Cross Process and Cross Host Taint Analysis on Socket Programming

Pranjal Dilip Naringrekar
Computing Science
University of Alberta
Edmonton, Alberta, Canada
naringre@ualberta.ca

Abstract

The evolution of computer systems has exposed us to the growing gravity of security threats. Nowadays, various system use the microservice architecture for efficient functioning of their entire application. Thus, an overhead to secure each microservice individually is introduced. Cross program taint propagation is listed among the OWASP's most critical security risks. This paper describes a novel approach to identify taints across applications using an example of socket programming. The approach is divided into two parts. In the first part, I analyze the taints in the program using white listing and various rules. In the second part, I convert the code to an intermediate representation form called call graph using Soot and identify the affected libraries using entry methods function call and also list the solutions to resolve it. If these are left unattended, they could severely impact the system running the software and also the network host. Hence, in order to ensure overall security, I illustrate the identification, impact, analysis and provide solutions to remove the software security vulnerability. The system is evaluated on the basis of how correctly it identifies various entities. The source code is publicly available at GitHub - https://github.com/pranjal080598/Taint-Analysis-on-Socket-Programming

Keywords: Taint Analysis, Socket Programming, Software vulnerabilities, Source code analysis, Call Graph, Intermediate Representation, Network security.

1 Introduction

In this project, I aim to develop a system that will analyze taints for cross process and cross host applications using an example of Socket Programming [13].

1.1 Motivation

The phenomenal growth in the world of internet makes it imperative to test this code during the development life cycle itself rather than after its deployment. Bugs are usually detected when an attack occurs or during the deployment stage of the application. These bugs are fixed using patch codes that would fix one bug, however, may often open several new vulnerabilities, which if left unattended, can pose a significant risk to the system.

Applications (example: ATM, email servers) have various distributed components that perform tasks in synchronisation and hence it becomes important to secure these components as well [15]. Socket Programming is a mechanism used in computer science to establish connection and transfer data between nodes over the network. If the code for either client or server (cross host and cross process) is not written appropriately it may result into multiple security attacks like resource injection, SQL injection and so on [7].

Therefore, the main motivation arises from the fact that various companies still make use of sockets in their coding practise that are sometimes the root cause for various security attacks [12]. Taint analysis tools identify the source and sink methods but does not specify the libraries that will get affected and respective attacks that might occur due to the presence of taint, thus making it difficult for the developer to analyse its impact. For example, if a source taint is generated due to the user input which is propagated all the way to the sink method, it will not only hamper the data transfer but can also cause an attack to various network protocols. This phenomena makes it important to track tainted data across hosts and processes [14] along with affected libraries to help the developers know about the security vulnerabilities beforehand.

1.2 Contribution

To address the problems described in the prior subsection, this paper proposes a novel approach, and realizes it as a framework. Moreover, verification of the practicality is done using real world client server application code. The contribution can be summarized as developing a Java code to analyse taints, detect the affected libraries using Call Graph and provide solutions to mitigate the risk.

1.3 Context

The focus is to develop a system that will be able to identify the taints present in socket programming (across hosts and processes in client.java and server.java), determine the affected libraries using Call Graph and solutions to resolve it.

UoA'21, December 08, 2021, Alberta, Edmonton, Canada
2021.

Client.java will be given as input to the main program that will convert it to Call Graph using Soot's typed 3-address intermediate representation which is suitable for optimization. A taint analysis check will be done to analyse if there exists a code that has user input and is not sanitized. If it is not sanitized it will be marked as tainted. Along with this, the call graph that was created previously will be used to determine the library that is affected due to the presence of this source taint and the resultant possible attack along with its solution. Furthermore, the main program will check if a connection is made to the Server.java code. If found, it will analyse it in a similar way to detect the propagation of taint and resultant sink method.

