View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ASU Digital Repository

FrozenNode: Static Linking of Node.js Applications

by

James Keith Hutchins

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2018 by the
Graduate Supervisory Committee:

Adam Doupé, Chair
Yan Shoshitaishvili
Ziming Zhao

ARIZONA STATE UNIVERSITY

May 2018

ABSTRACT

Web applications are ubiquitous. Accessible from almost anywhere, web applications support multiple platforms and can be easily customized. Most people interact with web applications daily for social media, communication, research, purchases, etc. Node.js has gained popularity as a programming language for web applications. A server-side JavaScript implementation, Node.js, allows both the front-end and back-end to be coded in JavaScript. Node.js contains many features such as dynamic inclusion of other modules using a built-in function named require which dynamically locates and loads code.

To be effective, web applications must perform actions quickly while avoiding unexpected interruptions. However, dynamically linked libraries can cause delays and thus downtime, because dynamically linked code must load multiple files, often from disk. As loading is one of the slowest operations a computer performs, seeking from disk can have a negative impact on performance which causes the server to feel less responsive for users. Dynamically linked code can also break when the underlying library is updated. Normally, when trying to update a server, developers will use test servers. However, if the developer accidentally updates a library in a dynamically linked system, it may be incompatible with another portion of the program.

Statically linking code makes it more reliable and faster (to load) than dynamically linking code. The static linking process varies by programming language. Therefore, different static linkers need to be developed for different languages. This thesis describes the creation of a static linker, called FrozenNode, for the popular back-end web application language, Node.js. FrozenNode resolves Node.js applications into a single file that does not rely on dynamic libraries. FrozenNode was built on top of Closure Compiler to accurately

process JavaScript. We found that the resolved application was faster and self-contained yielding significant advantages over the dynamically loaded application. Furthermore, both had the same output.

Vulnerabilities in web applications can be found using static analysis tools, however static analysis tools must reason about dynamically linked application. FrozenNode can be used to statically link a Node.js application before being used by a JavaScript static analysis tool.

DEDICATION

Special thanks to Dr. Adam Doupé, who has been my Thesis Chair, Faculty Advisor, and mentor. Dr. Doupé encouraged me in my education and career and wrote letters of recommendation for both the 4+1 Computer Science program and for my internship. My first class with Dr. Doupé was CSE 340 Principles of Programming Languages. I learned a lot in that class and Dr. Doupé encouraged me to enroll in his CSE 545 Software Security class. A highlight of that course was the opportunity to work together with a small group of talented students in a "Capture the Flag" competition. I really enjoyed developing strategies to defend and attack the vulnerabilities using C executables, while a teammate created scripts in Python that could automatically sign into the game and exploit vulnerabilities. Our team placed 3rd in the competition, in a class of 170 students. Dr. Doupé's CSE 545 Software Security was my favorite class at ASU!

From that class, I knew I wanted to work with Dr. Doupé. He was very supportive of my career goals and provided valuable career and education advice. Most importantly, Dr. Doupé shared brilliant ideas and suggestions on how to move forward when I felt stymied by this process.

Thanks to Dr. Yan Shoshitaishvili for quick and helpful responses to questions. I appreciate the support of Dr. Shoshitaishvili and Dr. Ziming Zhao as Committee Members.

I also appreciate the support of the Laboratory of Security Engineering for Future Computing (SEFCOM) and my SEFCOM lab mates in sharing their research and suggestions for resources to get past stumbling blocks in my research.

I’d also like to thank Christina Sebring, my Academic Success Specialist. She has been extremely helpful in assisting me to complete the School of Computing, Informatics, and Decision Systems Engineering (CIDSE) Master of Science requirements.

Thanks to my Mom, Dad, and Brother for supporting me throughout this process.

ACKNOWLEDGMENTS

This work has been supported by the National Science Foundation (NSF) CyberCorps® Scholarship For Service (SFS) program, NSF Award DGE-1129561, and the Information Assurance Center at Arizona State University (ASU), Stephen S. Yau, Director. ASU is certified as a National Center of Academic Excellence in Information Assurance Education (CAE/IAE) and a National Center of Academic Excellence in Information Assurance Research (CAE-R) by the National Security Agency and the Department of Homeland Security.

This work builds upon work completed by Jonathan Wasserman in his undergraduate Honor's Thesis: TSCAN: Toward Static and Customizable Analysis for Node.js where he identified several vulnerable sources and sinks for Node.js.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

INTRODUCTION 1

BACKGROUND 4

Static Linker 4

JavaScript 5

Compilers for JavaScript 6

Node.js 7

Node.js Require Statements 8

DESIGN 10

Require Resolution 10

Handling New Modules 11

Cached Modules 13

Variable Prepending 14

IMPLEMENTATION 14

EVALUATION 16

Robustness 16

Testing Methodology 17

Results 18

DISCUSSION 27

Limitations 27

Future Work 29

RELATED WORK 32

CONCLUSION 35

REFERENCES 36

LIST OF TABLES

Table Page

Table 1: Number of Unique Modules by Required Internal Module 19

Table 2: Number of Unique Modules by Library from NPM 20

Table 3: Largest Proportional Difference in Runtimes for the Library Test Cases 20

Table 4: Smallest Proportional Difference in Runtimes for the Library Test Cases........ 20

Table 5: Average Runtime for Eval/Process Test Using 10 Require Statements 22

Table 6: Number of Unique Modules by Real World App 26

Table 7: Largest Proportional Difference in Runtimes for the Real World Apps............ 26

Table 8: Smallest Proportional Difference in Runtimes for the Real World Apps.......... 26

LIST OF FIGURES

Figure Page

Figure 1: High Level Overview of Node Static Linker 10

Figure 2: Side by Side Comparison of JavaScript Function Wrappers........................... 12

Figure 3: Library Application Using Express and PG Time Results.............................. 21

Figure 4: Library Application Using Express and Tedious Time Results 21

Figure 5: Library Application Using Express and MySQL Time Results....................... 22

Figure 6: Eval ~10 milliseconds Uncompiled; ~0.2 milliseconds Compiled 23

Figure 7: Real World Application, json-server, Time Results....................................... 25

Figure 8: Real World Application, serverless, Time Results.. 25

Figure 9: Real World Application, hexo, Time Results.. 26

INTRODUCTION

Most computer science students in the United States do not learn about security as part of their studies. Because of this, rather than building in security protections from the very beginning, security is often an afterthought that is added in the context of pressure to finish, release, publish, and sell software. This has contributed to the large and growing number vulnerabilities identified and exploited.

Data breaches are reported in the headlines daily. This includes the 2017 Equifax breach where an application vulnerability led to exposure of the personal information (names, date of birth, social security numbers, etc.) of over 147.9 million consumers—almost every adult in the United States. [1]

Other recent large-scale cybersecurity breaches include the theft of credit/debit card information from 56 million Home Depot customers, [1], Target's loss of credit/debit card information for 70 to 110 million customers, the attack on JP Morgan Chase (76 million people) [2] and the breach of the United States Office of Personnel Management (22.1 million) [3].

A study by the Center for Strategic and International Studies estimated the annual national cost of malicious cyber activity in the United States to be between $70 billion to $280 billion [4]. Beyond economic costs, computers are so ubiquitous that a cyber-attack could "disable all of our civilian and military communications, cause airplanes to fall out of the sky," and could essentially end life as we know it by turning our power grid "into a pile of burned out rubble" [5].

Manual identification of vulnerabilities is a time-consuming, error prone and difficult process. Therefore, automated vulnerability static analysis tool such as JSAI are

being developed to save time and money in identifying vulnerabilities. However, these static analysis tools often miss vulnerabilities in dynamically linked code.

A significant root cause of data breaches is malicious input. A vulnerable source is where malicious data (user input) can enter an application. One well-known example of malicious input is SQL injection where an attacker attempts to insert code into a data entry field for execution.

A vulnerable sink is defined as the point in code where a vulnerability can be exploited. If unsanitized input can be passed from a vulnerable source to a vulnerable sink, this should be flagged as a vulnerability.

Node.js is a popular development tool for web applications. There are more than 7 million Node.js instances online. "Node.js is being used with many types of tools and technologies. Databases, front-end frameworks and Node.js frameworks are chief among them." [6]. The Node.js "module.require" is a built-in function that allows for dynamic linking. This greatly increases the flexibility and functionality of the application.

