | 11 sec   | 298 MB   | 193 MB  | 36         | 36       |
| AES-GCM    | cipher      | 9 sec   | 2 sec    | 2 sec   | 16 sec   | 387 MB   | 349 MB  | 126        | 52       |
| base64-encode | utility     | 5 sec   | < 1 sec   | < 1 sec | 11 sec   | 287 MB   | 192 MB  | 0          | 0        |
| base64-decode | utility     | 5 sec   | < 1 sec   | < 1 sec | 11 sec   | 296 MB   | 198 MB  | 4          | 4        |
| rsa        | signature   | 62 sec  | 18 sec   | 13 sec  | 82 sec   | 364 MB   | 1,926 MB | 223        | 111      |
| ed25519    | signature   | 124 sec | 33 sec   | 9 sec   | 144 sec  | 1,145 MB | 509 MB  | 0          | 0        |
| pbkdf2     | [43] 3.1.2, ≈ 13M weekly downloads |          |           |         |          |           |         |            |          |
| pbkdf2     | utility     | < 1 sec | < 1 sec   | < 1 sec | 6 sec    | 298 MB   | 179 MB  | 0          | 0        |
| tweetnacl  | [58] 1.0.3, ≈ 21M weekly downloads |          |           |         |          |           |         |            |          |
| secretbox  | cipher      | 2 sec   | < 1 sec   | < 1 sec | 8 sec    | 288 MB   | 189 MB  | 0          | 0        |
| box        | asymmetric  | 75 sec  | 20 sec   | 7 sec   | 91 sec   | 335 MB   | 487 MB  | 0          | 0        |
| ed25519    | signature   | 117 sec | 33 sec   | 9 sec   | 138 sec  | 1,137 MB | 509 MB  | 0          | 0        |

and leakage is exploitable, is left to future work. We recommend not having any secret-dependent non-constant-time code.

6.3.2 Elliptic curve implementations. node-forge and tweetnacl feature custom constant-time big number arithmetic that is specifically designed for the supported curves. The elliptic library, however, relies entirely on arithmetic from the general-purpose bn.js [11] library, which features a lot of input-dependent control flow and memory accesses. Thus, we see very high leakage over all supported primitives. The leakages detected in the big number and elliptic code itself are mostly assigned scores between 80 and 100.

In addition, for computing the signature, elliptic’s ECDSA implementation uses the hash.js [24] library, which offers pure-JavaScript implementations for SHA-1 and SHA-2. For ECDSA and EdDSA signatures with the curves p384 and ed25519, respectively, the leakage report points to a significant amount of leakage in lib/hash/sha/512.js for a variety of call stacks. Here, the implementation works around a limitation of JavaScript, which represents all numbers in IEEE-754 double precision floating point, and temporarily converts them to 32-bit signed integers for bitwise arithmetic. If the most-significant bit ends up being 1, JavaScript sign-extends it such that the result is negative. The implementation checks for this in an if statement and adds 0x100000000 to get a positive number. This leakage may pose a security issue, as ECDSA and EdDSA use the hash function for generating a nonce from the private key. Microwalk-CI assigns leakage scores between 60 and 70 for most of the leakages in lib/hash/sha/512.js. Future work could investigate whether the leakage of the most-significant bit can be used to learn parts of the private key. The libraries elliptic, bn.js and hash.js are from the same author.
6.3.3 Base64 encoding. We also found leakages in some of the various Base64 implementations. All of them were caused by the use of lookup tables, where 6-bit chunks are mapped to ASCII characters and vice versa. The only known attack against Base64 encoding relies on a precise controlled channel that is not available for common JavaScript deployments [54]. However, depending on the memory layout of the respective lookup tables, partial information may be accessible via a cache attack. js-base64 does also feature a vulnerable Base64 implementation; however, it first checks whether the Buffer class with native Base64 support is present, which is the case for our Node.js build.

6.4 Number of Test Cases

As mentioned in the performance analysis, computation time and, to a lesser degree, memory consumption, scale with the number of test cases. A higher number of test cases increases the chance of triggering uncommon code paths and thus finding more leakages. In the following, we analyze this trade-off and point out approaches for striking a good balance between accuracy and performance.

In our performance analysis, we ran 16 test cases for each library. This number is within the same order of magnitude as the one used for the evaluation in [62], where the authors recommend running 10 test cases. To check whether the small number of test cases had impact on the number of detected leakages, we repeated our analysis with 48 additional test cases (64 total) for each target and compared the results with those of the first analysis.