2 Background and Terminology

Privacy and security of applications is the top measure for any entity utilizing the internet, since the information is present on a digital platform. The distribution of malicious code on web applications in the form of various attacks still persists [4]. Most cyber criminals use legitimate web servers to distribute their malicious code. Therefore, we need an emphasis on protecting the web servers and web applications extensively [5]. Cross process and host taint analysis system can help the developers at a very early stage. It will be able to take the developer's code as input, analyse the code for vulnerabilities and inform the developer if any leaks are present in the code. The developer need not even run its code for finding these vulnerabilities as the system will inform it statically.

Various terminologies are used to address this problem. To secure cross process and cross host applications, taint analysis is performed [8]. Taint Analysis is a field of computer science that checks the flow of user input across the system. The source method deals with the user input directly. If it is an insecure source such as a file, the network, or the user itself then it is marked as tainted. This taint will affect all the variables or access paths in the system that it is connected to. The execution of this taint is called as sink method. If proper sanitization is performed, that is if the user input is checked for vulnerability then taint will not exist, however, skipping this step might result into various security attacks. To check the entry points in the program a call graph is created using Soot. Call graph is a form of intermediate representation that represents the calling relationship between the sub routines and Soot is a program analysis framework.

3 Motivating Example

Sina Weibo is one of China's largest social media platforms with over 500 million users. In March 2020, a security attack took place in which 438 million accounts were impacted. The data retrieved from this attack was sold onto Dark Web [1]. The most common architecture used in such applications is the microservice architecture as shown in Figure 1. Here, the entire application is divided into smaller applications called micro services that are responsible to perform only one task. All micro services are connected to one another. This can cause a major problem, as even if one of the microservice is affected, it might affect the others in the system as well. There are various scans in place that ensure the security of such applications, however, they are not precise and require lot of execution time with respect to global data flow.

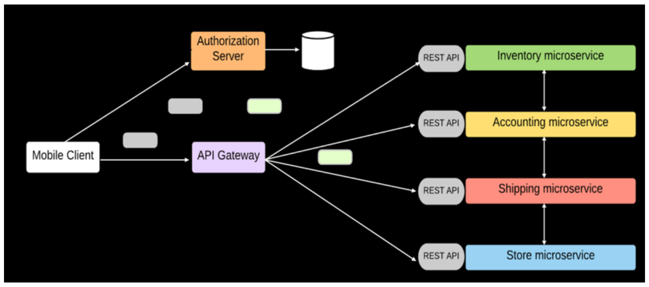
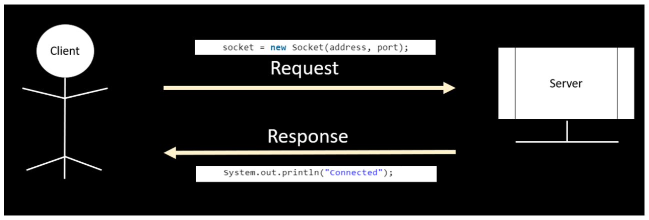


Figure 1. Microservcie Architecture

Therefore, to resolve this issue we have taken an example of Socket Programming. Sockets (nodes) are usually client and server that communicate with one another on network. The communication is initialized by the client by specifying the IP address and port number. If the server receives a valid request from the client a connection is formed as depicted in Figure 2.



```
socket = new Socket(address, port);
```

```
System.out.println("Connected");
```

Figure 2. Client Server Architecture

4 Overview of Approach

As described in Figure 3, the system (developed Java code) will take the developer's developed Client code as input and provide the analysis with respect to taints, affected libraries and provide solutions to the developer in the form of output.

The approach has a linear flow wherein the first step is to analyze the taint present in the code and then construct the call graph to determine the affected libraries and provided solutions.

4.1 Taint Analysis

Taint identification is performed using either white or black listing [10]. In our case, we have used white listing, that is the input only contains allowed content. Set of rules are defined on this white list. There are three categories of rules:

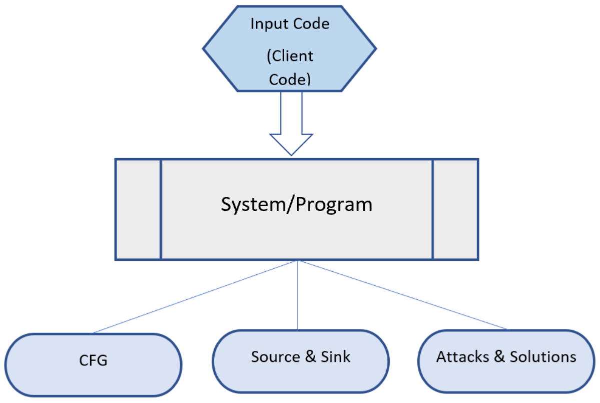


Figure 3. System Flow

- Input Rules: checks if the Java code has any line that requests for user input. Example, new BufferedReader
- Port Sanitization Rules: checks if the user requested variable 'port' is sanitized or not
- Connection Rules: checks if there exists a code in the Client.java that could form connection to Server.java

If first two rules are bleached, it means that there exists a taint in the code and source taint is identified at this stage. If the third rule is satisfied the system will go to the Server.java to analyze the server code as well. Once the white listing and rules are defined we analyze the developer's Client Java code by converting it to a text file and using a linkedlist to keep track of next execution step and branching statements in the code. This helps to cross verify the connection rule. If it is satisfied, we also analyze the developer's Server code in a similar format. This will provide us with the sink method in the code.

4.2 Call Graph Construction

For better analysis, the code is converted into an intermediate representation (IR) form called the call graph. Soot framework is utilized for call graph creation. There are different IRs in Soot called Jimple, Baf, Shimple and Grimp [11]. We will use Jimple for our implementation as it is faster and more precise than other IRs. After downloading and adding the Soot package to the project structure along with other dependencies, Scene.v().getCallGraph() is used to obtain the call graph. We used two call graph construction mechanism to determine its accuracy that is RTA (Rapid Type Analysis) and VTA (Variable Type Analysis) [16].

Along with this, the call graph also helps to determine the affected libraries using Scene.v().getEntryPoints() that determines all entry points to the program. It also provides a type of solution depending on the affected library. For example, if java.lang is affected that deals with user input then the solution provided to the developer is 'Sanitize the inputs'.

5 Evaluation

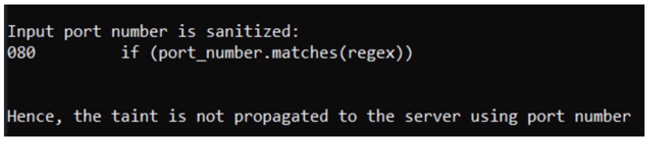
Evaluation will be done on the basis of correct output of the developed java program. The solution focuses on three different concerns:

- Is Taint Analysis done appropriately?
- How is the performance of the Call Graph?
- Are the affected libraries and solutions pertinent?

Hence, the evaluation of the system is classified into three different sub sections to address these questions independently.

5.1 Taint Analysis

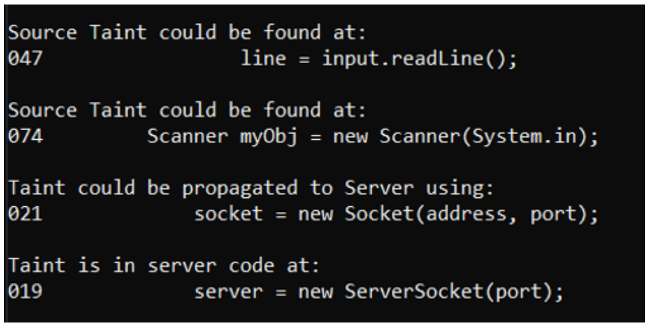
When the program would run with sanitize methods or clean inputs it should not classify anything in the code as tainted as seen in Figure 4, however, if sanitization is removed or if input is not handled appropriately the taints should be clearly mentioned by the output of the program as depicted in Figure 5.



```
Input port number is sanitized:
080        if (port_number.matches(regex))


Hence, the taint is not propagated to the server using port number
```

Figure 4. Sanitized Input



```
Source Taint could be found at:
047                 line = input.readLine();

Source Taint could be found at:
074         Scanner myObj = new Scanner(System.in);

Taint could be propagated to Server using:
021             socket = new Socket(address, port);

Taint is in server code at:
019             server = new ServerSocket(port);
```

Figure 5. Not Sanitized Input

5.2 Call Graph Performance

The call graph is created using Soot's intermediate representation called Jimple. Rapid Type Analysis (RTA) produces 163453 edges and takes 102244 ms to execute. However, Variable Type Analysis (VTA) only fabricated 59541 edges and took 24647 ms to execute. Therefore from Table 1, we can clearly infer that VTA is faster and produces relevant edges in call graph than RTA.

