Semi-Automated and Easily Interpretable Side-Channel Analysis for Modern JavaScript

Iliana Fayolle[1][0009−0000−6690−529X]*, Jan Wichelmann[2][0000−0002−5748−5462]*, Anja Köhl[2], and Walter Rudametkin[3][0000−0003−2903−7600], Thomas Eisenbarth[2][0000−0003−1116−6973] and Clémentine Maurice[1][0000−0002−8896−9494]

[1] Univ. Lille, CNRS, Inria
[2] University of Lübeck
[3] Univ. Rennes, CNRS, Inria

Abstract. Over the years, developers have become increasingly reliant on web technologies to build their applications, raising concerns about side-channel attacks, especially on cryptographic libraries. Despite the efforts of researchers to ensure constant-time security by proposing tools and methods to find vulnerabilities, challenges remain due to inadequate tools and integration issues in development processes.

We tackle the main limitations of state-of-the-art detection tools. While Microwalk is the first and, to the best of our knowledge, only tool to find side-channel vulnerabilities in JavaScript libraries, the instrumentation framework it relies on does not support modern JavaScript features. Moreover, and common to most state-of-the-art detection tools not aimed at JavaScript, writing tests is a tedious process due to the complexity of libraries, the lack of information about test coverage, and the rudimentary interpretability of the report. Furthermore, recent studies show that developers do not use these tools due to compatibility issues, poor usability, and a lack of integration into workflows.

We extend Microwalk in several directions. First, we design a generic AST-level tracing technique that is tailored to source-based dynamic side-channel leakage analysis, providing support for the latest language features. Second, we bring semi-automation to Microwalk analysis templates, considerably reducing the manual effort necessary to integrate side-channel analyses into development workflows. Third, we are the first to combine leakage reporting with coverage visualization. We evaluate the new toolchain on a set of cryptographic libraries and show that it can quickly and comprehensively uncover more vulnerabilities while writing tests with half as many lines of code as the previous Microwalk version. By open sourcing our new tracer and analysis template, we hope to increase the adoption of automated side-channel leakage analyses in cryptographic library development.

Keywords: Side channels · Vulnerabilities · Cryptography · Automated detection · Instrumentation · Constant Time

*These authors contributed equally to this work.

1 Introduction

Many developers are turning to web technologies to build their applications [4,5,6]. The security of web applications is of great concern, even more so when cryptographic libraries are involved. Side-channel attacks are a popular approach to attack such libraries, potentially rendering every application that uses the library vulnerable. One way to ensure that cryptographic libraries are secure against side-channel attacks is to make them constant-time (i.e., with no dependencies on branches and no memory accesses or allocations). However, ensuring that a library is constant-time can be a challenging and time-consuming task. For this reason, the research community has proposed numerous methods and tools [9,13,15,17,33] to check their programs. Nevertheless, according to Jancar et al. [22], cryptographic library developers do not seem to use these tools because they have poor usability, are not well documented, have compatibility problems, and are not integrated into development processes. More recently, Fourné et al. [16] conducted a user study that revealed that the majority of these tools have similar usability issues that prevent their effective use.

Microwalk-CI [34], proposed in 2022 as an extension of Microwalk [33], is a tool that automates side-channel analyses during Continuous Integration (CI). Microwalk-CI or, for short, Microwalk, combines dynamic analysis with statistical methods to locate and quantify side-channel leaks. It can be used on binary code like many other tools, but is the first to also support JavaScript. Developers must write at least one target file and associated test case files, each using one or more cryptographic primitives from the analyzed library, and the secret data of those primitives. When developers run Microwalk with these test files, the tool generates a radix tree from the execution traces by adding each execution trace to the tree, where each branch represents a divergence that could potentially be exploited by a side-channel attack. Finally, it returns a report in JSON that shows the type, the location, and a score for each leak.

Challenges. We have identified several issues with the current Microwalk tool, which are common to most other state-of-the-art detection tools [16,22]. We present and address them as follows:

1. *Some modern JavaScript libraries cannot be analyzed because of framework incompatibilities.* Indeed, the Jalangi2 framework [31], which Microwalk currently relies on, does not support post-2015 versions of ECMAScript. This means that Microwalk has a compatibility problem as it is unable to parse libraries that use modern JavaScript features such as scoped variables or asynchronous programming. We designed and implemented a new trace generator for JavaScript source code tailored to side-channel analysis, thus improving the compatibility of Microwalk.
2. *Writing tests is tedious and error-prone.* If the library to be analyzed is large and complex, the process of writing tests can become tedious due to their quantity and sophistication. This can encourage copy-pasting, which can lead to errors within the tests and can be a problem if the library is modified. We

introduce a novel semi-automated leakage analysis template that minimizes the number of lines a developer has to write in order to test their library.
3. *Developers cannot know whether their tests are sufficient to uncover all vulnerabilities.* The use of Microwalk's JSON report requires some effort and time from the user, since it provides only limited information about the location and quantification of each leak per file. Indeed, due to the source code not being included in the report, users must navigate between their code and the report to understand which statements in their program are not constant-time, which goes against the recommendations by Fourné et al. [16] for output generation. Furthermore, it can be tedious to identify all the leaks in a library if the tests do not cover the entire source code. We design a new output that improves the interpretability of the report by integrating the vulnerability reports with the code, and adding test coverage. This helps developers choose which tests to perform.

Contributions.

- We design a generic AST-level tracing technique for source-based dynamic side-channel leakage analysis, which can be easily adjusted to support new language features (Section 4);
- We extend the Microwalk analysis template with support for subtargets and test case generation, bringing semi-automation to the process (Section 5);
- We are the first, to the best of our knowledge, to combine leakage reporting with coverage visualization in a user-friendly way (Section 6);
- We use the new toolchain to comprehensively analyze a number of JavaScript cryptographic libraries. We demonstrate its efficiency and show that it can uncover vulnerabilities in modern ECMAScript code (Section 7).

While we chose to extend Microwalk rather than to build yet-another-tool, most of our contributions are generic and can be applied to other tools. The source code of our work is available in anonymous repositories: `https://github.com/EasyWalk-CI-project/EasyWalk` (trace generation); `https://github.com/EasyWalk-CI-project/EasyWalk-Evaluation` (new template and evaluation). Our work will gradually be added to the Microwalk main branch.

2 Background

2.1 Side-Channel Leakage

Microarchitectural side-channel attacks are based on observable information from microarchitectural components, which an attacker can measure and compare, resulting in the disclosure of secrets. Vulnerabilities to these attacks are located on the boundary between hardware and software. At the hardware level, optimizations and shared resources allow the attacker to interfere with their victim. For example, caches greatly speed up accessing frequently used memory. However, if the attacker resides on the same core (or processor), they can purposefully evict the victim's data to learn when it is accessed [30]. In recent years, many

such attacks have been published, ranging from coarse-grained attacks via the translation look-aside buffer [19] to subtle intra cache line leakages [28].

