

Lecture #9: Static Analysis for Security

UCalgary ENSF619

Elements of Software Security

Instructor: Lorenzo De Carli (lorenzo.decarli@ucalgary.ca)

Partly based on slides by Drew Davidson, University of Kansas

What's the topic of today's lecture?

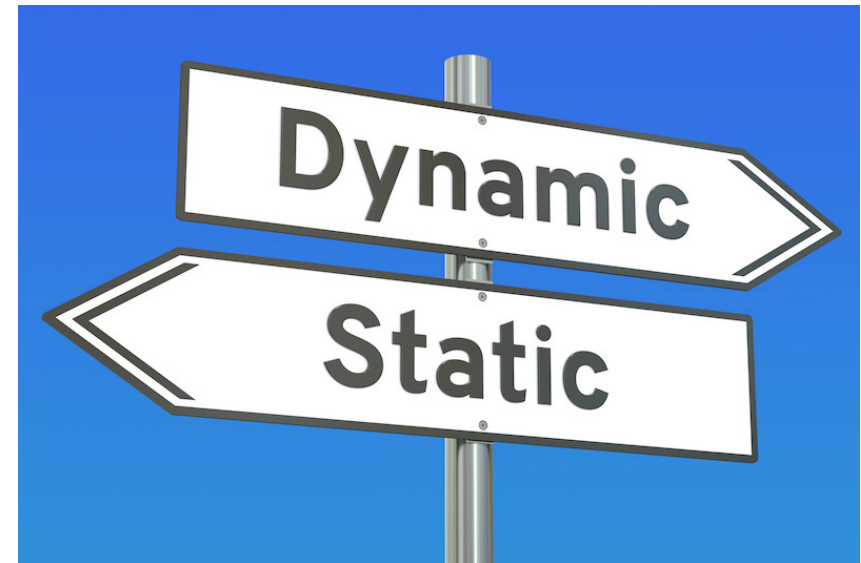
- Attack surface reduction in JavaScript programs, but more broadly...
- **Static analysis!**
- ... but what is static analysis? And what does “static” mean?

Analysis in Contrast


Static analysis – analysis that is done without running the program

Dynamic analysis – analysis that is done with running the program

Simplest example - testing



Static analysis for security

- **Static analysis** refers to a variety of techniques for analyzing software artifacts, for various purposes
 - **Optimization** (e.g., in compilers)
 - Detecting **software bugs**
 - Detecting **vulnerabilities**
 - Performing **software measurements**
 - **Improving security properties** 
- Common characteristics: program is analyzed **without executing it**

“Trivial” Syntax Analysis

Some troubling behavior of a program may be discoverable via simply observing syntactic structure

```
int main(int argc, const char * argv[]){  
    const char * password = argv[1];  
    if (password == "supersecret"){  
        authenticate();  
    }  
}
```

Static analysis to detect bad practices

Software engineering “code smells” / stats

Use of the forbidden / arcane constructs (e.g., “eval” in JavaScript)

Cyclomatic complexity

Long functions

STATIC Analysis – More Opportunities

Provide assurances about what a program will NEVER or ALWAYS do

- Static analysis might report EVERY program that (possibly) has a null-pointer dereference
- Static analysis might certify EVERY program that (definitely) is null-pointer dereference free

“Hey! Those are the same thing!”

Program verifier (detect “good” programs)

Complete (no FNs) – all good programs are reported

Sound (no FPs) – all bad programs are unreported

Bug finder (detect “bad” programs)

Complete (no FNs) – all bad programs are reported

Sound (no FPs) – all good programs are unreported

STATIC Analysis – More Opportunities



For security analysis, we want to lock out “bad” programs (even at the cost of locking out some “good” programs)

Program verifier (detect “good” programs)

Complete (no FNs) – all good programs are reported

Sound (no FPs) – all bad programs are unreported

Bug finder (detect “bad” programs)

Complete (no FNs) – all bad programs are reported

Sound (no FPs) – all good programs are unreported

The Good, The Bad and The Analysis

The good news about static analysis:

You can see beyond the instructions that are executed in an individual trace

The bad news about static analysis:

You need to construct the conditions/circumstances/context in which those instructions are executed

