MetropolJS: Visualizing and Debugging Large-Scale JavaScript Program Structure with Treemaps

Joshua D Scarsbrook
University of Waikato
jds30@students.waikato.ac.nz

Ryan K L Ko
University of Waikato
ryan.ko@waikato.ac.nz

Bill Rogers
University of Waikato
coms0108@waikato.ac.nz

David Bainbridge
University of Waikato
davidb@waikato.ac.nz

ABSTRACT

As a result of the large scale and diverse composition of modern compiled JavaScript applications, comprehending overall program structure for debugging is challenging. In this paper we present our solution: MetropolJS. By using a Treemap-based visualization it is possible to get a high level view within limited screen real estate. Previous approaches to Treemaps lacked the fine detail and interactive features to be useful as a debugging tool. This paper introduces an optimized approach for visualizing complex program structure that enables new debugging techniques where the execution of programs can be displayed in real time from a bird's-eye view. The approach facilitates highlighting and visualizing method calls and distinctive code patterns on top of code segments without a high overhead for navigation. Using this approach enables fast analysis of previously difficult-to-comprehend code bases.

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; • **Software and its engineering** → *Software prototyping*;

KEYWORDS

JavaScript, Treemaps, Debugging

ACM Reference Format:
Joshua D Scarsbrook, Ryan K L Ko, Bill Rogers, and David Bainbridge. 2018. MetropolJS: Visualizing and Debugging Large-Scale JavaScript Program Structure with Treemaps. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages.
https://doi.org/10.1145/3196321.3196368

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05.
https://doi.org/10.1145/3196321.3196368

1 INTRODUCTION

MetropolJS (pronounced "Metropolis") is a program visualization and comprehension tool designed to help developers visualize static and dynamic aspects of large-scale applications written in JavaScript. This visualization tool is particularly focused on improving the productivity of researchers in Cyber Security. During reverse engineering it is rare to have access to any of the source code. Without it the functionality of the program is hidden in a compiled block with all the forms of documentation removed. MetropolJS helps to reveal this lost structure.

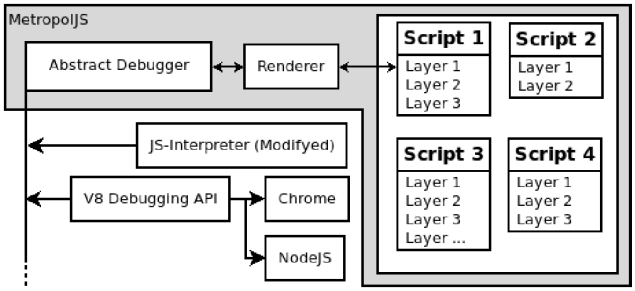
The teaser figure (shown above) is part of a rendering of the Underscore.js[1] test suite generated from MetropolJS data. The many peaks and colors represent different parts of the JavaScript source and information gathered during program execution. In operation MetropolJS is used to view execution of code in real-time, with the data displayed shown in 2D form as seen in Figures 2a–2d (video of the process is available in the source repository[8]).

Figure 1 shows the basic architecture of MetropolJS. The program being debugged is stored as an Abstract Syntax Tree (AST) organized by source script and tree layer. The debugging interface controls the operation of external JavaScript execution environments. Operation can best be explained by considering some use cases.

1.1 Visualizing Program Structure

The big challenge encountering a new codebase is comprehending it, especially when access to the source files is not possible. MetropolJS is designed to provide a high level visualization of tens of thousand line source files. In well maintained software projects the structure

Figure 1: An architectural overview of MetropolJS.



is logically split into a series of packages and classes. In compiled JavaScript the structure is no longer clear. It still exists but the whitespace has been removed and libraries have been collapsed into a single source files. Because of this reading the structure becomes challenging. By reducing the source code to an abstract syntax tree and laying out the resulting structure with a Treemap[11] the obfuscated structure can be clarified.

MetropolJS achieves this by parsing source code and converting the resulting Abstract Syntax Tree into a binary tree for rendering. Given input such as Figure 3a the resulting tree looks like Figure 3b.

Figure 2a shows part of UnderscoreJS with the AST nodes laid out. From this examples of single functions can be seen such as Figure 2a Ⓐ. As shown in Figure 3b a function definition consists of 2 major groups The two (sub) boxes on the left side of the example represent the function arguments and the right side contains the function body.