All these attacks exploit the *secret-dependent runtime behavior* of an application, which are varying memory access patterns or control flow. If, for example, the victim's application does a table lookup with a key-dependent index, the attacker can evict the table from the cache and subsequently measure which part of the table is immediately fetched back into the cache. This way, they learn which part of the table was accessed by the victim, and can correlate many such observations to extract the secret input.

The primary solution to prevent such attacks is to write a program without secret-dependent control flow and memory accesses, such that the program always exhibits a runtime behavior (statistically) independent from the secret. This approach is called *constant-time* programming.

2.2 Instrumentation

Instrumentation is the automated modification of an existing program in a transparent manner. Applications are *tracing* (i.e., gathering information about a program's behavior), and *modification* (i.e., injecting changes during execution). There are two main flavors of instrumentation: *static instrumentation* changes the program permanently so the injected code gadgets are executed every time the program is invoked; and *dynamic instrumentation*, which extends the program at runtime, usually by executing the program through a special dynamic instrumentation engine that takes ownership of the control flow and uses just-in-time compiling techniques to inject the requested code gadgets.

A common characteristic of any instrumentation framework is the deployment of a callback-based API, which allows the user to provide arbitrary code that is executed in a specific situation (e.g., a library is loaded, or on a memory access). For JavaScript, there are a few instrumentation frameworks. Aran [23] and Jalangi2 [31] both directly rewrite the JavaScript source code, as it is an interpreted language with no native binary representation. They load user-provided plugins for each event. However, neither support all features of modern JavaScript, with Jalangi2 not having received notable updates since 2017. Another noteworthy mention is OpenTelemetry[3], which supports various programming languages but is tailored to high-level tracing like function calls or web APIs.

2.3 Microwalk

Microwalk [33] is a micro-architectural leakage detection framework that combines a dynamic analysis approach with statistical methods to quickly locate and quantify side-channel leaks. It can be used on binary software and JavaScript libraries, and can be run in a Continuous Integration (CI) workflow [34].

The Microwalk framework [25] requires four steps for a developer to analyze a cryptographic library. (i) The developer chooses an analysis template, which are currently provided for C and JavaScript libraries on GitLab, GitHub, and locally. They copy the template into the root of their library repository and merge or

adjust the CI configuration and package files. (ii) The developer provides at least one *target*, that calls a library function (i.e., a cryptographic primitive) to analyze. (iii) They manually create a number of test case files, i.e., secret inputs, which are read by the associated target. These test cases need to be as random as possible to ensure a good analysis by Microwalk. The number of required test cases depends on the analyzed primitive, however, Wichelmann et al. [34] recommend 16 test cases as a good compromise between performance and the number of vulnerabilities found. (iv) The developer pushes a new commit (triggering the CI workflow) or runs the Microwalk analysis script locally.

The analysis script traverses all of the targets and their associated test cases, generates execution traces, and merges those into a radix tree, where each function call and trace entry forms the nodes, and where subsequent function calls and trace divergences form the branches. Those trace divergences are potential leaks that can be exploited by side-channel attacks, as this means that the attacker can learn something about the test case (a secret) by looking at the trace. After inserting the traces, the tree gets traversed to evaluate, for each statement in each call stack, whether it caused a leak. To quantify leakage severity, numerous metrics are computed, like the minimal conditional guessing entropy [34], which measures the smallest number of guesses needed for an attacker to map a given trace from the tree to a secret input. At the end of the analysis, the tool produces three artifacts: a control flow leakage analysis report, a code quality report for each target test, as well as a merged code quality report that can be displayed by GitLab and GitHub.

3 Overview

We first give an overview of the limitations of the state-of-the-art for efficiently finding side-channel vulnerabilities in cryptographic libraries and we outline our goals to address those limitations. Finally, we describe the improved development and leakage analysis workflow.

3.1 Goals

We identified three issues that specifically hinder the adoption of Microwalk for JavaScript analysis, but that also need to be addressed by side-channel leakage analysis tools in general. We address those limitations with the following three goals: (1) compatibility of the trace generator, (2) semi-automation for (sub)target and test case generation, and (3) interpretability of the vulnerability report by the developer.

Compatibility. JavaScript is actively developed. Every few years a new version of the ECMAScript (ES) specification is released, offering new features and improvements. However, the Jalangi2 framework [31] that Microwalk currently relies on does not support ES2015 (also called ES6) and beyond, which includes essential functionality like scoped variables or asynchronous programming. As

there are no comparable instrumentation frameworks, Microwalk is not able to analyze libraries that use modern JavaScript features. Additionally, the evaluation in [34] showed that trace generation takes the bulk of the leakage analysis time. A large part of this overhead is caused by numerous unused callbacks in Jalangi2. To address these issues, we build an AST-based tracing engine that does not need an external instrumentation framework, and thus can be easily adjusted to support new language features. In addition, we can directly extract the information needed for leakage analysis, reducing trace size and avoiding an error-prone trace preprocessing steps.

Semi-automation. The process of analyzing a library with a dynamic leakage analysis tool usually has some mandatory requirements: the developer needs to implement a wrapper for invoking the respective functionality (called *target*), and specify a set of secret inputs for testing (called *test cases*). However, complex libraries that offer a lot of primitives and flexibility require writing a large amount of targets, leading to extensive implementation and maintenance efforts. We noticed that this resulted in a lot of duplicate code, which is prone to copy/paste errors. The developer also has to generate lots of static test cases, which increase the size of the code repository. While the need to implement targets cannot be fully averted, we extend the template in a way that minimizes the amount of code to write by providing native support for *subtargets*, which combine several flavors of a given primitive type in a single target implementation. In addition, we move away from statically provided test cases to target-specific automated test case generation based on fixed seeds. This way, test cases are generated deterministically and on-the-fly during leakage analysis.

Report interpretability. In the free version of GitLab, developers do not get the pipeline details view of the code quality report, but rather a JSON report, forcing them to navigate between the report and their code. As a result, it can be cumbersome to match a vulnerability with the corresponding line of code. To address this, we create an alternative report that integrates each reported vulnerability in the library code, so that developers know exactly where a vulnerability has been found. Additionally, dynamic analysis tools may, by design, not cover all the lines of a program, especially as cryptographic libraries are increasingly complex. To address this, the report provides coverage information, highlighting which lines were not covered by the tests. Overall, this increases the interpretability of the report for developers who can subsequently take concrete actions to fix their libraries.