5.3 Affected Libraries and Solution

The call graph creation gives the entry point in the system using Scene.v().getEntryPoints() that helps to determine the

Parameters	Edges	Time
RTA	163453	102244 ms
VTA	59541	24647 ms

Table 1. Call Graph Performance

entry points to the code and thus identify the affected libraries. As depicted in Figure 6, java.lang is the affected library as it dealt with the client input code. For our example, the solution to avoid affected libraries occurrences is to sanitize the input.

6 Discussion

In this section, I will focus on the strengths and limitations of our system. The main strength is low overhead, as the developer just has to specify the file it wants to analyse in the system code and the taint analysis results will be displayed to the developer within fraction of seconds. Another strength of the developed system is that, it has a high code coverage which means it is able to successfully identify the source method in the Client and sink method in the Server code.

The limitation of the developed system is that it utilizes a lot of time to create the call graph. It takes approximately 24647 ms to create a call graph using VTA and even more time using RTA. Along with this, it is incomplete and can produce unsound results when more listings and rules are defined.

7 Related Work

Meghanathan [7] analyzes why the vulnerabilities occur in the code and also discuss the impact of leaving such vulnerabilities unattended. The paper illustrates the identification of taints present at the file reader server socket code which if left unattended could result into various categories of attacks, namely, resource injection, path manipulation, System Information Leak, Denial of Service and Unreleased Resource vulnerabilities. It also identifies potential performance trade offs like increase in code size and loss of features that could arise while incorporating the proposed solutions on the server program. Compared to our solution the paper classifies the attacks into different categories and propose solution to each of them.

To track taint across host and processes, Zavou et al. [14] have developed a system called Taint-Exchange which is built on top of libdft that used lmbench suite for their evaluation. It is a framework used for generic cross-process and cross-host taint tracking. It deals with I/O related system calls that interacts using pipes and sockets. They analyze the taint dynamically, however, I perform a static taint analysis check. The proposed tool does not require extra overhead of setup or maintenance as it can be availed as a plug and play option.

Along with applications, it is also important to track the data through databases to ensure no security threats are exposed on back-end. One such idea called 'DBTaint' was developed by Davis and Chen [3]. This has only been implemented for Perl and Java Web Services. It handles database communications, however, our system handles program communications. Their un-optimized prototype incurs less than 10-15% overhead in the benchmarks that they used for evaluation.

Mandal et al. [6] specify the presence of security vulnerabilities for IOT systems that communicate through different channels using different set of programs. OWASP IoT top 10 most critical security risks involve the presence of taint across cross programs. Thus, it becomes a mandate for all connected programs to pass the security check framework which they have built on top of intra-program taint analysis using Julia Static Analyser.

8 Conclusion

In order to prevent exploitation and to discover the latent software vulnerabilities, I provide a novel approach that is used to analyze taints. These are identified across process and hosts, the developer is signalled of a possible attack and solutions are provided to resolve it by creating the call graph. Since the microservice architecture is too large to analyze, I conducted the experiment on a small code snippet of Socket Programming. The result of the system clearly depicts that the system is efficient and effective. Even though the developed system is written in Java, it can be used in a more generic way and can be applied to other programming language environments.

In future, adding more rules with program slicing will be applied to optimize the framework [2]. Along with Jimple, other IRs can also be utilized to determine performance complexities. Attacks can be classified into various types and visualization can be incorporated. I also plan to work with C/C++ for Windows and Linux platform to analyze network socket programs. To improve the performance of taint analysis, a neural network model that learns from the past input to improve itself and identify taints more accurately can also be used. [9].