Figure 2a Ⓑ shows a boundary between 2 scripts or packages made clear by a nested structure resulting code patterns normally used to define modules in JavaScript.

1.2 Dynamic Code Coverage

Figure 2b shows how the code coverage analysis produced by MetropolJS reveals a high-level overview of program execution progress in real time. The connection to the live execution environment has provided the information for continuous updates to the visualization. Figure 2b Ⓐ captures this as the program is executing. The black overview line marks the current call stack and as it moves around the visualization the nodes are steadily highlighted. Figure 2b Ⓑ shows an example of visited source code that has been highlighted in gray.

1.3 Program Analysis for Complex Structure

During a security audit of a code base it is as important to look at what is being called as where it is being called from. As an example overwritten function definitions or unusual patterns of native calls could indicate compromised code. In our prototype this data is represented by highlighting nodes based on feedback from the interpreter. In Figure 2c assignment expressions targeting variables containing functions have been highlighted in blue and an example to the right of a call to a native function is highlighted in red.

This system is structured to encourage further plug-in development by utilizing the observer pattern for the debugger and providing high-level rendering access though a public API. In this example the function overwrite detection was implemented by adding a patch to JSInterpreter[5] to call back to MetropolJS when a variable that previously held a function declaration is being overwritten.

1.4 Scaling up to Production Applications

All the work mentioned so far relates to uses with a securely sandboxed JavaScript runtime called JSInterpreter[5]. While this approach has served well for access to internal runtime state it lacks the access to system services and speed to observe real-world applications run with the full feature set enabled. The solution to this has been to connect to the V8[2] runtime. V8 Provides a debugging API[3] which can be used to collect information and control the runtime of a Chromium based browser or Node.js.

Giving an example. The previous use cases focused on Underscore.js and it's test suite. This is a self-contained library without significant third party requirements. To demonstrate how MetropolJS can scale up, OpenMCT[6] was chosen as a test case. OpenMCT is a mission control dashboard used by NASA. It is comprised of a Node.js server component and a web application written in JavaScript. Using the V8 Inspector API it is possible to connect MetropolJS to both the server side and client side application.

One of the big differentiating factors between small-scale and production applications is the size of the code and how it has been split up. OpenMCT comprised of hundreds of libraries which all need to be loaded to produce a comprehensive viewpoint. Figure 2d shows a snapshot of this in practice. This is a zoomed in snapshot a NodeJS program with MetropolJS connected to the Node.js Server.

This data source is still not feature complete compared to the sandboxed interpreter. It gathers coverage by repeatably querying V8 for statement-level coverage. Compared to the JSInterpreter debugger this does not support node-level coverage or live execution traces. Possible solutions will be discussed later in the Implementation and Future Work sections.

2 PROTOTYPE IMPLEMENTATION

2.1 Tree Layout

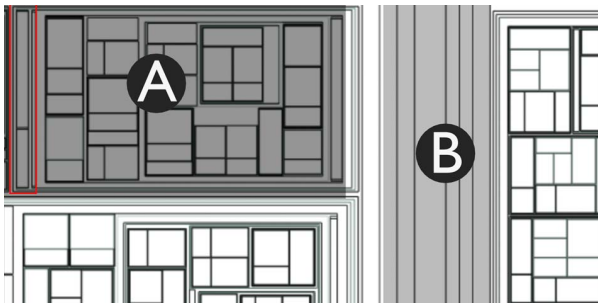
This visualization is implemented by considering each source document an independent entity. The source file is parsed into an AST then the resulting tree is walked to produce the geometry. The geometry generation process is based on the BinaryTree[11] algorithm and works by repeated binary divisions of the tree. Space is allocated for nodes by recursively calculating the total number of nodes in each subtree and dividing the node geometry in proportion to the number of nodes in each subtree. This takes into account internal nodes as well as leaf nodes to prevent deeply nested nodes from shrinking too much.

The format also means node lookups can be performed in $O(\log n)$ time using a binary tree search.

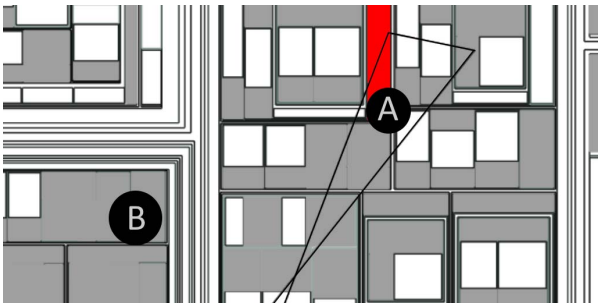
2.2 Application and Renderer

The visualization has been implemented as a web application with rendering built on top of THREE.js[4] and WebGL.