3.2 Workflow

Given our goals, we pursue the following development workflow (see Figure 1). The library developer starts by copying the analysis template into their repository and configures it to integrate into their CI workflow. They then implement a set of targets and associated test case generators. Each target relates to a certain primitive (e.g., AES) and consists of several subtargets, which invoke

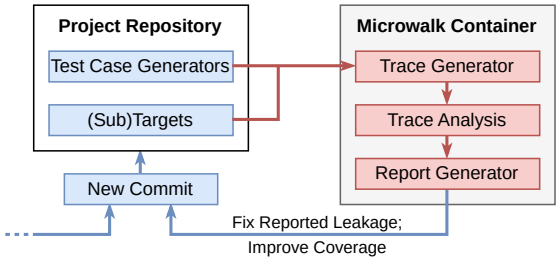

Fig. 1. Development workflow overview. The developer's tasks are blue, automated tasks are red. The developer implements (sub)targets and corresponding test case generators for analyzing the library. On each commit, the CI generates and analyzes execution traces, and produces a report that outlines leakages and code coverage. The developer uses the report to modify the code to fix leakages or to add more test cases to increase coverage.

the respective parts of the library. The subtargets share the same test case generator, but may tweak it through custom attributes (e.g., key length). After the setup, each commit triggers the CI workflow, starting a Microwalk container that executes every subtarget and analyzes the resulting execution traces. The vulnerability and code coverage analysis produce a human-readable report. The developer can verify if their commit introduced new leakages and whether the analysis missed parts of the library. To improve coverage, they may subsequently increase the number of test cases or introduce further subtargets.

4 AST-based Trace Generation for Leakage Analysis

To address our *compatibility* goal, we designed a new trace generator for JavaScript source code that is tailored to side-channel vulnerability analysis. This section has three parts. First, we assess the runtime information that needs to be collected to enable leakage analysis. Then, we specify the technical capabilities the new tracer should have. Finally, we describe how we implemented an instrumentation engine that satisfies these requirements by inserting trace generation logic into JavaScript source code.

4.1 Collected Information

The Microwalk trace format was originally designed for binary analysis [33]. However, as shown in [34], it can be adapted to source-based analysis by running the raw execution traces through a corresponding preprocessor. The generic traces that Microwalk processes contain three groups of entries: branches, memory accesses, and heap (de-)allocations.

To discover secret-dependent control flows, all **branches** must be collected. This translates to tracking whether `if` statements execute their body, which `case`

in a `switch` statement is hit (including fallthroughs) and whether a loop body executes or is terminated early. Calls and returns must be tracked in order to allow call-stack sensitive leakage analysis. ES2015 comes with additional complexity in form of deferred execution with `yield` and asynchronous code with `async`/`await`.

To find secret-dependent memory accesses, the target addresses of all **memory accesses** are needed. In source-based tracing, we do not have actual memory addresses, only arrays and objects. For the former, we record the accessed indexes; for the latter, the accessed keys. The trace preprocessor then generates artificial and unique memory addresses for each index and key.

JavaScript does not have explicit **heap allocations**. Arrays and objects are constructed when initialized and garbage collected when out-of-scope. Our traces assign each object a unique ID, and include that ID in all logged memory accesses. If an ID appears for the first time, the preprocessor generates an artificial heap allocation entry. As these IDs are never shared, we do not have to generate corresponding heap frees.

4.2 Technical Capabilities

The new tracer should be capable of handling modern JavaScript, which means supporting the most currently available standard (ES2023 at the time of writing), but also be backwards compatible to handle many package dependencies that are written in older versions of JavaScript and are no longer maintained. Modern JavaScript includes features like ES modules (ESM), asynchronous programming, deferred execution, and classes. Neither of these are supported by Jalangi2 or comparable instrumentation tools (e.g., Aran [23]). In addition, the execution traces should not only cover the analyzed library itself, but also its dependencies. Node.js packages are notorious for their deep dependency trees [24], so leakages in seemingly innocuous utility functions from external sources should be found as well. This requires recursive scanning of all module imports.

4.3 Implementation

As there is currently no general-purpose instrumentation framework for JavaScript that satisfies our requirements, we skip the abstraction and directly insert the trace calls into the library's code itself. For that, we use Babel [10]. Babel was originally conceived as a *transpiler* that converts post-ES2015 code to older JavaScript versions, allowing code written with modern syntax to be executed with older interpreters. The framework is well-maintained and supports all current language versions. It is also well suited for our application as it offers a rich API for parsing, modifying and emitting JavaScript code. At its core, Babel represents a JavaScript program as an abstract syntax tree (AST). We built a custom plugin for Babel that transforms the AST and enriches it with tracing code. The trace generation pipeline is illustrated in Figure 2.

In principle, we take a similar approach as Jalangi2. The instrumentation engine expects a single argument, the main target file. The file is parsed into its AST, which is traversed with a number of node visitors. Each visitor is dedicated

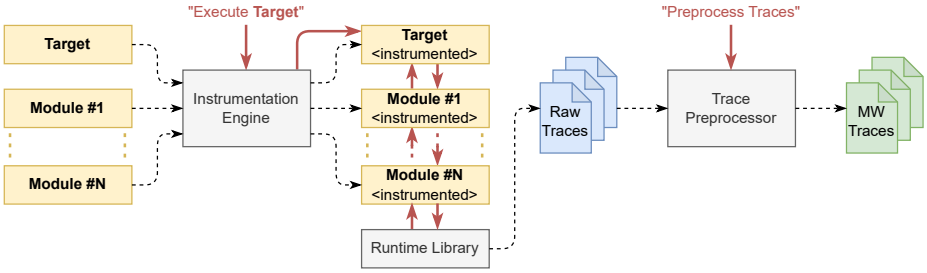

Fig. 2. Trace generation pipeline. The red arrows denote execution flow. When the instrumentation engine is invoked on a target, it recursively traverses imported modules and produces instrumented copies. After instrumentation is done, execution is handed over to the translated target. Every relevant location of the instrumented code contains calls to the runtime library, which emits corresponding raw execution traces. These raw execution traces are preprocessed into the Microwalk trace format.

to a specific node type and applies necessary transformations and inserts calls to the runtime library. Contrary to Jalangi2, which wraps every operation into callbacks, we directly insert logging calls into relevant operations. These are, for example, call and return statements in order to catch function entry and exit, the bodies of block statements to detect branches, and accesses to arrays and objects to trace memory accesses. To handle complex expressions, like nested or chained calls and accesses (e.g., `a().b(c()).d().e[4]`), we split them and store intermediate results in temporary variables. This way we avoid accidentally executing a function twice, which would make our instrumentation incorrect.