*p = 2

You exist in the context of all in which you live and what came before you

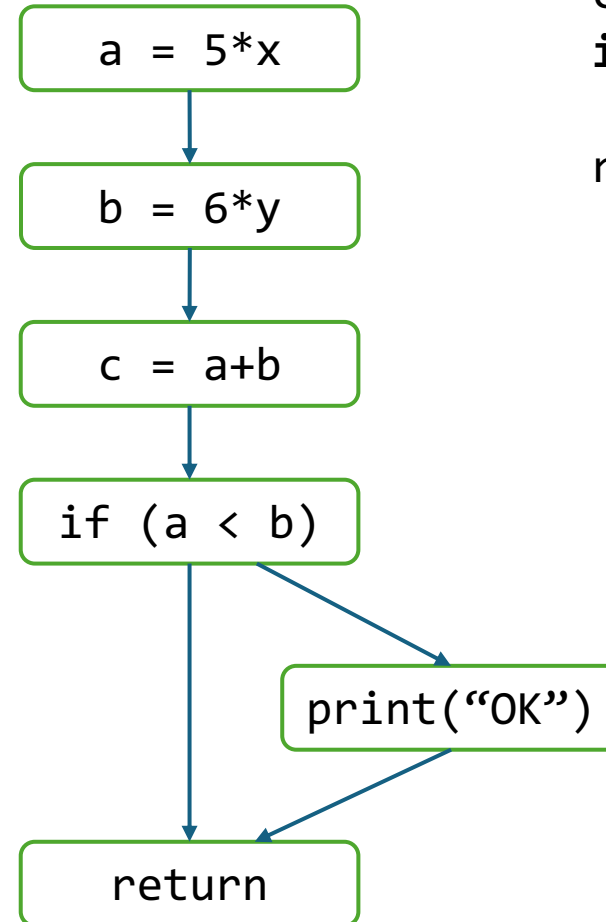
Static analysis – program representations

- Many (although not all) static analysis analysis are really **graph algorithms** – i.e., they operate on graphs!
- But... **programs are not graphs?**
- General idea: **convert** source code (or binary) to graph representation, **run the analysis** on graph

Graph representation 1 - CFG

- **Control-flow graph**
- Directly represents the **execution flow** of the program – nodes are instructions (or basic blocks), edges represent the order in which instructions are executed

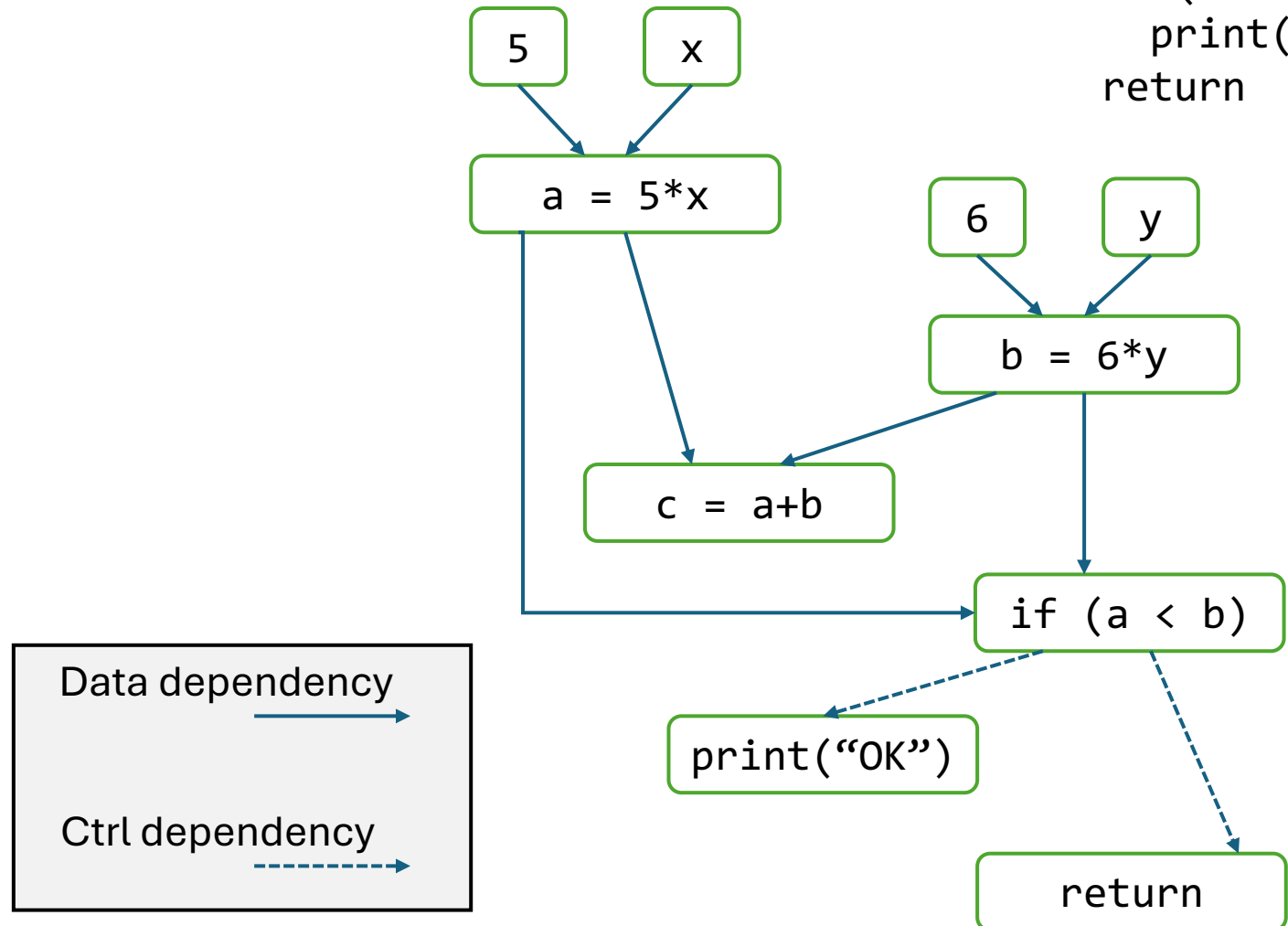
```
a = 5*x  
b = 6*y  
c = a+b  
if (a < b)  
    print("OK")  
return
```