However, dynamically linked code has drawbacks. Loading dynamically linked code from multiple files can slow run time. When a library is updated, it may make the dynamically linked portion incompatible with another portion of the program, causing it to crash. Dynamic linking also allows sources and sinks to be spread across multiple files. Static analysis tools like JSAI simulate JavaScript in order to find when an input from a vulnerable source can make it to a vulnerable sink. Therefore, dynamic linking, like the require statements used by Node.js, makes static analysis more difficult.

The solution is to develop static linkers. Static linkers support automated static vulnerability identification processes by combining code spread across multiple files into

one file for more accurate analysis. Every programming language handles dynamic linking in a different way. Therefore, a static linker needs to be developed for each programming language.

We built our static linker for Node.js, FrozenNode, using Closer Complier. Our results from FrozenNode testing show it created a self-contained application that had the same output as the resolved application. We demonstrated that resolved applications created by our static linker were faster than their unresolved counterparts.

Furthermore, JSAI was only able to find the vulnerabilities in the resolved code. This proved that the FrozenNode could be used to assist static analysis tools like JSAI. The creation of our static linker, FrozenNode is the first step toward creating a versatile static analysis system for Node.js.

BACKGROUND

STATIC LINKER

A Static linker is a programs that takes a program and determines what code would be imported during runtime then inserts the code into the main application. Once the static linker finds the code's location it modifies the original application to include the code statically, often by injecting the code of the dynamically linked program into the code of the application. This can be done for Dynamic Link Libraries (DLLs), imported libraries, and other forms of dynamic code inclusion. Static linkers can be used to link both compiled and interpreted programs.

Static linking varies by language because each language handles dynamic linking in a different way. For example Go, by default, performs static linking during compilation. Other languages, for example Python, by default, import the language during runtime. A high level view of the Python process is that it checks to see if a module is loaded. If it is not, it creates a module object and loads the module into that object.

There are several benefits of static linking. Typically it allows the code to load faster as the entire program is stored in the cache while it is runs. After the code is statically linked it contains the all of the dependencies in one file. It is no longer looking for external dependencies so that those external dependencies can be modified without impacting the statically linked file. Since it no longer needs external dependencies, the external dependencies can be updated without fear of crashing the program. Finally static linking of a program allows static analysis tools to assess functions that the static analysis tools may not be able to find with dynamically linked applications. Otherwise a human would have to tell the static analysis program what dynamically included functions do. This could

be very time consuming and can be prone to human error. However, with the statically linked code and the static analysis tools the user should not have to give function definitions to the program.

JAVASCRIPT (JS)

JavaScript is an interpreted programming language used primarily for client-side development. It is often used with languages like HTML and CSS to make webpages. When used in webpages, JavaScript is executed inside the browser, on the client's machine. This allows developers to offload computational work to the client's computer freeing up the server.

JavaScript supports dynamic typing and is weakly typed. JavaScript's use of dynamic typing allows the variable types to be determined when a variable is assigned. However JavaScript, as a weakly typed language, can combine variables of different types. This allows a simple expression like 1+1 to have a solution of 11 or 2. While either may be what the developer wanted, the developer needs to be sure that she/he receives the intended solution, otherwise this could cause the program to crash. There are methods to add/enforce strong typing to the weakly typed JavaScript default but they are not a part of the standards developed by Ecma International.

The Ecma organization released a standard starting in 1997 and have periodically released updates since then. Recently the updates have been released annually in June. Initially the naming conversion followed the naming pattern of ECMAScript # (ES#). Starting in June 2015 the naming convention switched to ECMAScript year so that the ES6 release was the first using this convention with an official name of ECMAScript 2015. Many of these more widely adopted standards are referenced throughout the paper.

However ES4 which was not widely adopted by major players (like Microsoft and Adobe) opting instead to improve ES3. This created ES3.1 which was later renamed to ES5.

With each update to the standard, JavaScript changes and gains more functionality. The update from ES5 to ES6 introduced major changes to JavaScript. These changes made development easier and quicker. However it had the side effect of breaking many interpreters and compilers. As of March 2018, most desktop browsers have been updated to support the new functions ES6 provided, but many compilers still have a ways to go before they can properly handle ES6. In fact, according to the official compatibility table, Babel is the compiler closest to completion with 71% of the features supported, followed by TypeScript with 59% of the features supported [7].

COMPILERS FOR JAVASCRIPT

The primary function of JavaScript compilers is to ensure adherence to established standards. JavaScript compilers do not typically compile JavaScript into machine or byte code, but instead compile JavaScript to make sure the code is syntactically correct. The compilers will then reprint the JavaScript following the standards given to them. This means they can be used to ensure company standards are met. However JavaScript compilers are more often used to convert code from one version of JavaScript to another. For example, from ES3 to ES6 or ES5 to ES3.

There are several JavaScript compilers in production. The four largest are Babel, TypeScript, Closure Compiler, and Traceur. All four of these tools are open source. Babel and TypeScript possess many add-ons to improve their functionality. These add-ons can specify transformations to standardize code or help to convert the code by specifying transformations from an ES6 command. TypeScript also acts as its own version of

JavaScript with several unique features, like the ability to declare types. TypeScript's main purpose is to convert TypeScript JS to ECMAScript. Closure Compiler does not possess the wide variety of plugins, but allows the user to specify options instead. Closure Compiler's main feature is that it reduces the size of the code. It does this by removing unused variables/functions, renaming variables, and removing comments. It is fully functional on ES3 and ES5. It can also produce ES6 code, but struggles to analyze it. Finally Traceur only converts from ES6 to ES5.

NODE.JS

Node.js is an open-source implementation of JavaScript based on Google's Chrome V8 engine designed for server-side use. It allows the front-end and back-end to be programmed in the same language, so one developer can read, write, and understand both. Node.js, written primarily in ES6, has a lot of functions built into the application including the require function which is discussed later. The Node.js application is compiled from the source code and uses some C code in addition to the JavaScript.

Node.js has several active versions as of March 2018. Two of the versions, named Boron (release 6) and Carbon (release 8), are marked as Active Long Term Support (LTS). Both LTS versions support full ES6. Boron is scheduled to be supported until April 2019, while Carbon is planned to be supported until December 2019. The current version does not have a code name and is numbered release 9. For this paper we focused on Boron as this was the main LTS when we started.

Node.js applications often use modules. Modules are similar to C and Python libraries. They allow the developer to use code from other files to avoid replicating work. Node.js uses Node Package Manager (NPM) to help manage these modules which NPM

sometimes refers to as packages. NPM is accessible via the web and shares useful information, such as downloads, version, license, and more. As of March 16, 2018, NPM claimed to have over 470,000 modules on their site. In addition the user can make his/her own modules as well as use some modules provided by Node.js. The modules provided by Node.js are compiled into the executable and can be found in the Node.js source code.

NODE.JS REQUIRE STATEMENTS

In Node.js, the built-in function, module.require allows for dynamic linking which can slow down applications. Node.js' dynamic linking is carried out through the "module.require" function, which is aliased as "require" by Node.js.

There are several interesting things to note about the require function. First, the require function loads JavaScript files by creating a function wrapper and calling it. This function wrapper creates a variable named *exports* and sets it as a property of module. Once the code is done executing the require function returns the exports object. The require function then stores the exports object in a cache that it uses if it finds the function again. Second, require function calls can be circular. For example, it is perfectly acceptable in Node.js for module A to require module B and for module B to also require module A. Finally, the require function can handle four different file types distinguished by extension. Three are always supported (.js, .json, .node) while one is only allowed in experimental mode (.mjs).

Dynamic linking, like the require statements used by Node.js, make static analysis more difficult. Static analysis tools like JSAI simulate JavaScript in order to find when an input from a vulnerable source can make it to a vulnerable sink. Dynamic linking allows

these sources and sinks to be spread across multiple files. In these cases, static analysis tools struggle to find the vulnerabilities hidden in these applications.