Jalangi2 has no understanding of control flow statements and loops, but instead only emits *expressions* (e.g., for `i++`) and *conditions* (e.g., for `i < 5`). This simplifies the instrumentation, as the framework does not deal with many special cases, like `else if` or `switch`, but complicates collecting the branch information necessary for side-channel analysis. Control flow tracking has to be fully implemented in the analysis callbacks and the old preprocessor, which used complex heuristics to determine when a branch trace entry should be emitted. The new tracer directly emits branch entries at the begin of control flow statements.

A notable ES2015 feature with impact on trace generation is deferred execution with `yield` (also called *generators*). It dilutes the otherwise clear concept of call/return, as execution jumps right back into a function after briefly leaving it. Since the Microwalk analysis modules expect clean call trees, we emit return/call trace entries when handling `yield` statements. While we could use Babel's transpilation capabilities to replace those post-ES2015 features with older syntax, this would break the mapping between the leakage report and the source code. Thus, we handle each feature manually.

Another issue are dependencies. ES2015 introduces a new dependency system that relies on asynchronous `import` statements, replacing the synchronous CommonJS `require()` functionality. These are incompatible, i.e., a CommonJS-based

library cannot load an ES module. To support both, we implement all runtime components as synchronous CommonJS modules, which is unfortunate as using synchronous I/O slows down trace generation. We will revisit this decision in the future when adoption of ES modules has increased. We recursively instrument all static dependencies that are known at instrumentation time, and install a handler for on-the-fly instrumentation of dynamic imports. Each instrumented module is cached so we only process them once.

To keep the instrumentation simple, we moved all trace writing code into a separate runtime module that is loaded by all instrumented modules. Similar to Jalangi2, the runtime contains callbacks for every type of trace entry. However, the callbacks are very thin and just emit the trace entries, as the domain-specific instrumentation already yields all necessary information.

5 A Semi-Automated Leakage Analysis Template

Contrary to other leakage analysis tools which are used mostly manually and ad-hoc, Microwalk aims for a high level of automation. However, the current infrastructure still requires lots of manual work and risks mistakes and increased maintenance effort. We thus analyzed what is the *minimal* amount of manual work a developer has to invest when integrating side-channel analysis into their workflow, and designed a new generic analysis template supporting that. The template only needs few adjustments to fit a new library. Our core improvements to the template are twofold: we introduce the concept of *subtargets*, and we fully automate test case generation. With these, in the best case, supporting a new variation of a cryptographic algorithm (e.g., key length) boils down to adding a single configuration value. While we implemented the new template for JavaScript analysis, it can be easily translated to other programming languages and works for both source- and binary-level analysis.

5.1 Subtargets

When the developer wants to analyze a primitive with Microwalk, they have to implement a *target file*, which consists of a test case processing function. While simple in principle, they need to create such a file for every variation of a primitive that is analyzed (e.g., AES-128-ECB, AES-256-ECB, AES-128-GCM), which either leads to a high amount of code duplication or to simply skipping these variations. To reuse code while staying compatible to Microwalk's existing workflows, we introduce subtargets. Subtargets are defined through a configuration object in the target file, which specifies the number of requested test cases, pointers to test case generation and execution functions, and other custom parameters.

When a target is selected for analysis, the controller script first queries its subtargets. For each subtarget, the script checks for a test case generation function and invokes it (see Section 5.2). Finally, to start the trace generation pipeline, the subtarget's test case execution function is called through the instrumentation

engine. We pass the current subtarget's configuration object to each of the functions. This way, the developer can check which subtarget is executed and potentially extract additional custom parameters. Figure 3 shows an excerpt from the `aes-js` target we used for the evaluation. To analyze a new variation of an existing primitive like AES-CBC, we would only need to add three lines of code, which is the corresponding `case` in the `switch` statement and the subtarget definition in `targetInfo`. Adding a primitive that has no similarity to existing ones is less efficient and generally indicates that a new target file is warranted to keep the code readable. Nevertheless, for illustration, we defined the `aes-js` hex conversion subtarget `utilHex` in the same target file anyway.

5.2 Test Case Generation

Microwalk offers two avenues for providing test cases: the first is to manually generate a set of static test cases, which are included in the repository. However, test cases are often binary files, which may not belong in a code repository and would bloat its size. Second, the Microwalk configuration format allows specifying a shell command for generating test cases. However, this was designed for the Pin-based tracer and is incompatible to JavaScript analysis, as the Microwalk analysis pipeline only runs after trace generation has completed. Additionally, it is highly inflexible, as the developer would need to write a configuration file for every target. In fact, the library that is analyzed most likely already provides avenues to generate suitable inputs, so it would make sense to enable the developer to use those for dynamic test case generation.

We move the entire test case generation logic into the analysis target template, extending our subtarget configuration described earlier, and avoid static test cases altogether. Besides the test case execution function, the configuration object now also specifies a test case generation callback and optional parameters. As a result, the developer may now either specify their own test case generation function or reference a standard test case generation function provided by the template. The analysis controller script calls the selected test case generator before executing each subtarget. This way, the test case generation is automated as much as possible. To ensure that the analysis results are reproducible, the test case generator is provided with a deterministic seed, so it can yield the same test cases on each execution.

6 Leakage Report with Coverage Information

To achieve our *report interpretability* goal, we have devised an HTML report generator that combines the leakage analysis results with the code coverage, given by c8 coverage reports, and embeds this information directly in the source code of the libraries where vulnerabilities were found. This feature increases the usefulness of the leakage report and helps developers find defects in the test suite.

```
import * as microwalk from './microwalk.mjs';
import aesjs from 'aes-js';

function processCipherTestcase(testcaseBuffer, subtarget) {
    const message = new Uint8Array(32);
    const iv = new Uint8Array(16);
    const key = new Uint8Array(testcaseBuffer);
    let aes;
    switch(subtarget) {
        case targetInfo.subtargets.ecb:
            aes = new aesjs.ModeOfOperation.ecb(key); break;
        //case targetInfo.subtargets.cbc:
        //    aes = new aesjs.ModeOfOperation.cbc(key, iv); break;
        default: throw new Error(`Unknown subtarget ${subtarget}`);
    }
    let encryptedBytes = aes.encrypt(message);
}

function processUtilTestcase(testcaseBuffer, subtarget) {
    const inputStr = testcaseBuffer.toString("utf-8");
    const data = aesjs.utils.hex.toBytes(inputStr);
    const newStr = aesjs.utils.hex.fromBytes(data);
}

export const targetInfo = {
  generate: microwalk.generateRandomBytes, // default generator callback
  generateOptions: { length: 16 },         // default options
  process: processCipherTestcase,          // default execution callback
  subtargets: {
    ecb: { testcaseCount: 16 },
    //cbc: { testcaseCount: 16 },
    utilHex: {
      testcaseCount: 16,
      process: processUtilTestcase,
      generate: (...args) => microwalk.generateRandomBytes(...args)
                                       .toString("hex"), // custom generator
      generateOptions: { length: 64 }
    }
  }
};
```

Fig. 3. Excerpt from the `aes-js` analysis target. The developer only needs to supply short wrappers for calling the library. The `targetInfo` object is parsed by the analysis controller and holds all necessary information for test case generation and execution.