Graph representation 2 - PDG

- **Program dependency graph**
- Rather than order of execution, represent **data** and **control dependencies** between instructions

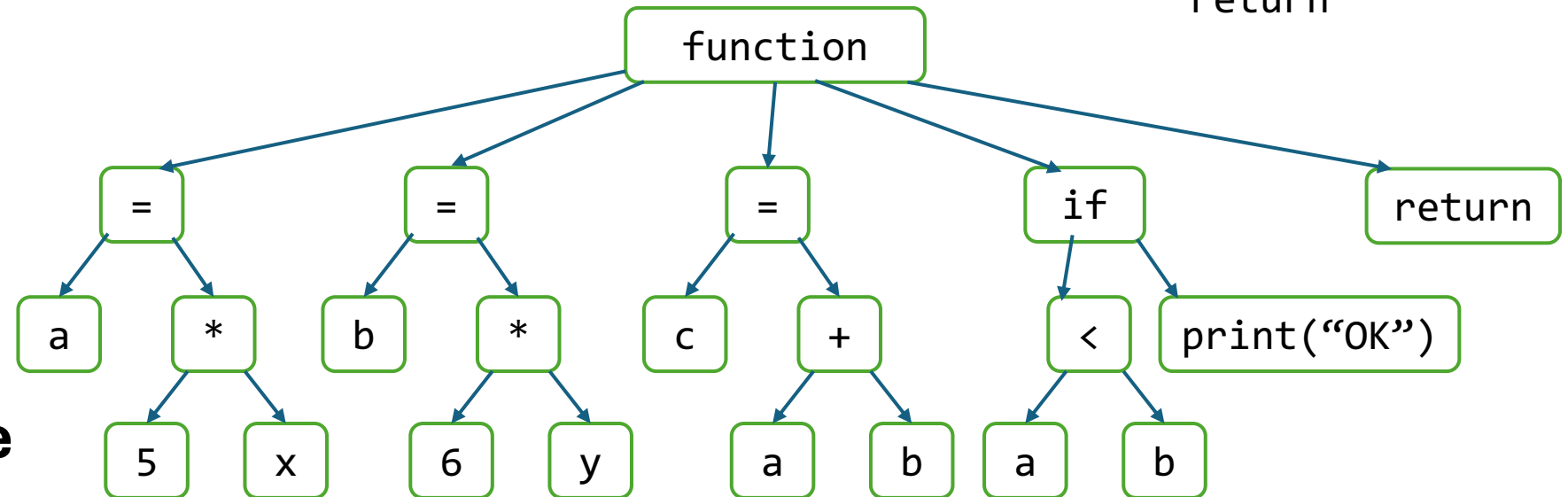
```
a = 5*x
b = 6*y
c = a+b
if (a < b)
    print("OK")
return
```