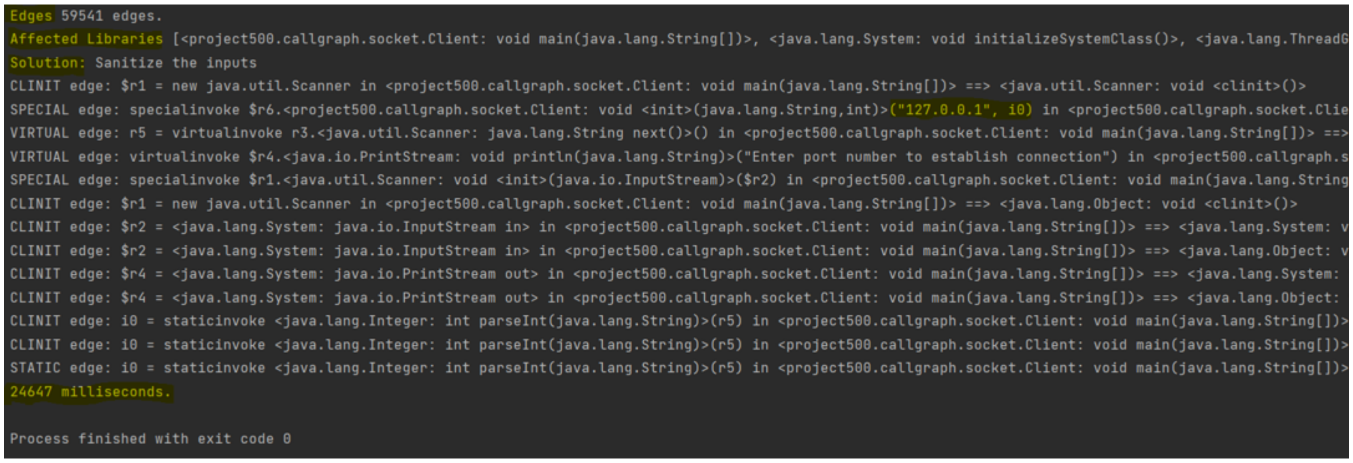
References

[1] Meisam Navaki Arefi, Rajkumar Pandi, Michael Carl Tschantz, Jedidiah R. Crandall, King wa Fu, Dahlia Qiu Shi, and Miao Sha. 2019. Assessing Post Deletion in Sina Weibo: Multi-modal Classification of Hot Topics. arXiv:1906.10861 [cs.SI]

[2] Fabian Berner, Rene Mayrhofer, and Johannes Sametinger. 2020. *Dynamic Taint Tracking Simulation.* 203–227. https://doi.org/10.1007/978-3-030-52686-3_9

[3] Benjamin Davis and Hao Chen. [n.d.]. DBTaint: Cross-Application Information Flow Tracking via Databases .