6.1 Report Generation

To improve the existing JSON report which summarizes all the leakages found in the library so far, we combine the leakage information with the library's code, and the code coverage of the test suite, to generate a new report in a format that displays more information appropriately.

In the new report, in HTML format, we highlight where leakages are detected and we include an information box describing the leak. The box includes the type of leak (i.e., *secret-dependent branch*, *secret-dependent memory access*) and the severity of the leak (i.e., *minor*, *major*, *critical*). Due to the absence of memory allocation commands, there are no memory allocation leaks in JavaScript, so there are only two possible leakages types. This report also shows the code coverage of

the targeted test, highlighting code that was not executed. We explain how we compute coverage in Section 6.2.

The header summarizes the most important details of the leakages and the code coverage report, such as the total number of leaks found in all files, the percentage of characters covered, the percentage of lines covered, as well as the number and percentage of critical leaks and memory access leaks in the current file. We chose to display information only on the memory access leaks because if the percentage of the memory access type of leaks is known, the percentage of branch type of leaks is deducible.

Microwalk may detect several leaks for the same line of code, which happens if a line of code appears in multiple contexts during the analysis. This is due to the call tree-based analysis algorithm, where the same divergences can occur in different parts of the radix tree, each with its own leakage score. Thus, some leaking lines of code are counted multiple times in the "total leakages". The "unique leakages" score counts the number of unique leaking code lines, regardless of how many times they appear in the report. With this in mind, if a line contains the same leak many times, sometimes with different associated leakage scores, the report displays only the details of the leak with the highest leakage score.

6.2 Computing Coverage

To compute code coverage, we use the c8 library [11] and modify the analysis controller script to execute the code coverage JSON report generator alongside the leakage analysis. Then, we build the HTML report combining the library source code, detected vulnerabilities and measured code coverage. To properly display the code coverage on the source code, our program first sorts the coverage per file then compares the uncovered character intervals for each file and target test. We use four counters for the comparison, one for the first character of the line, one for the length of the line, and two for the start and end of each uncovered character interval. These intervals can occur in four possible positions within a line: (i) The entire line is uncovered. (ii) Only the beginning of the line is uncovered. (iii) Only the end of the line is uncovered. (iv) Both the beginning and the end of the line are covered, but a part in the middle of the line is not. However, if a line has more than one interval, looping over multiple intervals is sufficient to identify which parts of the line are covered. Once we've identified all of the intervals in the file that are not covered, the HTML report highlights all of the characters that fit into those intervals and marks the line numbers associated with them. Appendix A shows a screenshot from part of an HTML report.

7 Evaluation

We evaluate our enhancements by showing how they allow a developer to fully analyze four selected cryptographic libraries with minimal effort. The results of our evaluation are summarized in Table 1 and Table 2.

7.1 Experimental setup

To show the capabilities of the new tracer and template, we perform a comprehensive analysis of a few popular Node.js cryptographic libraries. We chose `aes-js` [8] and `node-forge` [2] as these libraries are known to leak and we can compare our results to existing work [34]. Additionally, we analyze `@noble/curves` [26] and `@noble/ciphers` [27], two libraries written in modern JavaScript that provide clean implementations of common cryptographic primitives, with a clear focus on security. The `@noble` libraries are incompatible with Jalangi2 and thus could not be analyzed by Microwalk until now.

We wrote an analysis target for each library, using subtargets to call the respective primitives. For the old Microwalk workflow, we implemented equivalent targets per library and generated static test cases. The analysis repositories are hosted on a local GitLab instance, and automatically invoke the preconfigured CI pipeline when a new commit is pushed. The server the CI jobs are executed on has a Intel Xeon Gold 6438Y+ CPU and 500 GB memory. However, we restrict the jobs to 4 processor cores and 64 GB memory.

Metrics. We consider several metrics. To understand the performance of the new tracer, we look at the time and memory needed for generating and preprocessing the execution traces. In addition, we check its correctness by comparing the detected leakages to the results from [34]. The same code line may show up in multiple contexts, e.g., a leaking utility function. This is due to the call tree-based analysis algorithm, where callees may occur in different subtrees, each with their own leakage score. Thus, some leaking code lines are counted multiple times. The "# Unique" column counts the number of unique leaking code lines, ignoring how often they appear in the leakage report. To assess the efficiency of the new analysis template, we count and compare the files and lines of code needed to be written by the developer to achieve full analysis of their library.

7.2 Results

Performance. To understand the performance characteristics of our new JavaScript tracer, we measure the CPU time of trace generation, and the CPU time and memory consumption of trace preprocessing. We observe that our instrumentation is slower than Jalangi2's, but the performance of the instrumented code is better, despite collecting larger traces. As instrumentation is only done once for the entire library, its impact on the overall analysis duration diminishes when adding more targets. In addition, our tracer is a work-in-progress that, unlike Jalangi2 which is mature, has not been optimized.

Memory consumption never exceeds 2 GB, in most cases staying even well below 1 GB, which is well within the limits of common CI runners. We also measured the memory consumption of the tracer and the analysis module, but those were always below that of the preprocessor. The total analysis duration of the four libraries with the new pipeline was 9 minutes and 7 seconds, which is arguably acceptable for use in everyday programming. The reasons for the

Table 1. Comparison between Microwalk V3.2.0 and Microwalk using the enhanced analysis toolchain on `aes-js` [8] and `node-forge` [2] libraries. Results are averaged over 10 runs. **Bold** numbers are the results from the enhanced Microwalk tool in case they change. # F represents the number of files created for tests and test cases.