Graph (tree) representation 3 - AST

```
a = 5*x  
b = 6*y  
c = a+b  
if (a < b)  
    print("OK")  
return
```

- **Abstract syntax tree**
- Represent source (or binary) code more or less directly in **tree form**



Uses?

- **CFG:** many compiler optimizations/analyses
- **PDG:** some analyses/optimization, parallelization
- **AST:** lots of uses! Code normalization, simple static analysis, executing interpreted code

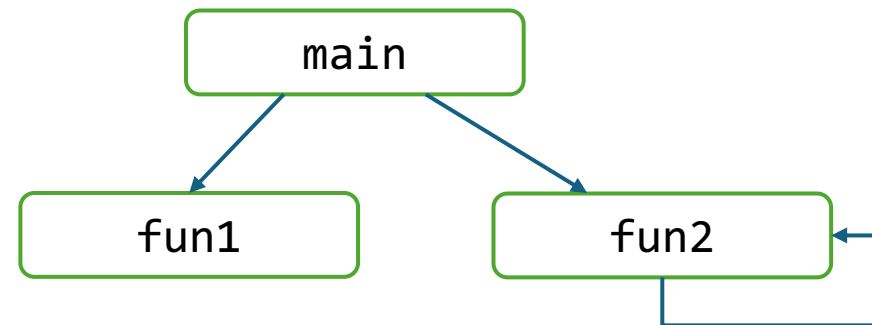
One more thing: call-graph analysis

- Call-graph analysis is an analysis of **which function calls which**
- Its result is a **graph**, with **nodes** being **functions**, and **edges** being **caller-callee relationships**
- Can be generated via **static** or **dynamic analysis**
- **Call-graph construction** via **static analysis** can be complicated if language allows to **pass functions as parameters**

```
def fun1(a):  
    return a*2
```

```
def fun2(p):  
    if p > 0:  
        return fun2(p-1)*p  
    return 1
```

```
def main():  
    fun1(4)  
    fun2(5)
```



Uses?

- Identifying **unused functions**
- Tracking data/control flow **across function calls**
- ...

Why are we talking about this?

- The paper presents a **simple application of static analysis techniques** to the **security of JavaScript programs**
- A **gentle introduction to applying static analysis to security!**

Let's talk about the paper

Mininode: Reducing the Attack Surface of Node.js Applications

Igibek Koishybayev
North Carolina State University
ikoishy@ncsu.edu

Alexandros Kapravelos
North Carolina State University
akaprav@ncsu.edu

Abstract

JavaScript has gained traction as a programming language that qualifies for both the client-side and the server-side logic of applications. A new ecosystem of server-side code written in JavaScript has been enabled by Node.js, the use of the V8 JavaScript engine and a collection of modules that provide various core functionality. Node.js comes with its package manager, called NPM, to handle the dependencies of modern applications, which allow developers to build Node.js applications with hundreds of dependencies on other modules.

In this paper, we present Mininode, a static analysis tool for Node.js applications that measures and removes unused code and dependencies. Our tool can be integrated into the building pipeline of Node.js applications to produce applications with significantly reduced attack surface. We analyzed 672k Node.js applications and reported the current state of code bloating in the server-side JavaScript ecosystem. We leverage a vulnerability database to identify 1,660 vulnerable packages that are loaded from 119,433 applications as dependencies. Mininode is capable of removing 2,861 of these vulnerable dependencies. The complex expressiveness and the dynamic nature of the JavaScript language does not always allow us to statically resolve the dependencies and usage of modules. To evaluate the correctness of our reduction, we run Mininode against 37k Node.js applications that have unit tests and reduce correctly 95.4% of packages. Mininode was able to restrict access to the built-in `fs` and `net` modules in 79.4% and 96.2% of the reduced applications respectively.

products. One of the reasons for its popularity is in Node.js architecture choice. Node.js uses a non-blocking event-based architecture which gives an ability to developers to scale up Node.js applications easily. Nowadays Node.js is used to develop critical systems [49] that require security attention.