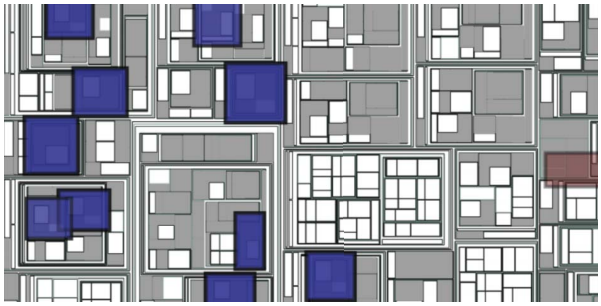
(a) A section of Underscore.js rendered with MetropolJS. (Ⓐ and Ⓑ are overlaid for explanation)



(b) A section from the same program rendered with code coverage information overlaid.



(c) MetropolJS being used to overlay locations of function overwrites and specified native calls.



(d) MetropolJS connected to NodeJS though the V8 Inspector Protocol.

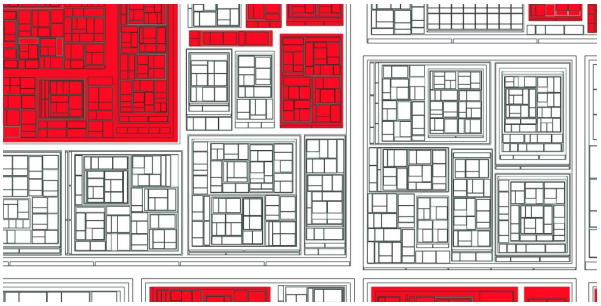


Figure 2: Examples of MetropolJS in use.

(a) Example JavaScript code to be parsed.

```
1  /**
2    Add 2 numbers together
3    @param {number} a
4    @param {number} b
5    @return {number}
6  */
7  function test(a, b) {
8    return a + b;
9  }
```

(b) The resulting AST and sizes used by MetropolJS.

```
1  -FunctionDeclaration     (size=9)
2    -Identifier[test]      (size=1)
3    -Identifier[a]         (size=1)
4    -Identifier[b]         (size=1)
5    -BlockStatement        (size=5)
6      -ReturnStatement     (size=4)
7        -BinaryExpression  (size=3)
8          -Identifier[a]   (size=1)
9          -Identifier[b]   (size=1)
```

Figure 3: An example of JavaScript code being parsed and the resulting AST

The layout produces a binary tree containing location information which is compiled into geometry. This is done with a layered approach to enable deferred loading and improve memory bandwidth efficiency.

The binary tree is rendered by converting it into a series of layers with pointers to facilitate small updates to the geometry such as tinting the color of nodes. The stage can take multiple seconds as larger trees can have over 1,000,000 nodes that need to be rendered. For this reason the rendering process is completed over multiple frames to keep the user interface interactive while rendering occurs. The approach to rendering helps reduce work on the GPU during this process by splitting a very large model into a series of easy to manage layers.

2.3 Scaling the Renderer

Table 1 shows metrics collected during the testing of MetropolJS. Testing was conducted on a desktop system with an Intel Core i5-7600K and an Nvidia Geforce GTX960 with MetropolJS running in Chrome 64.0. Performance was well within acceptable limits for all scripts tested with all of them meeting the 60FPS baseline for high-quality animation. Of particular note is three.js[4] where MetropolJS maintained high performance even while rendering 1.6 million vertices.

Table 1: Scripts Rendered with MetropolJS.

Name	Version	Vertex Count	Layer Count	Lines of Code	Frames per Second
jQuery	3.2.1	371,705	43	3,258	444.44
three.js	r88	1,682,892	36	18,821	462.96
Lodash	4.17.4	336,089	31	3,600	476.19
Vue	2.5.3	368,041	35	3,563	490.20
Polymer	0.5.5	451,435	29	5,286	425.53
Moment	2.19.2	203,554	26	1,752	512.82
Underscore	1.8.3	75,503	25	713	518.13
Underscore+unittests	1.8.3	456,898	26	8,310	497.51

2.4 Debugging and Data Collection

With the dynamic updating capabilities it is possible to augment the display with additional information gathered from script execution. This has been achieved with a live JavaScript runtime modified to extract information about the script being executed.