Subtarget	Trace CPU	Prep. CPU	Prep. RAM	# Leakages	# Unique
`aes-js` [8], v3.1.2, statement coverage: **80%**					
(*instrument*	2/**2** sec)				
aes-128-ecb	2/<**1** sec	< 1 sec	208/**209** MB	24/**32**	24/**29**
aes-192-ecb	2/<**1** sec	<1 sec	207/**213** MB	24/**32**	24/**29**
aes-256-ecb	2/<**1** sec	<1 sec	213/**215** MB	28/**36**	28/**33**
aes-128-ctr	2/<**1** sec	<1 sec	208/**216** MB	24/**31**	16/**20**
aes-128-cbc	2/<**1** sec	<1 sec	210/**212** MB	24/**30**	24/**28**
aes-128-cfb	2/<**1** sec	<1 sec	211/**210** MB	24/**29**	16/**19**
aes-128-ofb	2/<**1** sec	<1 sec	207/**212** MB	24/**29**	16/**19**
hex-convert	2/<**1** sec	<1 sec	166/**192** MB	0/**1**	0/**1**
all tests	16/**8** sec	8 sec	| Target size: 288/**95** lines | # F: 136/**1**		
`node-forge` [2], v1.3.1, statement coverage: **45%**					
(*instrument*	1/**54** sec)				
aes-128-ecb	5/<**1** sec	< 1 sec	201/**205** MB	36	36
aes-192-gcm	8/**2** sec	3/**4** sec	246/**245** MB	126/**113**	52/**43**
decode	6/<**1** sec	<1 sec	189/**197** MB	4	4
encode	5/<**1** sec	<1 sec	185/**199** MB	0	0
ed25519-sign	95/**60** sec	43/**41** sec	664/**1,282** MB	0	0
all tests	119/**65** sec	49/**48** sec	| Target size: 176/**111** lines | # F: 85/**1**		

improved runtime performance are twofold. First, our instrumentation is tailored to the use case (i.e., we only trace events which are relevant for leakage analysis). Second, we inline as much as possible. Jalangi2's plugin approach allows for greater flexibility but comes with the cost of more layers of indirection and unnecessary callbacks. We directly log branches, so the preprocessor does not need to reconstruct them from logged expressions, which considerably reduces the size of the raw traces and thus the footprint of the preprocessor itself. These reductions in turn allow us to trace *more* information relevant to leakage analysis without negatively affecting performance. In summary, our new tracer performs better than the legacy Jalangi2 backend in most aspects, while supporting modern post-ES2015 JavaScript.

Development effort and coverage. We wrote a single target for each library, aiming for conciseness and maintainability, while maximizing coverage of the respective library. As outlined in Section 5, we implemented simple wrapper functions for calling each primitive, and defined a configuration object that points to the respective wrapper function and an associated test case generator. In total, the target files have 491 lines. The targets for `aes-js` and `node-forge` are

Table 2. Results of using the enhanced analysis toolchain with two modern JavaScript cryptographic libraries: `@noble/ciphers` [27] and `@noble/curves` [26]. Results are averaged over 10 runs.

Subtarget	Trace CPU	Prep. CPU	Prep. RAM	# Leakages	# Unique
`@noble/ciphers` [27], v0.5.1, target size: **119 lines**, statement coverage: **93%**					
(*instrument*	7 sec)				
chacha20-poly1305	8 sec	3 sec	450 MB	0	0
xchacha20-poly1305	8 sec	3 sec	455 MB	0	0
xsalsa20-poly1305	7 sec	3 sec	490 MB	0	0
aes-256-ecb	7 sec	3 sec	475 MB	61	25
aes-256-ctr	8 sec	3 sec	465 MB	72	12
aes-256-cbc	8 sec	3 sec	460 MB	61	25
aes-256-siv	9 sec	3 sec	471 MB	147	15
aes-256-gcm	8 sec	3 sec	492 MB	208	14
hex-convert	7 sec	3 sec	447 MB	1	1
`@noble/curves` [26], v1.3.0, target size: **166 lines**, statement coverage: **72%**					
(*instrument*	9 sec)				
secp256k1	37 sec	6 sec	595 MB	169	29
secp256r1	43 sec	7 sec	659 MB	85	27
secp384r1	56 sec	11 sec	817 MB	85	27
ecdh	29 sec	6 sec	507 MB	73	28
schnorr	44 sec	8 sec	607 MB	371	56
ed25519	17 sec	6 sec	632 MB	101	25
x25519	9 sec	4 sec	480 MB	10	2
ed448	13 sec	8 sec	577 MB	83	25
x448	10 sec	4 sec	489 MB	10	2

considerable shorter for the new semi-automated template with 206 vs. 464 lines, while the new template also includes test case generation.

Even with these comparably small target files, we managed to achieve a statement coverage of 55.6% (84.5% when excluding `node-forge`). The remaining code mostly concerns primitives from `node-forge` and, for the other libraries, utility functions (e.g., UTF-8 conversion) that are unlikely to be used with sensitive data, basic input validation, and error checking code. Thus, the single target file with several subtargets streamlines test case generation and maintenance while allowing us to substantially increase coverage.

Vulnerabilities. We validate the findings from [34], who found table lookup leakages in `aes-js`. However, we increased coverage significantly, allowing us to find additional leakages in that library, for example, in the hexadecimal conversion utility function. Our results also highlight that it is worth analyzing *all* variations of a given primitive, as we found that the key expansion for AES-256 takes a slightly different execution path and was thus missed by earlier analyses. Coverage helps identifying such missed code paths. The new toolchain does not report several false positives that are reported by the old workflow

in `node-forge/aes-192-gcm`; we are still investigating the root cause for this behavior. We verified that the new toolchain reports the same (or more) true positives in all cases. In `@noble/ciphers`, our toolchain detects secret-dependent lookup table accesses in the AES and GHASH/POLYVAL implementations. For the elliptic curve implementations in `@noble/curves`, the analysis reports many leakages. However, we found that all curve implementations use blinding, so it is not straightforward to distinguish false-positives that are caused by varying behavior from independent randomness. This is a known shortcoming of Microwalk, which we leave to future work. We have relayed our new findings to the library authors.

8 Discussion and Future Work

Source-based vs. binary analysis. Our improved workflow follows Microwalk's approach in analyzing JavaScript at the *source code level*. However, JavaScript is a scripting language that is just-in-time compiled into machine code. The JIT compiler may insert numerous optimizations and new leakages, which we would not catch in our model. This does not mean that our approach is unsuitable – leakages that are visible at source code level are very likely to also surface after compilation. In fact, we argue that this is the only guarantee a JavaScript developer can possibly get regarding the side-channel security of their application: the JIT compilers are constantly changing, and the code transformations and optimizations depend heavily on the respective workload and even program runtime. Thus, a leakage-free program for a number of tests is not applicable to other systems and use-cases. A potential future research direction to mitigate this issue is the development of a JIT mode that is guaranteed to not insert new side-channels, e.g., by disabling certain optimizations for selected functions. Similar functionality is already available for preventing Spectre attacks [7].

Accessibility of coverage information. Currently, the combined coverage/leakage reports are accessible through the HTML reports provided in the job artifacts, but could be further integrated into developer workflows. For example, GitLab offers built-in support for visualizing coverage in the merge request diff view [18], and many IDEs allow loading and displaying coverage information. Another benefit of integrating with existing tools is that developers could be provided with a comparison to prior coverage and leakage analysis results, actively alerting them when new commits introduce uncovered or leaking code.