Node.js developers distribute community-developed libraries using an in-house built package manager system called NPM. NPM is considered to be the largest package manager by the number of packages [12] it hosts (over million) and growth rate of almost 800 pkg/day [9]. Since 2014, the NPM registry traffic has grown 23,500%, which shows its increasing popularity among developers [47]. This staggering amount of packages hosted in NPM gives developers the power to build apps very quickly by using already implemented functionality by others. In this paper, we argue that overusing third-party libraries comes with its own security risks.

The drawbacks of extensive dependence on third-party packages are: (1) developers need to trust others on the security and maintenance of the libraries; (2) the popularity of NPM makes it lucrative for adversarial users to distribute malicious libraries using attacks such as typosquatting [20, 43, 44], ownership takedown and introducing a backdoor [45, 52]; (3) upgrade or removal of the package from NPM may break the build pipeline of an application [46].

Our study of 1,055,131 packages shows that on average only 6.8% of the code in the application is original code according to source logical lines of code (LLOC) or putting in different words 93.2% of the code in Node.js application is developed by third-parties. One of the reasons why developers

Reasons for discussing this paper

- ...ok, it is technically **not malware detection!**
- But it is a great introduction to **lightweight static analysis** for **security**
- **Meta-goal:** reflect on how static analysis can assist in improving the security of software artifact

Goal/approach of this paper

- Goal: reduce attack surface of Node.js applications
- Approach: lightweight static analysis to identify which components (e.g. which functions), among those present in dependencies of a package, are actually used...
- ... then, remove/prevent access to unused components

Why do this?

- Suppose someone manages to **compromise** a running Node.js application...
- ... then, they'll probably attempt to **build an exploit**
 - Need **capabilities** (e.g., read/write files, send network requests) to do anything useful
 - Normally, attacker can just use existing Node.js functionality, or import extra modules if needed
- With this approach, anything which is not required by the original (unexploited) code is **not available** to the attacker

Attack surface

- By **attack surface** of a system we intend all the **components** of such system that an **attacker can access** in an attempt to **exploit the system**
- “Reducing the attack surface” means **minimizing the components** that the attack can reach
- This is an example of application of the **Principle of Least Privilege**
 - Have you ever heard of it?

Results – package dataset + functionality removal

Job statuses and reasons	Packages
Succeeded packages	672,242
Failed packages	382,889
Package does not have main entry point	188,630
Non-resolvable dynamic import detected	128,533
Failed to install	26,875
Package's main entry point is not CommonJS	20,977
Others	5,013
TOTAL	1,055,131

Table 3: NPM measurement experiment overall status

	Number
Removed fs built-in module	549,254
Removed net built-in module	623,646
Removed http built-in module	606,981
Removed https built-in module	614,030
Percentage of removed JavaScript files	79.1%
Percentage of removed LLOC	90.5%
Percentage of removed exports	90.4%
TOTAL	672,242

Table 4: NPM measurement experiment results

Results – vulnerability removal

Category names	Vulnerable packages	Partially removed	%	Fully removed	%
Prototype Pollution	91,184	5,333	5.85%	3,633	3.98%
Regex Denial of Service	42,163	3,930	9.32%	1,228	2.91%
Denial of Service	21,312	403	1.89%	370	1.74%
Uninitialized Memory Exposure	6,433	690	10.73%	592	9.20%
Arbitrary Code Execution	5,324	413	7.76%	396	7.44%
Cross-Site Scripting	5,142	665	12.93%	590	11.47%
Arbitrary Code Injection	3,451	1,715	49.70%	1649	47.78%
Remote Memory Exposure	3,323	16	0.48%	15	0.45%
Arbitrary File Overwrite	3,240	383	11.82%	381	11.76%
Information Exposure	3,088	47	1.52%	47	1.52%

Table 5: Common vulnerability categories and their reduction results. Some vulnerabilities might not be exploitable since their code is not directly reachable and it might not be possible to chain the vulnerabilities due to additional constrains.

Wrapping up

- Today's paper uses lightweight static analysis to improve the security of Node.js application
- Idea: reduce the **attack surface** of said applications by removing unnecessary components in dependencies
- Results suggest significant reduction in exploitable vulnerabilities in a representative samples of applications

See you next time!