DESIGN

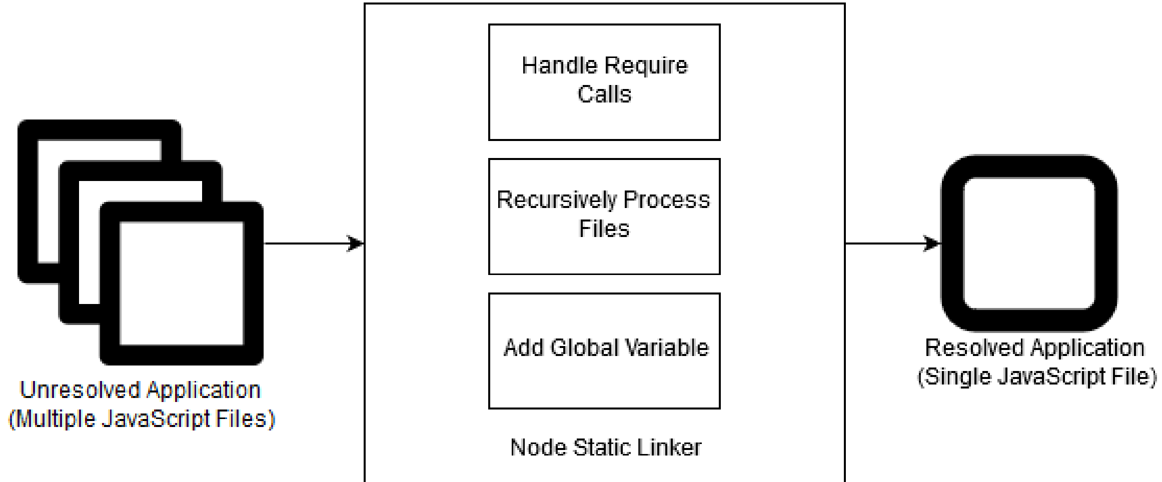

Figure 1: High Level Overview of Node Static Linker

Our design for the Node.js Static Linker tried to replicate how Node.js executed applications. We did this to avoid complications that might occur from changing when and where modules were loaded in. For example, when a module with console.log statements was loaded all of those statements print to the console. If it was loaded in a different order or loaded multiple times then the application was noticeably changed. Our method resolved applications once, like Node.js, and would only run if the application's modules loaded in the same order as Node.js. To accomplish this our design needed to emulate Node.js as much as possible. At a high-level our design found require calls, determined their type and processed accordingly. Finally it used global variables to avoid reloading modules.

REQUIRE RESOLUTION

The first challenge was to determine the resolution method for the require statements. We looked through the Node.js module source code and discovered Node.js made five different checks, while the documentation for Node.js mentioned four. According to the documentation Node.js resolved the modules in the following order: core

modules (referred to as internal modules in this paper), relative path modules, absolute path modules, node_module modules. The fifth module location Node.js checked for was the NODE_PATH environment variable. Officially NODE_PATH was kept for historical reasons and was not recommended for use although it still worked.

When a new require statement was found the static linker must perform the above resolution starting from the directory of the module being processed. If this is not done the wrong source module could be found and the application may no longer function. Because of this we recommend keeping the directory of the currently processing file with the name of the file. This allows the static linker to accurately search and find modules.

Once the module location was determined we needed to determine how to retrieve the module. As a Node.js static linker we knew we needed the source code for the modules linked in the application. The most problematic modules to retrieve the source code for were the internal modules. The internal modules were compiled into the Node.js executable and therefore not useable for static linking JavaScript. This meant we needed to get the source code from somewhere else. Luckily the source code can be found on GitHub. The rest of the modules were trivial as their source code could be found at the location of the module, unless it was a ".node" file. We chose not to handle ".node" or ".mjs" files as explained in the restrictions section.

HANDLING NEW MODULES

The next challenge was to determine how to paste the source code of the module into the application's main file, in a way that did not break the program and emulated the dynamic loading performed by Node.js. The code could not just be pasted in as it would not be functional. We also could not use function like eval as it did not load the code in a

way that allowed it to be used again. Again we looked through the module documentation and code. We found that Node.js used a function wrapper to load JavaScript files. We modified the function wrapper to allow us to store the module's exports to a global variable. This was our static method to simulate module caching. In Figure 2 we show a side by side comparison of the modified function wrapper and the original function wrapper.

Modified Function Wrapper

```
1. (function(){
2.         var module = { exports: {} };
3.         <Global_Variable> = module;
4.
5.         (function(exports, module, filename, dirname) {
6.              // code
7.
8.          })(module.exports,module,"<PathToFile>","<FileDirecroty>");
9.
10.        return module.exports
11. }()
12.);
```

Function Wrapper

```
1. (function(){
2.         var module = { exports: {} };
3.
4.         (function(exports, require, module, filename, dirname) {
5.              // code
6.
7.         })(module.exports,require,module,"<PathToFile>","<FileDirecroty>");
8.
9.         return module.exports
10. }()
11.);
```

Figure 2: Side by Side Comparison of JavaScript Function Wrappers

To be useable for our static linker the code inside the function wrapper needed to be statically linked too. Once a require statement for a JavaScript was found we stopped processing the current module and began processing the new module with the static linker. This simulated the depth first module resolving method used by Node.js. It also created a recursive call that stopped once there were no require statements or it only found encountered modules. After completing processing for a module, the static linker would replace the require statement in the module that required it, with the function wrapper and linked code from the required module.

This design also used a function wrapper for JSON modules too. Node.js does not do this but does still cache the results for later use. This function wrapper was made to allow us to easily assign the global variable to simulate the caching.

CACHED MODULES

As mentioned our design used global variables to simulate caching performed by Node.js during runtime. Using this method, each module needed to have a uniquely named global variable otherwise the simulated caching would not work. To load a cached module the global variable would be used to replace the require statement. Specifically, the require statement was replaced with global variable's *exports* field. Anything attached to the original require statement was reattached to the exports field of the global. For example, the static linker has encountered module "a" before and assigned it to the global variable, globalA. Now the static linker has found the code snippet "require('a').fun1();". It finds that require('a') resolves to module "a" and that module "a" has been cached before. It uses the global associated with module "a" and replaces the require statement. The result would be "globalA.exports.fun1();".

For the simulated caching to work properly, the global variables needed to be kept across all modules processed for the application. This had two major benefits beyond simulating Node.js runtime. The first was that it kept the file as small as possible. Whenever the module was encountered after the initial time it was replaced with a global variable rather than the function wrapper and code. This also meant that the code would run faster as there were fewer instructions it reread. The other benefit, which was a major challenge of this project, was that the simulated caching and tracking of modules across the application allowed the static linker to avoid getting stuck in infinite loops. Static linkers could get stuck in loops when resolving require statements because Node.js supports circular dependencies. In fact the internal modules make extensive use of this. The caching and tracking method allowed the static linker to detect circular dependencies

and stop processing which gave our recursive calls the other stopping condition mentioned earlier.

VARIABLE PREPENDING

Another challenge came from the fact that it was hard to tell how many modules were in a Node.js application. The main reason for this was that modules can and do require other modules. The global variables needed to be declared in order for the code to function. This could of course be solved by just declaring 1000 variables at the top to begin with, but that may not be enough or it could be way too many. To solve this the variable declaration was appended to the top of the document after all the other processing for the application was completed. This allowed for a dynamic number of global variables. To get the number the prepender used the tracked modules/current variable name and declares the global variables at the top. This allowed the variables to be used throughout the document without errors.

IMPLEMENTATION

Our implementation of the static linker, FrozenNode, was comprised of a preprocess wrapper, a variable prepender, and a require resolver. The preprocess wrapper was the entry point for FrozenNode. Many flags and filenames were needed for the require resolver to function. The preprocess wrapper, called the require resolver with all the necessary filenames and flags set. Once the require resolver finished, the preprocess wrapper then called the variable prepender with all the correct flags. In this way, FrozenNode allowed the entire static linking process to be called quickly and easily.

Both the preprocess wrapper and the variable prepender assisted in the static linking process but the majority of the static linking was carried out through the require resolver.

The require resolver was built on top of Google's Closure Compiler to accurately process JavaScript. The require resolver detected require statements by forward checking for module.require and require functions once the require resolver detected a call statement. The require resolver called node's require.resolve function from the folder of the processing module to find the location for the module. The require.resolve function returns a name if it is an internal module, a path if it existed, or an error if it is a normally unusable internal module or the module was not downloaded. It used this information to determine where to find the file.

In some cases, minor manual fixes needed to be applied. The require functions were resolved in order of appearance on the page not in the order they appeared during runtime. This meant that at most one global variable per required module needed to be swapped with the function wrapper and code of that module. However the actual number of swaps was usually fewer than the number of modules. In some cases, naming conflicts occurred during the resolving process. This typically occurred when Node.js globals were overwritten in a required file causing globals to be invalid once resolved. After these fixes were applied, the code was a self-contained, functional equivalent of the unresolved code.