Randomized implementations. A remaining weakness of Microwalk are implementations that generate internal randomness (e.g., the ephemeral key of ECDSA) or temporarily mask (i.e., blind) sensitive values during computations. Both lead to variations in the traces, making the analysis module report many leakages. It is tempting to dismiss such findings as false positives, but there may be *partial* leakage despite the randomization, which would then be missed. As a solution, leakage analysis tools should provide a randomization-capable analysis module, as proposed by DATA [32] or CacheQL [35].

Other programming languages. While we implemented our new tracer for JavaScript, the approach is not tied to that language and can be applied to other programming languages. Most modern programming languages have frameworks for converting between source code and the corresponding AST, so the tracing logic can be readily adapted to those languages. The trace format is also very generic and can, alongside the preprocessor, likely be reused without changes.

9 Related Work

9.1 Leakage Analysis

For the past 15 years, developers have witnessed the emergence of numerous analysis strategies and tools to detect side-channel vulnerabilities [17]. These tools fall into two categories: static and dynamic.

Static tools focus more on program verification. Their goal is to infer security properties from the program without executing it. To this end, they have many different approaches. First, the logical reduction approach consists of transforming a program so that verifying its security against side-channel attacks is equivalent to proving the security of the original program. Second, the type systems approach is to verify the type safety of a program. In this case, developers simply need to type the secret values with annotations into their program. The type system then propagates these types throughout the program, as compared to static taint analysis. Third, the abstract interpretation approach addresses the difficulty of formally verifying non-trivial properties due to program semantics. This approach over-approximates its set of reachable states. As a consequence, if the approximation is safe, then the program is safe. CacheAudit [14] exemplifies this approach well, as it performs a binary-level analysis and quantifies the amount of leakage depending on the cache policy by finding the size of the range of a side-channel function. This side-channel function is computed through abstract interpretation, and the size of its range is determined with counting techniques. Finally, the symbolic execution approach verifies the properties of a program by executing it with symbolic inputs instead of concrete ones. Then, logical formulas are built from the conjunction of all conditionals leading to each explored execution path, and a solver is used to check whether a set of concrete values satisfies the generated formulas. For example, the Binsec/Rel [12] framework completes a bounded exploration of reachable states and displays counterexamples for the identified vulnerabilities.

Dynamic tools focus more on finding bugs. Their approaches are based on the security guarantees provided by execution traces of targeted programs. There are two groups of dynamic tools: those that analyze a single trace, and those that compare multiple traces. The first group tends to sacrifice coverage for scalability but can be easy to deploy, as it is for ctgrind [1], which checks for CT by defining a secret as undefined memory and applying the taint analysis of Valgrind. Two main approaches are possible when comparing multiple traces. First, statistical tests can be used to check whether different secrets induce statistically significant differences in the traces. For example, Microwalk [25] uses Mutual information

(MI), which gives a score analogous to a leakage estimation by quantifying the information shared between secret values and recorded traces. On the other hand, CacheQL [35] does not assume a uniform distribution of the secret, nor reformulates deterministic execution traces. To that aim, CacheQL turns MI into conditional probabilities. Second, fuzzing techniques can be used to maximize coverage and side-channel leakage.

9.2 Fuzzing and regression testing

Fuzz testing or fuzzing techniques aim to throw exceptions (e.g., crashes, memory leaks, or failed built-in code assertions) by automatically providing the code with invalid, unexpected, or random data as program input. Some tools use fuzzing as a means to cover more code and thus find more side-channel leaks with less testing, such as DiffFuzz [29] and ct-fuzz [21]. The former uses fuzzing to find side-channel vulnerabilities based on the number of instructions, memory usage, and response size in Java programs. The latter extends this method to binary executables and cache leakage.

Regression testing techniques provide a set of tests performed on a previously tested program to determine whether a bug or malfunction has been added or found in unmodified parts of the software after the program has been modified. In particular, the Triggerflow [20] tool proposes tracking execution paths that dynamically analyze the binary through the debugger using source annotations.

In addition, Jancar et al. [22] made suggestions for developers of tools, compilers, cryptographic libraries, and standardization to make their cryptographic code resistant to timing attacks. Our tool responds to most of their suggestions while offering alternatives or equivalents to the techniques presented above to keep execution time efficient and find a fair number of vulnerabilities.

10 Conclusion

In this paper, we have shown how the usability of side-channel leakage analysis tools can be further improved in order to aid their practical adoption. To support recent JavaScript language features, we have implemented a new trace generation pipeline that injects tracing logic directly into the program's AST. We have developed a new analysis template for JavaScript libraries that aims for conciseness and maintainability, greatly reducing the effort developers need to spend for integrating side-channel analysis into their workflows. Our new report generator combines vulnerability information with detailed coverage data, allowing developers to easily spot untested code paths and suitably adjust the test case set. In our evaluation, we have shown that our new toolchain is capable of comprehensively analyzing modern JavaScript cryptographic libraries, and we have seen that the tracer outperforms the previous Jalangi2-based implementation. The combination of fast analysis with detailed coverage reporting enabled us to uncover new vulnerabilities. In summary, our toolchain helps developers find more vulnerabilities with less effort.

Acknowledgments

This work was supported by the project ANR-21-CE39-0019/Deutsche Forschungsgemeinschaft (DFG) 491039149 FACADES, and by Bundesministerium für Bildung und Forschung (BMBF) through the SAM-Smart project.