Debugging with this visualization benefits most from location information in the source code. Since the visualization is directly produced from the Abstract Syntax Tree this is represented as a node in the tree. One issue encountered is that in most JavaScript engines the AST is used after the parse step to generate low level code, but the debugging is conducted using source code locations.

To overcome this a JavaScript Interpreter was modified to extract the information about AST nodes from its current application stack. JSInterpreter executes the AST directly without intermediate byte code so the patch fetched the current stack frame and resolved it to an AST Node.

This approach simplifies demoing MetropolJS but does not provide all the system level APIs necessary to run real world applications. To solve this the V8 debugger API was leveraged to connect to live applications and websites.

3 RELATED WORK

Treemaps are a well developed visualization technique with work dating back to 1991[10]. For example, WinDirStat[9] builds a Treemap of a file system to show file sizes. However existing work in visualization of software structure has focused on the production of off-line analysis tools.

NDepend[7] provides a variety of visualizations one of which uses a Treemap to visualize source code metrics by overlaying code coverage over top a Treemap built from the code structure of a project. This solution has similar goals to MetropolJS in providing a high-level visualization of file and program structure. However it is not connected to a debugger and can not provide real time coverage information as the program executes.

Most similar in appearance to MetropolJS is Code Cities[12]. This technique uses a similar Treemap layout but uses height to highlight complex or incompletely tested classes. In contrast MetropolJS uses a live visualization of program structure to aid comprehension in debugging and reverse engineering.

4 CONCLUSION AND FUTURE WORK

MetropolJS as introduced here is capable of rendering and supervising execution for self-contained applications in a sandboxed environment. Its rendering techniques have also been shown to scale up to real-world applications, shown in Table 1 allowing smooth real-time navigation. As illustrated in Sections 1.2 and 1.3 the system can accumulate and dynamically display code coverage information and interesting code patterns.

Further work is needed to pull more debugging information from V8. The version connected to the V8 debugger can already display coverage but it lacks the deep information to interpret program flow. In the future it may be possible to expand on the existing V8 debugging APIs to support this.

To support future refinement MetropolJS has been released as open source at https://github.com/Waikato/MetropolJS.

ACKNOWLEDGMENTS

The authors wish to thank the members of Cyber Security Researchers of Waikato (CROW Lab) particularly Jeffery Garae and STRATUS (Security Technologies Returning Accountability, Trust and User-Centric Services in the Cloud - https://stratus.org.nz), a science investment project funded by the New Zealand Ministry of Business, Innovation and Employment (MBIE).

REFERENCES

[1] Jeremy Ashkenas and Contributors. 2009. Underscore.js. (2009). http://underscorejs.org/
[2] V8 Authors. 2008. Chrome V8. (2008). https://developers.google.com/v8/
[3] V8 Authors. 2018. Debugging over the V8 Inspector API. (2018). https://github.com/v8/v8/wiki/Debugging-over-the-V8-Inspector-API
[4] Ricardo Cabello and Contributors. 2010. three.js - Javascript 3D library. (2010). https://threejs.org/
[5] Neil Fraser. 2018. JS-Interpreter: A sandboxed JavaScript interpreter in JavaScript. (Feb. 2018). https://github.com/NeilFraser/JS-Interpreter original-date: 2013-10-30T01:00:51Z.
[6] NASA. 2014. Open MCT - Open Source Mission Control Software. (2014). https://nasa.github.io/openmct/
[7] NDepend. 2006. NDepend. (2006). http://ndepend.com/docs/getting-started-with-ndepend
[8] Joshua D Scarsbrook and Contributors. 2018. Waikato/MetropolJS. (2018). https://github.com/Waikato/MetropolJS
[9] Bernhard Seifert and Oliver Schneider. 2003. WinDirStat. (2003). https://windirstat.net/
[10] Ben Shneiderman. 1992. Tree Visualization with Tree-maps: 2-d Space-filling Approach. *ACM Trans. Graph.* 11, 1 (Jan. 1992), 92–99. https://doi.org/10.1145/102377.115768
[11] Ben Shneiderman. 2005. Treemaps for space-constrained visualization of hierarchies. http://www.cs.umd.edu/hcil/treemap-history/
[12] Richard Wettel and Michele Lanza. 2008. CodeCity: 3D Visualization of Large-scale Software. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. ACM, New York, NY, USA, 921–922. https://doi.org/10.1145/1370175.1370188