EVALUATION

ROBUSTNESS

Our tool, FrozenNode, is available on GitHub[1] and uses Travis CI to test commits. We created a series of automated tests that make sure FrozenNode will retain its functionality before developers modify the code. The simplest of these tests makes sure FrozenNode can handle JavaScript and JSON files. For these tests, the relative location was hardcoded in. This means that when FrozenNode placed the resolved application inside of the default output folder the resolved application could not rely on the require function to get the desired functionality. This allowed us to confirm that it was indeed resolving the code and not just reprinting the file. This allowed us to make sure that the tool could at least handle simple files as expected.

The next set of tests check for multilayer require calls, caching, and circular dependencies. These tests had a higher chance to break and needed to be performed in the order: 1. multilayer and caching 2. circular. Circular dependencies will resolve correctly only if both multilayer and caching are working as intended. Multilayer and caching both could work without circular dependencies working.

The multilayer test made sure FrozenNode recursively processed require statements. Recursive processing is needed to completely resolve most applications. The caching test made sure FrozenNode did not reload a module it had encountered. We accomplished this by using console.log statements. Console.log statements that were not attached to exports objects would only print on the initial load in Node.js applications. However if caching was not properly done it would print more than once on resolved applications. We used this to compare the outputs of the program.

1: https://github.com/sefcom/FrozenNode-nodeRequireResolver

The circular dependency tests built off of the caching and multilayer tests by having two modules require each other. Circular dependencies need the cached variable in order to not get stuck in loops. The cached variable must be used to replace the require statement before the variable has useful values which can lead to errors. This test makes sure that we get the same errors as Node.js during runtime. In addition, the circular dependency test made sure that the cached variable was used (no-loop) and that the program did not resolve extra layers (using the console.log method again).

We also made tests that used .node files. These tests made sure that resolving would not break the code even if it encountered require functions it did not know how to handle. These test cases required that the resolved code and unresolved code ran in the same location so that they could both use the .node module.

Finally we made some test cases that were meant to fail. For these test cases we made sure the logs caught the error and reported it properly. We also made sure that the application gave the expected output. This ensured that the program did not add a module that did not exist.

TESTING METHODOLOGY

Several tests were performed on the applications before and after they were resolved. These tests helped to prove the statically linked code generated by our program was self-contained, faster, and useable for static analysis. Resolved code refers to both code with every module except the internal modules resolved and code with every module resolved.

To determine if the code was self-contained, we took both the unresolved code and resolved code and moved them to a different location. We then ran the unresolved

code to make sure that it failed. With this confirmed, we knew that the code could not rely on require for external modules. We then ran the resolved code. If it ran then we determined it was self-contained.

To measure time improvements, we modified the code to create test cases that could be run repeatedly. We also modified the code to measure how long it took to get from the start to an identified exit point. This was done using a JavaScript global called console. We used console's properties named time and timeEnd. After it was modified, we determined how to automatically get to that branch. This often involved passing parameters to the application or generating and sending specific http packets. Once this method of input was identified, we made a python script that called the unresolved and resolved code 3n times. The first n were calls to unresolved or resolved code exclusively. The next n calls alternated resolved then unresolved code. The final n calls alternated unresolved then resolved code. We then averaged these run times to get the mean runtime for each version of the run and the overall average.

To test whether or not the code could be run through a static analyzer, we used JSAI. We added some Node.js globals to the notJS language used by JSAI to analyze the JavaScript. Once this was done, we ran JSAI on the resolved and unresolved code.

RESULTS

We obtained results on the performance of FrozenNode starting with several types of test applications, then we extended the tests to real world applications from GitHub. We made the test applications from several JavaScript files that used different Node.js functions and modules. Most of the test applications used internal modules, like child_process and http. If the test application used only internal modules, we referred to

it as an internal module application. We also made several test applications from popular modules downloaded from npm. The modules we used were express, mysql, pg, request, and tedious. We called these test applications library applications. We also made some simple test applications that used only the globals, process and eval. These globals were separated into different files, with at least one require to connect them. Finally we found popular real world applications on GitHub. On all four types of application, we ran the tests described in the testing methodology. For the time improvement tests we used 250 for our n.

The number of unique modules FrozenNode resolved for each internal module can be found in Table 1 along with the number of manual changes needed. For both of the internal modules we swapped one global variable with its associated function wrapper to allow the internal module application to run. When we ran the internal module applications through the tests we got interesting results. For the self-contained tests, both unresolved and resolved applications were able to run as long as Node.js was installed. This was because these modules were compiled into the Node.js executable. Since the modules were compiled we expected the unresolved and resolved applications to have comparable runtimes. In our timed tests we found the unresolved code was a little less than 10 milliseconds faster on average.

	Fixes	Number (Modules)
child_process	1	27
http	1	40

Table 1: Number of Unique Modules by Required Internal Module

We used FrozenNode to process the library before we processed the library modules. This saved time as the fixes only had to be applied to the library once, as opposed

to every time FrozenNode was run on a case that used the library. The number of unique modules FrozenNode resolved by library is shown in Table 2 along with the number of manual fixes needed to run the library modules. The self-contained test was passed by both the resolved applications, but not the unresolved application. The speed test was conducted with unresolved code and the resolved application that did not resolve internal modules. The exact speed differed by the libraries used in the application, but the ranges, in milliseconds, are shown in Tables 3 and 4.

	Fixes	Excluding Internal (Modules)	Including Internal (Modules)
Express	9	95	149
Mysql	2	74	115
Pg	0	32	74
Request	0	161	212
Tedious	2	216	257

Table 2: Number of Unique Modules by Library from NPM

	Unresolved (ms)	Resolved (ms)
Average (Only)	262.912216	212.719048
Average (Alternating)	269.037968	208.682964
Average (Rev Alt.)	264.762512	219.299592
Average (Overall)	265.570898	213.567201

Table 3: Largest Proportional Difference in Runtimes for the Library Test Cases

	Unresolved (ms)	Resolved (ms)
Average (Only)	693.066764	603.487456
Average (Alternating)	690.3389	606.859408
Average (Rev Alt.)	693.051384	606.56224
Average (Overall)	692.152349	605.636368

Table 4: Smallest Proportional Difference in Runtimes for the Library Test Cases

Figures 3, 4, and 5 show a box plot of three different test applications.