References

1. Imperialviolet - checking that functions are constant time with valgrind, https://www.imperialviolet.org/2010/04/01/ctgrind.html
2. node-forge, https://www.npmjs.com/package/node-forge
3. OpenTelemetry. https://opentelemetry.io/, accessed: 2024-02-21
4. Python developers survey 2022 results, https://lp.jetbrains.com/python-developers-survey-2022/
5. Stack overflow developer survey 2022, https://survey.stackoverflow.co/2022/
6. State of JavaScript 2022: Usage, https://2022.stateofjs.com/fr-FR/usage/
7. Untrusted code mitigations. https://v8.dev/docs/untrusted-code-mitigations, accessed: 2024-04-24
8. AES-JS: https://github.com/ricmoo/aes-js, accessed: 2024-02-21
9. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: USENIX Security Symposium (2016)
10. Babel: Babel is a JavaScript compiler. https://babeljs.io/, accessed: 2024-02-21
11. Coe, B.E.: c8 - native v8 code-coverage, https://github.com/bcoe/c8, original-date: 2017-10-26T06:40:38Z
12. Daniel, L., Bardin, S., Rezk, T.: Binsec/rel: Symbolic binary analyzer for security with applications to constant-time and secret-erasure. ACM Trans. Priv. Secur. **26**(2), 11:1–11:42 (2023)
13. Daniel, L.A., Bardin, S., Rezk, T.: Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In: S&P (2020)
14. Doychev, G., Feld, D., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: A tool for the static analysis of cache side channels. In: USENIX Security Symposium (2013)
15. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: CacheAudit: A tool for the static analysis of cache side channels. ACM Trans. Inf. Syst. Security **18** (2015)
16. Fourné, M., Braga, D.D.A., Jancar, J., Sabt, M., Schwabe, P., Barthe, G., Fouque, P.A., Acar, Y.: "these results must be false": A usability evaluation of constant-time analysis tools. In: USENIX Security Symposium (2024)
17. Geimer, A., Vergnolle, M., Recoules, F., Daniel, L.A., Bardin, S., Maurice, C.: A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries. In: CCS (2023)
18. GitLab: Test coverage visualization. https://docs.gitlab.com/ee/ci/testing/test_coverage_visualization.html, accessed: 2024-02-21
19. Gras, B., Razavi, K., Bos, H., Giuffrida, C.: Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In: USENIX Sec. Symp. (2018)
20. Gridin, I., García, C.P., Tuveri, N., Brumley, B.B.: Triggerflow: Regression testing by advanced execution path inspection. In: DIMVA (2019)
21. He, S., Emmi, M., Ciocarlie, G.F.: ct-fuzz: Fuzzing for timing leaks. In: ICST (2020)
22. Jancar, J., Fourné, M., Braga, D.D.A., Sabt, M., Schwabe, P., Barthe, G., Fouque, P.A., Acar, Y.: "they're not that hard to mitigate": What cryptographic library developers think about timing attacks. In: S&P (2022)

23. L. Christophe: Aran. `https://github.com/lachrist/aran`, accessed: 2024-02-21
24. Liran Tal: NPM security: preventing supply chain attacks. `https://snyk.io/blog/npm-security-preventing-supply-chain-attacks/`, accessed: 2024-02-21
25. Microwalk Project: Source code and templates. `https://github.com/microwalk-project`
26. Miller, P.: `https://github.com/paulmillr/noble-curves`, accessed: 2024-02-21
27. Miller, P.: `https://github.com/paulmillr/noble-ciphers`, accessed: 2024-02-21
28. Moghimi, A., Wichelmann, J., Eisenbarth, T., Sunar, B.: Memjam: A false dependency attack against constant-time crypto implementations **47**(4), 538–570
29. Nilizadeh, S., Noller, Y., Pasareanu, C.S.: Diffuzz: differential fuzzing for side-channel analysis. In: ICSE (2019)
30. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: CT-RSA (2006)
31. Samsung: Jalangi2. `https://github.com/Samsung/jalangi2`, accessed: 2024-02-21
32. Weiser, S., Zankl, A., Spreitzer, R., Miller, K., Mangard, S., Sigl, G.: DATA - differential address trace analysis: Finding address-based side-channels in binaries. In: USENIX Security Symposium (2018)
33. Wichelmann, J., Moghimi, A., Eisenbarth, T., Sunar, B.: Microwalk: A framework for finding side channels in binaries. In: ACSAC (2018)
34. Wichelmann, J., Sieck, F., Pätschke, A., Eisenbarth, T.: Microwalk-ci: Practical side-channel analysis for javascript applications. In: CCS (2022)
35. Yuan, Y., Liu, Z., Wang, S.: Cacheql: Quantifying and localizing cache side-channel vulnerabilities in production software. In: USENIX Security Symposium (2023)

A The HTML report

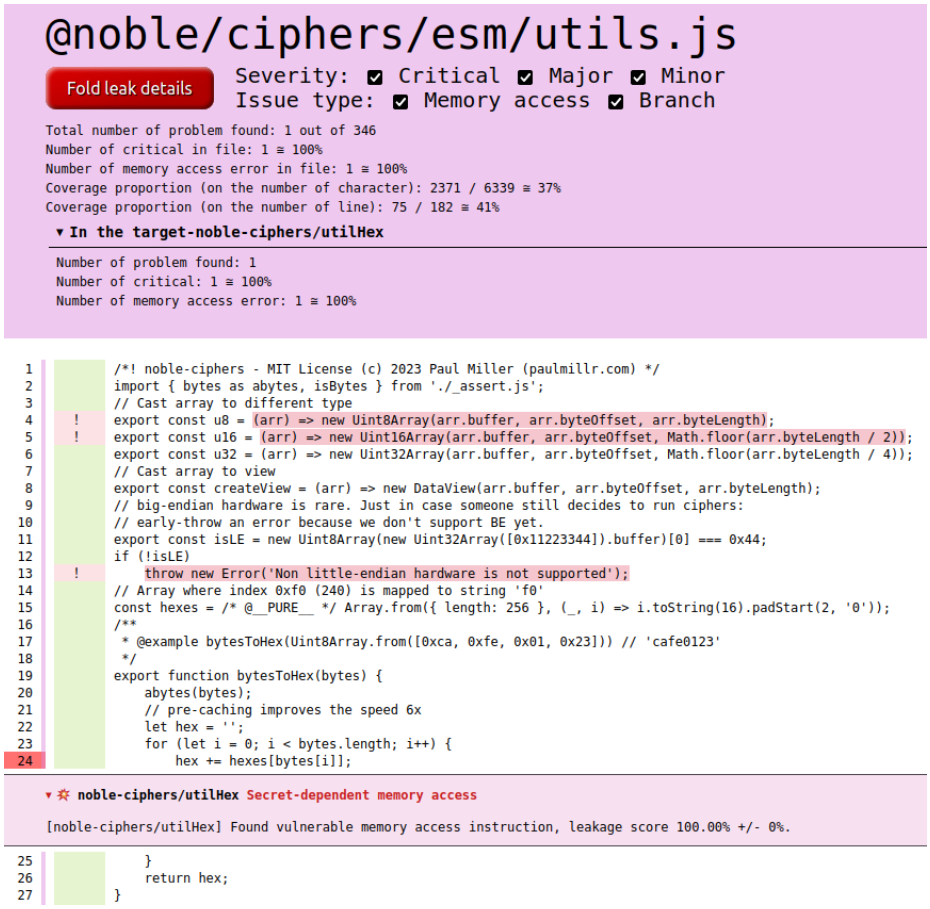

Fig. 4. Extract of the HTML report after analyzing the `@noble/ciphers` [27] library. The report shows the header, a critical secret-dependent memory access leak (line 24), and, highlighted in light red, instructions that were not covered (lines 4, 5 and 13).