6.4.1 Performance. Increasing the number of test cases does not affect every pipeline step in the same way. Doubling the number of test cases roughly doubles the CPU time needed for trace generation, but that does not apply to the analysis step: There, the first test case takes much longer than subsequent ones, as it needs to build the tree from scratch, which involves spending a lot of time in the memory allocator. Later non-diverging test cases only need to iterate the existing tree, which takes considerably less resources. We observed that the duration increased by factor 3 to 5.5, although we ran 4 times as many test cases.

6.4.2 Leaks. Except for targets in the libraries elliptic and node-forge, Microwalk-CI found the same amount of leakages with 64 test cases as with 16. For elliptic, all targets show a small single digit increase in the number of overall and unique leakages. For all new leakages, we determined that these were initially missed due to a saturation effect (see Section 6.6) and not by lack of coverage, and would have been found by re-running the analysis after fixing the preceding leakages.

For node-forge’s RSA implementation, the difference is a bit larger. While Microwalk-CI finds 223 overall and 111 unique leakages with 16 test cases, it was able to discover 255 overall and 125 unique leakages with 64 test cases. Manual investigation shows again that most leakages were missed due to a saturation effect. However, a small number was missed due to insufficient coverage of the initial 16 test cases.

6.4.3 Recommendations. We recommend the developer to choose an overall duration that is acceptable during ongoing development and determine an according test case number. In addition, the coverage of the generated test cases could be checked with a separate tool to ensure that all relevant code gets executed. Finally, the developer could add another larger collection of test cases that runs as a final check before releasing the next version, where a longer analysis time is acceptable.

6.5 Comparison with Microwalk’s original Analysis Module

Microwalk originally features two analysis modules that implement the memory access trace (MAT) analysis method for finding leakages. The method was first presented in [63]. For each memory accessing instruction, the modules generate a hash over all accessed offsets. By comparing the hashes between traces, the amount of leakage for each memory accessing instruction is computed. Due to the focus on memory accesses, control flow leakages are only discovered indirectly or may even be missed entirely.

The first module, that was originally published with [63], generates only one leakage report per instruction. The later added second module (referred to by us as CMAT module) is an extension of the first module that additionally distinguishes between call stacks to achieve a higher accuracy. To compare the existing analysis method with our new approach, we ran a selection of the targets with the CMAT module, using the same 16 test cases as for the initial analysis. The results are shown in Table 2.

Since the CMAT module only stores a single mapping of call stacks and instructions to hashes, it generally takes less resources than our new tree-based approach, both in computation time and memory consumption. However, the preceding trace generation and preprocessing, which take most of the time, are identical, so the actual difference in overall duration is limited.

For aes-js’ AES-ECB implementation, the CMAT module reports a number of secret-dependent table accesses with full leakage, which are identical to the leakages reported by our new analysis module. This is the kind of leakage that the MAT analysis was designed for: Through hashing the sequence of memory addresses that a given instruction accesses, secret-dependent variations are discovered. Our new analysis detects these leakages through the address lists stored in the individual memory access trace entries, which ultimately yields the same result, but takes more memory.

The result from the CMAT module for elliptic’s p192 is very imprecise and contains many false positives: It reports 811 leaking lines in total, which includes lines like “this.pendingTotal = 0;”. As a fixed offset is accessed, this line is a clear false positive. The

| Target   | CPU Duration | RAM  | # Lkgs. | # Unique |
|----------|--------------|------|---------|----------|
| aes-js   | < 1 sec      | 168 MB | 16      | 16       |
| elliptic | p192         | 253 sec | 289 MB | 4,003    | 811     |
|          | ed25519      | 126 sec | 286 MB | 0        | 0       |

Table 2: Results of the analysis step of selected targets with the original Microwalk CMAT module, and its resource usage. Time and memory consumption of the trace generation and preprocessing steps are identical to those shown in Table 1.
leakage in question was in fact caused by a control flow variation higher up in the call chain, leading to a varying number of executions of the given instruction, which in turn produced a different memory access offset hash. The other false positives follow a similar pattern. Our new tree-based approach handles control flow and memory access leakages separately, which reduces false positives and allows accurately attributing a leakage to a specific code line.

6.6 Limitations of the Analysis Algorithm
As other dynamic analysis approaches, Microwalk-CI needs a good coverage of the program in order to give an accurate leakage detection result. If a particular path is never executed, it does not appear in the traces and thus never reaches the analysis modules. However, for cryptographic code, randomly generated test cases tend to work very well [62, 63]. For other targets, it may be worth exploring other methods for generating coverage, e.g., fuzzing.