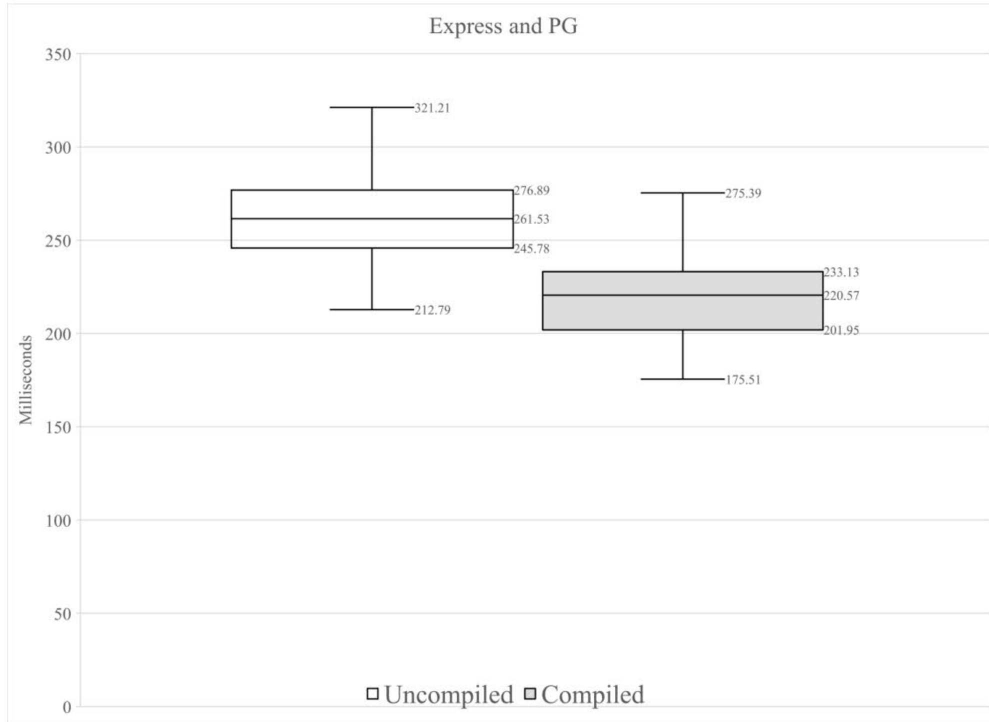

Figure 3: Library Application Using Express and PG Time Results

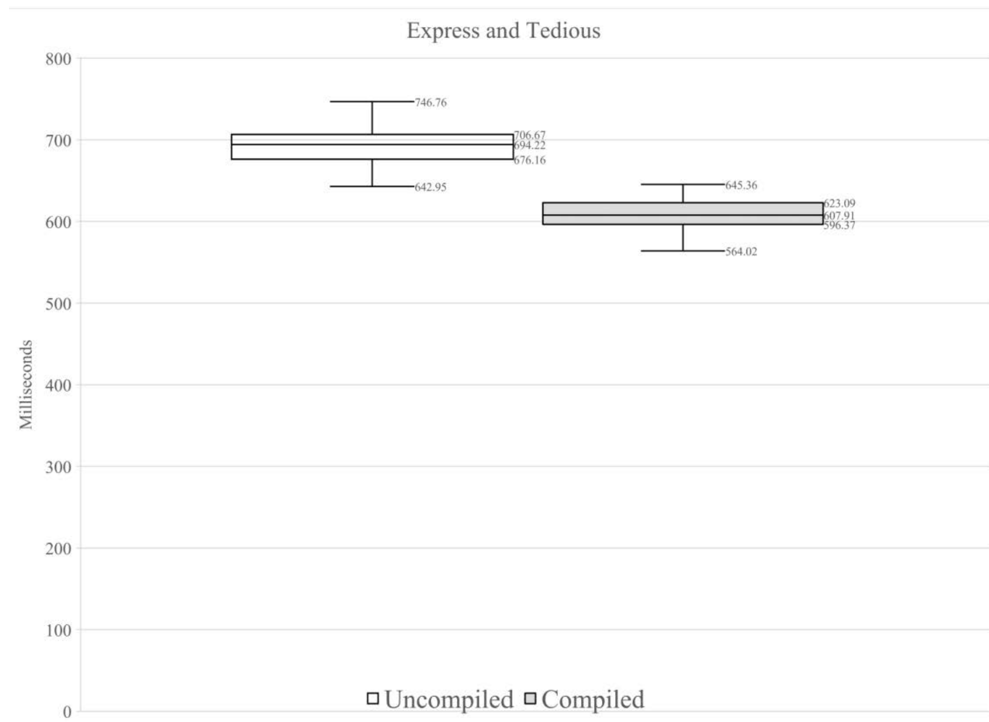

Figure 4: Library Application Using Express and Tedious Time Results

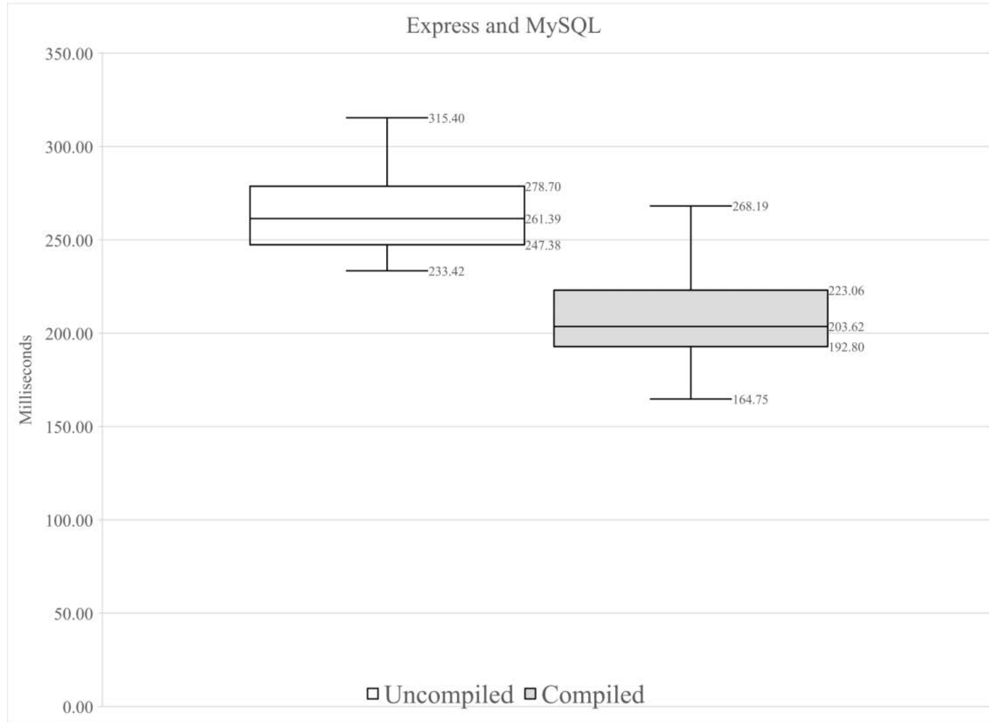

Figure 5: Library Application Using Express and MySQL Time Results

In the library application the compiled code ran faster than the un-compiled code.

The simple applications did not have any require statements except for those that connected the files and the number only affected the runtime. Depending on how the applications were moved, unresolved applications did not function. However the resolved applications always did. In Table 5 we show the average run times for an application that used 10 requires to separate process and eval.

	Unresolved (ms)	Resolved (ms)
Average (Only)	10.58612	0.212028
Average (Alternating)	10.555044	0.21184
Average (Rev Alt.)	10.678272	0.212448
Average (Overall)	10.606478	0.2121053

Table 5: Average Runtime for Eval/Process Test Using 10 Require Statements

As shown in Figure 6, the Eval test case demonstrated that the compiled code ran more than 50 times faster than the uncompiled code.