[4] Julian Jang-Jaccard and Surya Nepal. 2014. A survey of emerging threats in cybersecurity. *J. Comput. System Sci.* 80, 5 (2014), 973–993. https://doi.org/10.1016/j.jcss.2014.02.005 Special Issue on Dependable



```
Edges 59541 edges.
Affected Libraries [<project500.callgraph.socket.Client: void main(java.lang.String[])>, <java.lang.System: void initializeSystemClass()>, <java.lang.ThreadG
Solution: Sanitize the inputs
CLINIT edge: $r1 = new java.util.Scanner in <project500.callgraph.socket.Client: void main(java.lang.String[])> ==> <java.util.Scanner: void <clinit>()>
SPECIAL edge: specialinvoke $r6.<project500.callgraph.socket.Client: void <init>(java.lang.String,int)>("127.0.0.1", i0) in <project500.callgraph.socket.Clie
VIRTUAL edge: r5 = virtualinvoke r3.<java.util.Scanner: java.lang.String next()>() in <project500.callgraph.socket.Client: void main(java.lang.String[])> ==>
VIRTUAL edge: virtualinvoke $r4.<java.io.PrintStream: void println(java.lang.String)>("Enter port number to establish connection") in <project500.callgraph.s
SPECIAL edge: specialinvoke $r1.<java.util.Scanner: void <init>(java.io.InputStream)>($r2) in <project500.callgraph.socket.Client: void main(java.lang.String
CLINIT edge: $r1 = new java.util.Scanner in <project500.callgraph.socket.Client: void main(java.lang.String[])> ==> <java.lang.Object: void <clinit>()>
CLINIT edge: $r2 = <java.lang.System: java.io.InputStream in> in <project500.callgraph.socket.Client: void main(java.lang.String[])> ==> <java.lang.System: v
CLINIT edge: $r2 = <java.lang.System: java.io.InputStream in> in <project500.callgraph.socket.Client: void main(java.lang.String[])> ==> <java.lang.Object: v
CLINIT edge: $r4 = <java.lang.System: java.io.PrintStream out> in <project500.callgraph.socket.Client: void main(java.lang.String[])> ==> <java.lang.System:
CLINIT edge: $r4 = <java.lang.System: java.io.PrintStream out> in <project500.callgraph.socket.Client: void main(java.lang.String[])> ==> <java.lang.Object:
CLINIT edge: i0 = staticinvoke <java.lang.Integer: int parseInt(java.lang.String)>(r5) in <project500.callgraph.socket.Client: void main(java.lang.String[])>
CLINIT edge: i0 = staticinvoke <java.lang.Integer: int parseInt(java.lang.String)>(r5) in <project500.callgraph.socket.Client: void main(java.lang.String[])>
STATIC edge: i0 = staticinvoke <java.lang.Integer: int parseInt(java.lang.String)>(r5) in <project500.callgraph.socket.Client: void main(java.lang.String[])>
24647 milliseconds.

Process finished with exit code 0
```

Figure 6. Affected Libraries and Solution

and Secure Computing.

[5] Yuchong Li and Qinghui Liu. 2021. A comprehensive review study of cyber-attacks and cyber security; Emerging trends and recent developments. *Energy Reports* 7 (2021), 8176–8186. https://doi.org/10.1016/j.egyr.2021.08.126

[6] Amit Mandal, Pietro Ferrara, Yuliy Khlyebnikov, Agostino Cortesi, and Fausto Spoto. 2020. Cross-Program Taint Analysis for IoT Systems. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 1944–1952. https://doi.org/10.1145/3341105.3373924

[7] Natarajan Meghanathan. 2013. Source Code Analysis to Remove Security Vulnerabilities in Java Socket Programs: A Case Study. *International Journal of Network Security Its Applications* 5 (02 2013). https://doi.org/10.5121/ijnsa.2013.5101

[8] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, USA, 317–331. https://doi.org/10.1109/SP.2010.26

[9] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. 2019. Neutaint: Efficient Dynamic Taint Analysis with Neural Networks. arXiv:1907.03756 [cs.CR]

[10] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486

[11] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

[12] Ping Wang, Kuo-Ming Chao, Chi-Chun Lo, and Wun Chao. 2014. Using Taint Analysis for Threat Risk of Cloud Applications. https://doi.org/10.1109/ICEBE.2014.40

[13] Ming Xue and Changjun Zhu. 2009. The Socket Programming and Software Design for Communication Based on Client/Server. In *2009 Pacific-Asia Conference on Circuits, Communications and Systems*. 775–777. https://doi.org/10.1109/PACCS.2009.89

[14] Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis. 2011. Taint-Exchange: A Generic System for Cross-Process and Cross-Host Taint Tracking. In *Advances in Information and Computer Security*, Tetsu Iwata and Masakatsu Nishigaki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 113–128.

[15] Ruoyu Zhang, Shiqiu Huang, Zhengwei Qi, and Haibing Guan. 2012. Static program analysis assisted dynamic taint tracking for software vulnerability discovery. *Computers Mathematics with Applications* 63, 2 (2012), 469–480. https://doi.org/10.1016/j.camwa.2011.08.001 Advances in context, cognitive, and secure computing.

[16] Xilong Zhuo and Chenyi Zhang. 2019. A Relational Static Semantics for Call Graph Construction. arXiv:1907.06522 [cs.PL]