Finally, in our analysis algorithm, some leakages may be obscured by other leakages at a higher tree level. If leakages on higher levels cause splits that result in a unique sub tree for each trace, the lower leakages cannot cause any more divergences and thus are overlooked. This “saturation” is an inherent property of the analysis approach, and the price paid for having a linear-time algorithm. We do not believe that this impacts practical usage: After having a library reach a certain state of “constant-time-ness”, we only expect few new leakages being reported, as certain functions are touched. And even if a leakage is not reported in a first pass, it will show up after committing the fixes for the previously reported leakages. It is unlikely that a number of unfixed low-severity leakages obscure a subsequent severe leakage. This would imply a fully split up tree, which, in itself, signals a high-severity leakage.

Other work tries to find all trace leakages in a single pass, but uses significantly more resources with every CI run and thus is not suitable for integration into an everyday-development workflow.

7 RELATED WORK
Constant-time program analysis has a long tradition as there are different classes of vulnerabilities that can be found through various analysis techniques [36]. Some tools for checking constant-time behavior depend on the availability of source code. Irazoqui et al. [28] introduce secret-dependent cache trace analysis, ct-fuzz [25] specializes fuzzing for timing leakages. ct-verif [5] describes constant-time through safety properties and CaSym [13] uses symbolic execution to model the execution behavior of a program. Microwalk-CI does not require access to the source code for compiled languages.

Unlike Microwalk-CI which uses dynamic program analysis and compares real execution traces, static binary analysis tries to simulate the execution of every possible program path. BINSEC/REL [16] uses relational symbolic execution of two execution traces to efficiently analyze binary code, however is limited by the high performance impact of static analysis. CacheS [59], based on CacheD [60], combines taint tracking and symbolic execution to find cache line granular leakage and secret-dependent branches. Moreover, CacheAudit [17] tracks relational information about memory blocks to compute upper bounds for leakages. In contrast with these works, Microwalk-CI finds any leakage with byte granularity.

DATA [62] and its (EC)DSA-specific extension [61] find microarchitectural and timing side-channels in binaries via dynamic binary analysis. The trace alignment approach of DATA is based on computing pairwise differences between traces, leading to a computation time that is quadratic both in the number of traces and in the trace length. While it yields more leakage candidates after a single pass, it needs more computational resources and thus is not a suited for use in a CI environment. Abacus [6] identifies secret-dependent memory access instructions using symbolic execution. Then, the authors use Monte Carlo sampling to estimate the amount of leaked information. A shortcoming of the approach is that Abacus only uses one trace and therefore suffers from low coverage. dudct [46] measures timing behavior in a statistical way without any model of the underlying hardware, which is fast, but also yields imprecise results. ctgrind [33] and TIMECOP [39] search the code for secret-dependent jump or memory accesses like table-lookups and variable-time CPU instructions, but are rather manual.

Analysis of JavaScript code recently received more focus in the research community as it is widely used in browsers including many security-critical workloads. Basic properties of JavaScript regarding security of code have been widely analyzed [30, 31, 52, 57]. Just as in other programming languages, various attacks on secret-dependent behavior have been conducted [51, 53]. A common prerequisite for exploiting timing-dependent properties of code is having precise timers [47], though this can be bypassed [53]. Apart from countermeasures like disabling timers or blocking certain functionality [50], little work has gone into finding non-constant-time JavaScript code.

8 CONCLUSION
With Microwalk-CI we have shown how one can design a side-channel analysis framework that is suitable for integration into a day-to-day development workflow. We have presented a new trace processing algorithm that merges the recorded traces into a call tree, allowing us to precisely localize and quantify leakages in a short time frame. Moreover, by “dockerizing” the analysis, we have provided the means for easy and fast usage without the necessity of understanding the details of the framework.

With the design and implementation of a tracer for JavaScript and the integration with Microwalk-CI, we have built the first comprehensive constant-time verifier for JavaScript code and demonstrated how analysis techniques originally developed for binary analysis can be used for interpreted or just-in-time compiled languages. Microwalk-CI is constructed in a modular fashion and allows to add tracing backends for other languages with limited effort.

Overall, Microwalk-CI carries the potential to increase the side-channel security for many popular libraries written in potentially any programming language, and raises awareness for the risks of non-constant-time code in new communities.

ACKNOWLEDGMENTS
The authors thank Julia Tönnies for her help in evaluating the suitability of the leakage metrics, and the anonymous reviewers for their helpful comments and suggestions. This work has been supported by Deutsche Forschungsgemeinschaft (DFG) under grants 427774779 and 4399797619, and by Bundesministerium für Bildung und Forschung (BMBF) through projects ENCOPIA and PeT-HMR.