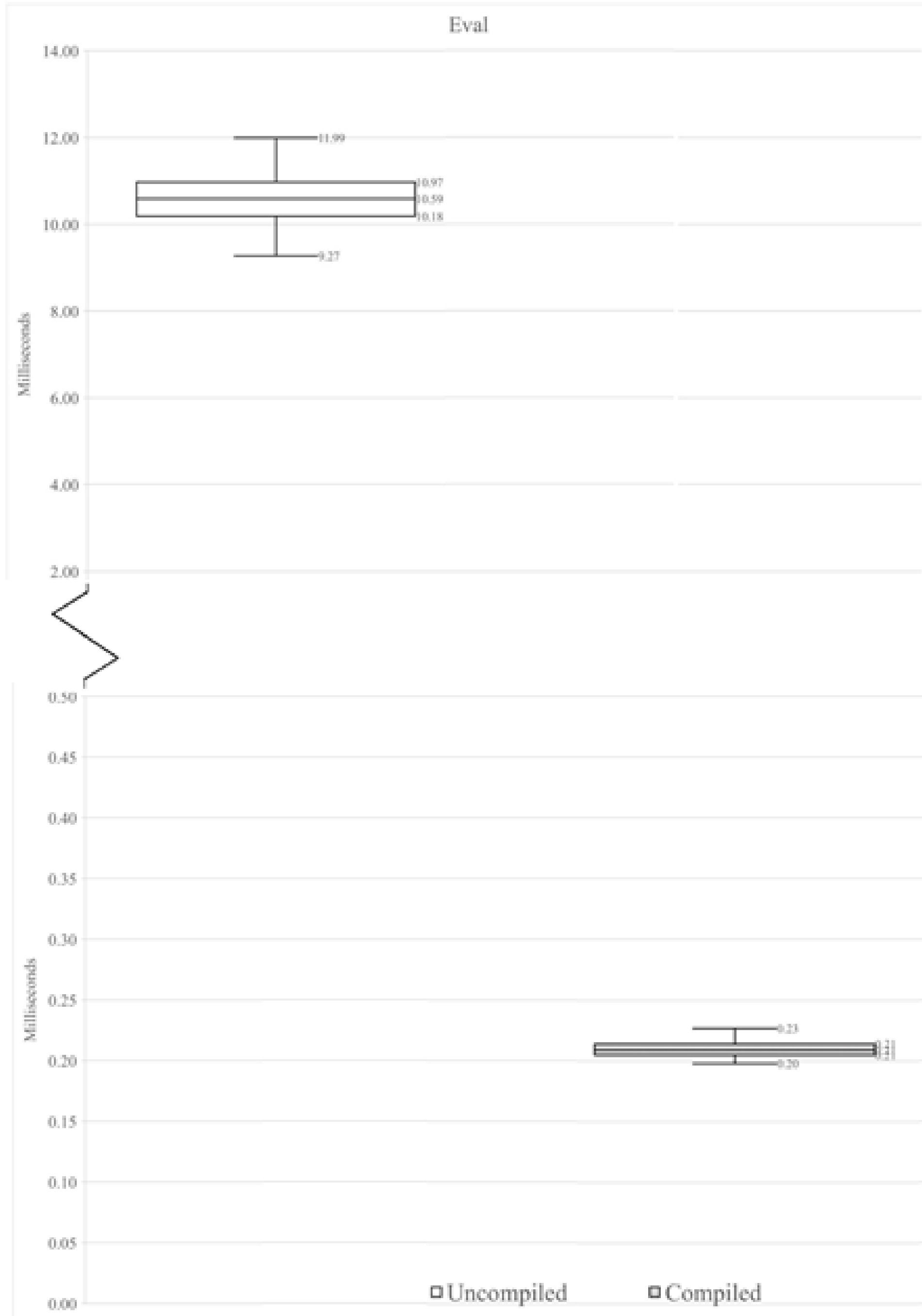

Figure 6: Eval ~10 milliseconds Uncompiled; ~0.2 milliseconds Compiled

However the most important test for the simple applications was the static analysis test. JSAI was able to successfully run all the way through the resolved application. In

addition, JSAI was only able to find the vulnerabilities in the resolved code proving that the FrozenNode could be used to help static analysis tools.

Finally we tested four popular real-world applications. One, named ghost, relied heavily on dynamic requires and could not be resolved. The other three resolved successfully. We resolved the applications three times instead of two. For the third time, if the application used one of the five libraries we resolved for the library tests, we made the application use the resolved version of the library. In Table 6 the fixes column is the number of fixes that needed to be applied to the pre-resolved library application. For the self-contained tests we used all the functions we could to make sure the application was not broken after it was resolved. All three applications passed the self-contained tests. The speed tests used an identified path that could be ran repeatedly to get the average runtime. Figure 7, 8, and 9 show the results of the speed tests using box plots. The proportionally slowest and fastest average runtimes of are shown in Table 7 and Table 8.

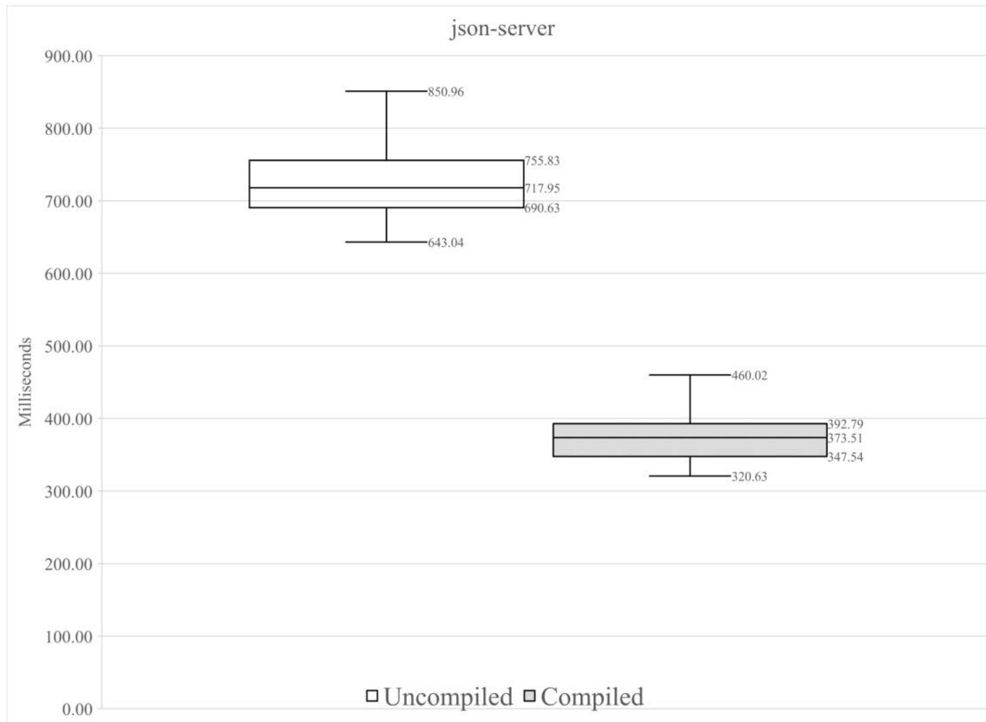

Figure 7: Real World Application, json-server, Time Results

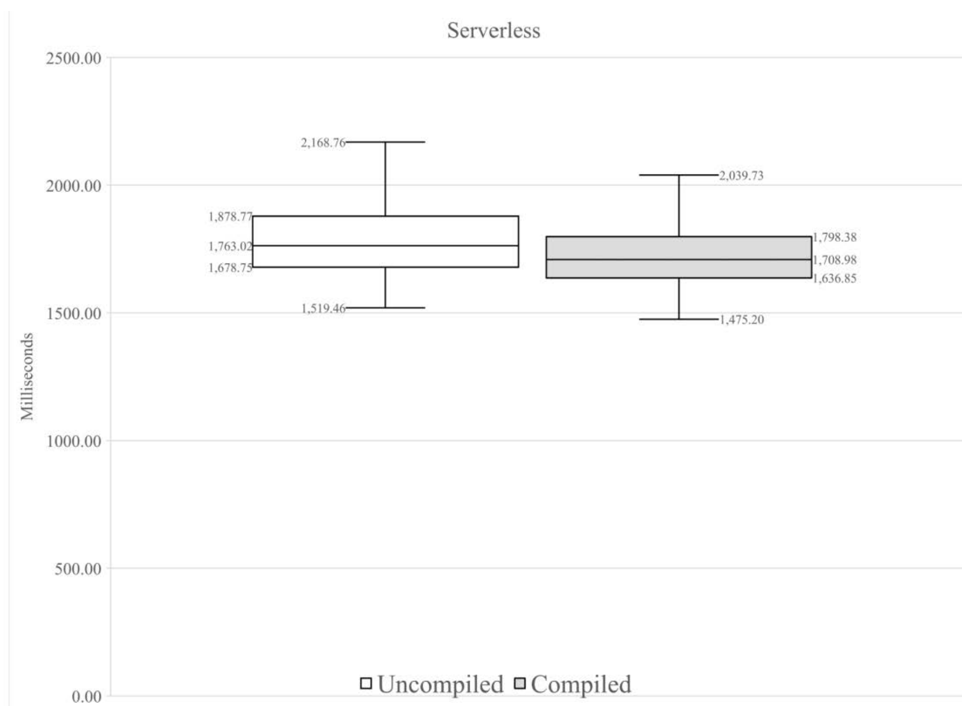

Figure 8: Real World Application, serverless, Time Results

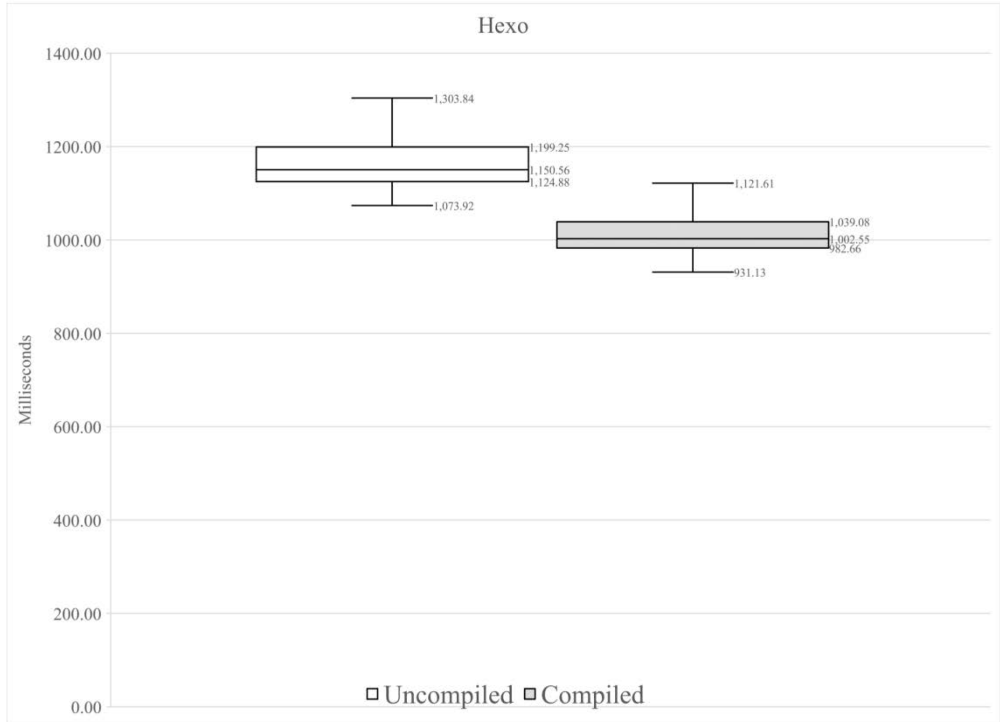

Figure 9: Real World Application, hexo, Time Results

	Fixes	Excluding Internal	Including Internal
json-server	1	349	453
Serverless	2	363	424
Hexo	3	169	202

Table 6: Number of Unique Modules by Real World App

	Unresolved (ms)	Resolved (ms)
Average (Only)	729.867428	373.913556
Average (Alternating)	714.122956	383.000888
Average (Rev Alt.)	734.757004	368.232988
Average (Overall)	726.249129	375.049144

Table 7: Largest Proportional Difference in Runtimes for the Real World Apps

	Unresolved (ms)	Resolved (ms)
Average (Only)	1787.286696	1725.219964
Average (Alternating)	1785.841572	1719.745992
Average (Rev Alt.)	1780.003268	1726.593916
Average (Overall)	1784.377178	1723.853290

Table 8: Smallest Proportional Difference in Runtimes for the Real World Apps

DISCUSSION

FrozenNode was able to resolve the majority of Node.js application and libraries we tried. We found that the applications ran faster when they were statically resolved. However this may not always be the case. Large unused module in may slow down the loading more than the speed gain. However for all our real world test cases this was not the case.

LIMITATIONS

FrozenNode, as implemented, cannot handle reassignment of require or module.require, aliasing of the require function, seemingly dynamic require statements, dynamic require statements, or the require statements that are compiled (.node) or experimental (.mjs) Node.js modules. In addition to these restrictions, unique to FrozenNode, FrozenNode also suffered from some restrictions imposed by Closure Compiler.

FrozenNode assumed that any module.require or require call was a call to Node.js' module.require function. However, these two aliases are not keywords and can be reassigned. If both are reassigned FrozenNode will be incapable of resolving required modules. If either is changed FrozenNode will attempt to resolve values passed into the changed function as modules. This can lead to bad static linking if a string is passed into a changed function and that string resolves to a valid module. For simplicity we assumed developers would not reassign these two aliases. The real world projects we looked at did not rename require or module.require, however it is still possible and FrozenNode will not be able to handle it.

Aliasing is when a variable is used to give a new name to a function or object. Aliasing of the Node.js' module.require function can be accomplished by setting a variable equal to module.require or require before reassignment of either, if they are reassigned. This allows the developer to make a call using the alias they defined, but will cause FrozenNode to miss the function call leading to incomplete programs. Like reassignment we assumed that the developer would not do this as they typically have no reason to. However, some of the libraries from NPM did alias the function. This was discovered during the self-contained tests as the example tests would fail. We were able to easily fix this by replacing the alias with module.require or require, however all of NPM libraries we tested that did this also used truly dynamic requires in one of their dependencies, which we were unable to resolve.

FrozenNode does not track the values of variables. This means that FrozenNode will not resolve any require function calls that uses a variable. When referring to seemingly dynamic require statements we are referring to statements that use a variable, but the variable is statically declared. Most of the real world examples of seemingly dynamic requires used environment variables. Seemingly dynamic requires can be manually resolved by replacing the variable with the string name or location of the module. Dynamic require statements on the other hand are generated during runtime. These are much harder to resolve for static analysis and may not be possible to resolve completely.

FrozenNode does not support compiled (.node) modules because this tool statically links source code. Since the compiled programs are written in languages like C and then compiled into machine code, their source code will not be able to be interpreted by the Node.js executable. This means that the code would no longer function. In addition, most

static analysis tools support specific languages and would have trouble working on a file that contained both C and Node.js source code. Experimental files are also known as ECMAScript Modules. FrozenNode does not support this file type because Node.js code may or may not work if these modules are used.

For now though, it may need some manual resolving of name conflicts and/or require functions that appeared to be dynamic. The code may also need to have some of the global variables swapped with their associated wrapper and code because FrozenNode resolves require functions in order they appear.

Closure Compiler, which FrozenNode is built upon, has trouble converting ES2015 to ES5 or below. Since Node.js is written primarily in ES2015 FrozenNode can only create ES2015 JavaScript files when internal modules are resolved. It also has trouble performing code cleaning on ES2015, so FrozenNode needs to be run in the compilation mode, white space only.

FUTURE WORK

In this study, we created a new tool, FrozenNode, to statically link Node.js applications. Future work will need to address dynamic require functions and aliasing. To get static analysis ready to be used with any Node.js project, work will need to be done to abstract process.binding function calls as well as some internal code. Since the Node.js source code is in ES6, the static analyzer will also have to be able to handle ES6 or the ES6 will have to be converted to ES5/ES3.

As mentioned FrozenNode did not resolve require functions that used variables, even if these variables were statically assigned. Work can be done to expand FrozenNode's versatility by developing a method and code to track variable equivalencies. Depending on

how the equivalencies are tracked this can also solve the aliasing issue. FrozenNode does not support renaming of “module.require” or “require”. This means that it could miss require functions if they were aliased. However, with methods for tracking variables, variables equivalent to the require function could also be tracked. The code would need to be modified to forward check for these dynamically.

If needed, further work can help FrozenNode support “.node” or “.mjs” files. To add these, a method to statically extract the exports would need to be developed to allow the JavaScript to use the file. Barring that, it might be necessary to convert the source code of those files to JavaScript. Another set of issues occur when resolving internal modules statically. Some functions from the internal C code can cause Node.js applications to crash as they are not declared globally outside of the Node.js executable. The most prominent of these is in the internal module net. Whenever it tries to write to the socket it calls a function from “src/node_lttng.cc”. It does not use process.binding to include this. A JavaScript version may need to be included at the top of the resolved code to be fully functional.

As of the writing of this paper, manual fixes may need to be applied to create functional files. Finding a method to automate this would make the tool more accessible to the everyday users. It will also help save man-hours and by extension money.

So far the fixes can be easily solved by hand and will only save time in creating a functional application, but there are issues that need to be solved for vulnerability analysis. The biggest is that the Node.js source code was written in ES6. Test cases that resolved the internal modules could not be translated into usable ES3 or ES5 code by Babel, Closure-Compiler, or TypeScript.

However TypeScript converted applications were close to working with JSAI. JSAI currently does not support ES6 processing as it uses Rhino 1.7R3 to read the JavaScript. This gives JSAI a JavaScript Abstract Syntax Tree (AST) which it then converts to a NotJS's AST. The conversion process and Rhino need to be updated to work with ES6 standards. JSAI also defines JavaScript globals in NotJS so that it can understand what the global does. Node.js adds some new globals to JavaScript that should be added to NotJS. However, most of these do not need to be defined. We already defined a little bit of the global "process" for the test cases we made, but process.binding still needs to be abstracted. Process.binding is similar to Node.js require statements, but is used for C files that are compiled into the Node.js executable. As mentioned above, some C files (e.g. lttng.cc) do not need to be included with process.binding. These files will likely need a JavaScript equivalent or NotJS abstraction in order to be analyzed appropriately.

Once either JSAI is updated or the ES6 conversion tools can convert ES6 to ES3/ES5 reliably, these tools will be effective for large projects that use internal modules and projects that are written in ES6

RELATED WORK

Wassermann and Su (2004) in [8] presented the design of the first static analysis framework to identify where potential attacks may occur. This initial effort looked only at SQL injection, while our study focused on creating code that will eventually be able to be used to conduct automatic static analysis of Node.js web application vulnerabilities that include SQL injection as well as code/command injection.

Much of the previous work [8, 9, 10, 11, 12] on web-based static analysis focused on PHP projects. Huang et al (2004) in [9] implemented static analysis on PHP. The implementation was tested on sourceforge projects. Jovanovic et al (2006) in [10] and Xie and Aiken (2006) in [11] conducted taint analysis for PHP XXS attacks. Balzarotti et al (2008) in [12] used static and dynamic analysis to validate sanitization in web applications and tested the sanitation method on PHP projects. Sun et al (2011) in [13] was the first to examine access control vulnerabilities in web sites and implemented it on PHP projects. Our work is a step toward extended the analysis beyond PHP to Node.js.

Livshits and Lam (2005) in [14] used the sources and sinks method of static analysis. They also described a tool that seemed similar to JSAI, but this tool was implemented for Java. Our work takes vulnerable sources and sinks identified for Node.js by Wasserman et al. (2017) in [15] a step forward by statically resolving Node.js applications to allow the entire application to be processed at once.

Doupé et al (2011) in [16] focused on Execution After Redirect (EARs), a type of logic flaw that occurs when a program does not stop executing when it redirects. The paper looked at 18,127 open source ruby-on-rails application and found 100s of vulnerable statements with a low false positive rate.

Bisht et al (2011) in [17] used a white box method to identify parameter tampering vulnerabilities. This study focused on LAMP applications. They also used a black box analysis tool they developed to show the effectiveness of the white box tool.

Kashyap (2014) in [18] described JSAI, a formally specified, robust abstract interpreter for JavaScript. Their goal was to promote static analysis in Javascript. Our work used JSAI to determine if a static analysis tool could use the linked code produced by FrozenNode.

Doupé et al. (2013) in [19] presented a program based on static analyzers that separated JavaScript from HTML in ASP.NET. It was intended to secure legacy web applications (of benign code) by removing inline JavaScript. It did this with non-dynamic JavaScript by statically rewriting the code into a file. This was done at the binary level for access to the system libraries. Once the process was done, the paper found that the original load time and the new load time were indistinguishable. Similarly, our work statically re-wrote Node.js files to be able to include all the libraries and other functions.

Collberg et al. (2005) in [20] made a static linking system that tried to mitigate the draw backs of static linking. It did this by sharing chunks of data with a copy on write methodology. Although we did not use this method we did perform static linking for many of the reasons pointed out in this paper. Kell et al. (2016) in [21] did an in depth look at static linking with ELF files. This study reviewed the roles linking played and described the model for ELF on Unix platforms. Likewise we detailed the static linking process used by Node.js.

Tilkov in [22] and Lei in [23] looked at Node.js' viability and speed as a server-side language. [22] noted that, "Node.js's architecture makes it easy to use a highly

expressive, functional language for server programming, without sacrificing performance " Lei et al. (2014) in [23] found that Node.js was faster than Python and PHP. This is important for this paper as one goal was to try and increase the server's speed. Although, in 2018, the proof that Node.js is a viable backend language as many servers are now using it, these both served as early proof of its viability.

Finally Madsen et al. (2015) in [24] created an application to statically analyze Node.js. It was designed specifically to work with event based JavaScript. It was implemented on Radar. The tool abstracted the internal modules which will need to be updated as Node.js is updated. Our tool, on the other hand, was capable of resolving internal modules so that static analysis tools can analyze them without worry of inaccurate abstractions. In addition they stated that only core modules needed to be abstracted. They continued that if a module was written in pure JavaScript it could be directly fed to the analysis, however, they did not look for errors that were started in one file and ended in another. Finally their tool was not for vulnerability analysis, but for bug/error analysis.

CONCLUSION

We have described a new static linker for Node.js named FrozenNode. Our tests demonstrated that the resolved application ran much faster than the unresolved application. Furthermore, the static linker made it possible to analyze an entire project to find vulnerabilities spread across files. To be useful for vulnerability analysis with more complex Node.js projects involving internal modules, additional work, especially developing a static analyzer that can handle ES6 and/or effective ways to convert ES6 to ES5/ES3, will be needed. This will be the focus of future work.

REFERENCES

[1] T. Armerding, “The 17 biggest data breaches of the 21st century”, *CSO*, January 26, 2018. [Online]. Available: https://www.csoonline.com/, CSO. [Accessed: 3/19/2018]

[2] J. S. Greenberg, M. Goldstein, and N. Perlroth. “JPMorgan Chase Hacking Affects 76 Million Households”, *DealBook*, October 2, 2015. [Online]. Available: http://dealbook.nytimes.com/. [Accessed: 1/5/2016]

[3] E. Nakashima. “Hacks of OPM databases compromised 22.1 million people, federal authorities say”, *The Washington Post*, July 9, 2015. [Online]. Available: https://www.washingtonpost.com/. [Accessed: 1-5-2016]

[4] J. Lewis. “The Economic Impact of Cybercrime and Cyber Espionage”, *Center for Strategic and International Studies*, July 22, 2013. [Online] Available: https://www.csis.org/ [Accessed: 2/21/2016]

[5] J. McAfee. “We aren't talking enough about cybersecurity”, *Business Insider*, January 17, 2016. [Online]. Available: http://www.businessinsider.com/. [Accessed: 2/23/2016]

[6] “Node.js 2017 User Survey”. *The Linux Foundation*. [Online] Available: https://foundation.nodejs.org/. [Accessed: 3/21/2018]

[7] kangax, webbedspace, and zloirock. “Compatibility Table”. [Online] Available: http://kangax.github.io/. [Accessed: 3/28/2018]

[8] G. Wassermann and Z. Su, “An Analysis Framework for Security in Web Applications”, *SAVCBS 2004*. Newport Beach, California, USA, October 31-November 1, 2004

[9] Y. Huang et al. “Securing Web Application Code by Static Analysis and Runtime Protection”, *Proceedings of the 13th international conference on World Wide Web (WWW ’04)*. New York, NY, USA, May 17 - 20, 2004

[10] N. Jovanovic et al. “Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities”, *2006 IEEE Symposium on Security and Privacy (S&P'06)*. Oakland, California, USA, May 21-24, 2006

[11] Y. Xie and A. Aiken. “Static detection of security vulnerabilities in scripting languages”. *Proceedings of the 15th USENIX Security Symposium*. Vancouver, BC, Canada, July 31-August 4, 2006

[12] D. Balzarotti et al. "Saner: Composing static and dynamic analysis to validate sanitization in web applications". *2008 IEEE Symposium on Security and Privacy (sp 2008)*. Oakland, California, USA, May 18-22, 2008.

[13] F. Sunm, L. Xu, and Z. Su. "Static Detection of Access Control Vulnerabilities in Web Applications", *Proceedings of the 20th USENIX conference on Security (SEC'11)*. San Francisco, California, USA, August 08-12, 2011

[14] V. Livshits and M. Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis", *USENIX Security Symposium*. Baltimore, Maryland, USA, July 31-August 5, 2005

[15] J. Wasserman, A. Doupé, G. Ahn, and Z. Zhao, "TSCAN: Toward a Static and Customizable Analysis for Node.js", B.S. thesis, CIDSE, ASU, Tempe, Arizona, 2017.

[16] A. Doupé, B. Boe, C. Kruegel, and G. Vigna. "Fear the EAR: Discovering and Mitigating Execution After Redirect". *ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, USA, October 17-21, 2011.

[17] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan, "WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit", *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS '11)*. Chicago, Illinois, USA, October 17-21, 2011

[18] V. Kashyap, and B. Hardekopf, "Security Signature Inference for JavaScript-based Browser Addons", *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. Orlando, FL, USA, February 15-19, 2014

[19] A. Doupé et al. "deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation". *ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, November 4-8, 2013.

[20] C. Collberg, J. Harman, S. Babu, and S. Udupa. "Slinky: Static Linking Reloaded", *USENIX Annual Tech Conference 2005*. Anaheim, California, USA, April 10-15, 2005

[21] S. Kell, D. Mulligan, and P. Sewell. "The Missing Link: Explaining ELF Static Linking, Semantically", *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. Amsterdam, Netherlands, November 2-4, 2016

[22] S. Tilkov and S. Vinoski. “Node.js: Using JavaScript to Build High-Performance Network Programs”. *IEEE Internet Computing*, vol. 14, no. 6, pp. 80-83, November, 2010

[23] K. Lei, Y. Ma, and Z. Tan. “Performance Comparison and Evaluation of Web Development Technologies in PHP, Python and Node.js”, *2014 IEEE 17th International Conference on Computational Science and Engineering*. Chengdu, China, December 19-21, 2014

[24] M. Madsen, F. Tip, and O. Lhoták. “Static Analysis of Event-Driven Node.js JavaScript Applications”, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. Pittsburgh, PA, USA, October 25-30, 2015.